WHAT ALGORITHMS COULD NOT BE

BY WALTER DEAN

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Philosophy
Written under the direction of
Robert Matthews
and approved by

_____

_____

_____

_____

New Brunswick, New Jersey
October, 2007

ABSTRACT OF THE DISSERTATION

What algorithms could not be

by Walter Dean

Dissertation Director: Robert Matthews

This dissertation addresses a variety of foundational issues pertaining to the notion of *algorithm* employed in mathematics and computer science. In these settings, an algorithm is taken to be an effective mathematical procedure for solving a previously stated mathematical problem. Procedures of this sort comprise the notional subject matter of the subfield of computer science known as *algorithmic analysis*. In this context, algorithms are referred to via proper names (e.g. Mergesort) of which computational properties are directly predicated (e.g. Mergesort has running time $O(n \log(n))$). Moreover, many formal results are traditionally stated by explicitly quantifying over algorithms (e.g. there is a polynomial time primality algorithm; there is no linear time comparison sorting algorithm).

These observations motivate the view that algorithms are themselves abstract mathematical objects – a view I refer to as *algorithmic realism*. The status of this view is clearly related to that of Church's Thesis – i.e. the claim that a function is computable by an algorithm just in case it is partial recursive. But whereas Church's Thesis is widely accepted on the basis of several well-known mathematical analyses of algorithmic computability, there is presently no consensus on how (or if) we can uniformly identify individual algorithms with mathematical objects.

In the first chapter of this work, I attempt to illustrate the theoretical significance of algorithmic realism. I suggest that this view is not only of foundational significance to

computer science, but also worth highlighting due to the role algorithms play in the fixation of mathematical knowledge and their relationship to intensional entities such as propositions and properties. Chapter Two presents a formal framework for reducing computational discourse to mathematical discourse modeled on contemporary discussion of mathematical nominalism. Chapter Three is centered on a case study intended to illustrate the technical exigencies involved with defending algorithmic realism. Chapter Four explores various ways in which algorithms might be identified with models of computation. And finally, Chapter Five argues that no such identification can uniformly satisfy all statements of algorithmic identity and non-identity affirmed by computational practice. I suggest that the technical crux of algorithmic realism lies in the necessity of reducing recursive specifications of algorithms to iterative models of computation in a manner which preserves algorithmic identity.

Acknowledgements

This work was composed over the course of too many pages, years and places. I hope that those who know of its many exigencies will thus be heartened my adoption of the following device. I preemptively acknowledge its failure to adequately reflect and distribute the depth and breadth of my gratitude.

I hereby wish to thank and acknowledge the following parties.

My committee:

Robert Matthews (chair), Paul Benacerraf, Rohit Parikh, Jason Stanley.

Others:

Eva Antonakos, Sergei Artemov, Eric Barry, Johan van Benthem, John Burgess, Dorrance Dean, Howard Dean, Max Deutsch, Dick de Jongh, Mel Fitting, Harvey Friedman, Haim Gaifman, Clemens Grabmeyer, Joel Hamkins, Kent Johnson, Karen Kletter, Roman Kossak, Saul Kripke, Hide Kurokawa, Roman Kuznets, Florian Lengyel, Ginny Mayer, Doug McLellan, Richard Mendelsohn, Yiannis Moschovakis, Alex Orenstein, Eric Pacuit, Matt Phillips, Angel Pinillos, Frank Pupa, Bryan Renne, Ellen Schweitzer, John Schweitizer, Liz Schweitzer, Susan Schweitzer, Thor Simon, Simon Thomas, Maxim Smyrni, Chris Steinsvold, Mike Valdman, Susan Viola, Noson Yanofsky, Ann Yasuhara, Stathis Zachos, Domenico Zambella.

Dedication

To Susan Schweitzer for providing an occassion to invoke the `verse` environment:

> Allons! the road is before us!
> It is safe – I have tried it – my own feet have tried it well – be not detain'd!
>
> Let the paper remain on the desk unwritten, and the book on the shelf unopen'd!
> Let the tools remain in the workshop! let the money remain unearn'd!
> Let the school stand! mind not the cry of the teacher!
> Let the preacher preach in his pulpit! let the lawyer plead in the court, and the judge expound the law.
>
> Camerado, I give you my hand!
> I give you my love more precious than money,
> I give you myself before preaching or law;
> Will you give me yourself? will you come travel with me?
> Shall we stick by each other as long as we live?
>
> Whitman, [153]

Table of Contents

List of Figures

Chapter 1

Algorithmic Realism

1.1 Introduction

In the beginning, there were informal mathematical procedures. Descriptions of practical methods for computing quotients, products, roots and areas can be traced to near the beginning of the Sumerian and Egyptian mathematical traditions around 3000 BCE. Computational methods of these sorts were first systematically described by the Greeks and later transmitted to the Arabs who described them algebraically. By the early Middle Ages, classical methods of calculation began to be reintroduced to Europe where they quickly replaced the ad hoc techniques which had been used for calculating with Roman numerals or other non-positional forms of notation. These included a number of methods which are still in use today, including Euclid's greatest common divisor algorithm, the "grade school" long division algorithm and the Russian peasant multiplication algorithm.

The application of computational methods of these sorts arguably represents one of the most fundamental means we have of coming to know mathematical propositions which relate a function to its values – i.e. propositions we would typically state in the form $f(\overline{a}) = b$ such as $510510/17 = 30030$, $59 \times 289 = 17051$ or $gcd(43928, 27149) = 17$. However, throughout much of the history of mathematics, mathematical procedures were not recognized as constituting a class of entities which could be studied in their own right. Much of this has changed with the advent of modern computer science. Within computer science, what were originally treated as informal calculating techniques extraneous to the subject matter of mathematics itself are now referred to as *algorithms*. These entities serve as a focus of study in at least three of its major subfields: the *analysis of algorithms*, *complexity theory* and the *semantics and verification of programs*. In the practice of these fields, it is typical for individual algorithms to be given proper names (e.g. Euclid's algorithm), quantified over (as

in "There does not exist a polynomial time algorithm for deciding whether a propositional formula is a tautology") and treated as the bearers of various computationally significant properties (e.g provable correctness with respect to a function, running time complexity, etc.). For all of these reasons, it seems reasonable to maintain that algorithms constitute the notional *subject matter* of a field such as the analysis of algorithms in much the same sense that we customarily speak of groups forming the subject matter of group theory or topological spaces the subject matter of topology.

This dissertation will be devoted to trying to make sense of what it would mean to take this view seriously in light of traditional concerns about the status of abstract objects in mathematics. In particular, I will be concerned with the view that algorithms correspond to a distinctive class of abstract objects, related to, but potentially distinct from, the mathematical objects on which they operate. This view should be contrasted with the view that algorithms – qua abstract methods of computing – either do not exist or should be identified with the members of some other already recognized class of abstract or concrete objects. I will refer to variants of the former view as forms of *realism* about algorithms and variants of the latter view as forms of *nomialism* or *reductionism* about algorithms. My primary technical interest will be in assessing the viability of the former thesis against the latter two.

My primary motivation for elevating algorithmic realism to an explicit thesis about the foundations of computer science and mathematics derives from the central role which computational methods appear to play in our acquisition of mathematical knowledge – in particular with respect to the verification of simple propositions of the form considered above. In Sections 3 and 4 of this chapter, I will describe the role of algorithms in mathematical practice in detail. And in Chapter 2, I will examine the various ways in which the practice of contemporary computer science appears to take algorithmic realism for granted. But in order to get an initial impression of what is on the table in proposing that mathematical procedures like Euclid's algorithm are themselves abstract objects, it will be useful to first try to characterize the general notion of an algorithm from outside any particular

theoretical framework. To this end, consider the following dictionary definition

> A procedure for solving a mathematical problem (such as finding the greatest common divisor) in a finite number of steps that frequently involves the repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end. [151]

This definition suffers from many of the standard failings of dictionary entries – for instance, it defines an algorithm as being a procedure, without further specifying what makes something a procedure. This is a problem we will frequently encounter in seeking a fully explicit definition of algorithm in terms of elementary or non-procedural notions. However, it turns out that the foregoing definition does a substantially better job at characterizing how the notion of algorithm functions in contemporary mathematical practice than do many standard computer science textbooks, most of which either provide overly general definitions or circumvent matters of definition by citing features of individual examples.[1]

I will thus take the preceding definition as the basis for collecting several common observations about algorithms which will serve to guide my discussion in this chapter:

Algorithms are mathematical procedures Algorithms are typically described as *procedures*, *rules*, *techniques* or *methods* for performing mathematical calculations. While it is difficult to break into this circle of terms from the outside, conventional wisdom suggests that an algorithm is a procedure for operating on mathematical objects in much the same way that a recipe is a procedure for operating on culinary ingredients or a voting method is a procedure for operating on ballots. In particular, each of these sorts of entities is standardly described as taking a class of items as *inputs* (e.g., natural numbers, eggs, marked ballots, etc.) and returning an *output* (e.g., a natural number, completed dish or elected candidate). They are also typically composed of instructions which are expressed as imperative statements (e.g., "Add the values of

[1]The former characterization applies to most popular modern textbooks such as [24], [125], [46] or [57]. A partial exception to this generalization is Donald's Knuth's seminal and encyclopedic survey of algorithmic analysis [72]. Knuth devotes most of the first chapter of the first volume of this work to an attempt to provide a foundationally sensitive analysis of the notion of algorithm. He cites five properties of algorithms which are also present (at least implicitly) in the definition cited above: finiteness, definiteness, input, output and effectiveness. However, not only does Knuth fail to provide a general definition of procedure, several of the specific claims he makes about algorithms (e.g. that an algorithm halts on all inputs) are not in accord with contemporary usage. For both of these reasons, I will postpone a detailed exposition of Knuth's otherwise well-developed analysis until Chapter 5.

n and m," "Beat the eggs," "Count the number of first place votes received by each candidate") and that are *repeatable* in the sense that may be applied to different inputs to yield different outputs on different occasions.

Algorithms are used to solve mathematical problems In almost all instances where we employ a name or definite description to denote an algorithm, this procedure was introduced expressly to solve a mathematical problem (canonically, that of computing the values of a function) in which we had some prior interest. For instance, Euclid's algorithm was introduced to compute the greatest common divisor of given pairs of natural numbers, MERGESORT was introduced to sort lists of natural numbers, Strassen's algorithm was introduced to multiply matrices, etc. In order to use an algorithm to solve a problem it must be carried our or *executed* for a particular input. This typically involves carrying out the constituent steps in terms of which it is specified.

Algorithms are finitary Most authors distinguish algorithms from mathematical procedures in general by requiring that the former but not the latter must satisfy three related finiteness conditions. The first of these is that an algorithm must be finitely specifiable – i.e. describable using a finite number of primitive expressions of which the canonical example is a statement in a programming language. This is meant to rule out mathematical procedures in whose execution each step is mediated by one of infinitely many cases instructions or whose execution depends on evaluating a conditional with infinitely many which must be specified individually. The second standard finiteness condition which is imposed concerns the nature of the primitive instructions themselves. These are typically required to describe *effective* mathematical operations – i.e. operations which can be carried out by a idealized finitary agent using only finite resources. Finally, it is also occasionally required that an algorithm return an output after executing at most finitely many primitive steps for all inputs in its domain. It should be noted that there are examples of procedures which are customarily referred to as algorithms which lack either or both of the latter properties. Thus despite the fact that almost all individual procedures which will be considered

outside of this chapter satisfy all three finitariness requirement, I will not take the latter conditions as essentially attaching to our most general concept of algorithm.

Algorithms involve repetition Most well-known algorithms are related to methods for solving mathematical problems which repeatedly reduce problem instances to other simpler instances. Such a reduction is typically achieved either through the use of *iteration* – i.e., the repeated execution of one or more computational steps as controlled by a parametric or "loop" variable – or *recursion*, i.e. the repeated structural decomposition of a problem instance into a simpler one generated by a predetermine means of structural or mathematical decomposition. While the use of these techniques is not strictly required for a procedure to be counted as an algorithm in practice, virtually all non-trivial examples of algorithms which are used in mathematics use either iteration or recursion. Accounting for the formal properties of these operations thus turns out to be one of the central sources of technical complexity in constructing mathematical models of informally-specified algorithms.

Algorithms are abstract Algorithms are standardly specified linguistically, canonically as sequences of imperative-like statements over a natural or formal language. It is, however, conventional to distinguish between such specifications and the algorithms which they are taken to denote, in much the same way that we typically distinguish between a sentence and the proposition which it is taken to express. For this reason, algorithms are often treated in practice as being *abstract* in much the same sense as propositions, i.e., they are taken to be mind- and language- independent bearers of procedural "meanings."

Again, I will not treat the foregoing list of characteristics as a *definition* of the concept algorithm. But since the characteristics just described are at least typical of the sorts of procedures which are referred to as algorithms in practice, it is reasonable to begin our discussion of algorithmic realism by considering what it would mean for there to exist abstract objects possessing these properties simultaneously. One apparent commitment of this view is that there is an abstract object corresponding to the denotation of each term we employ in practice to make apparent reference to a mathematical procedure. For instance,

when a computer scientist or mathematician makes a statement like "Euclid's algorithm computes the greatest common divisor of pairs of natural numbers" or "Mergesort has running time complexity $O(n \log_2(n))$," the algorithmic realist must presumably maintain that there are abstract objects corresponding to the denotation of the terms "Euclid's algorithm" and MERGESORT. He must additionally hold that there is some sense in which these objects have some or all of the common properties of algorithms just enumerated in addition to those which are explicitly predicated of them. For instance, the algorithm realist must hold that corresponding to the term "Euclid's algorithm" there is an abstract object A_{EUCLID} which not only is a finitely specifiable procedure which operates on pairs of natural numbers but which also may be repeatedly executed so that it outputs single natural numbers. And moreover he must also hold that, suitably interpreted, the operation of A_{EUCLID} may be taken to induce the mathematical function $gcd(x, y)$ which will be defined explicitly below.

In Chapter 2, I will present a general framework for making sense of these requirements in the context of traditional standards for assessing ontological commitment both inside and outside mathematics. For the time being, however, it should be clear that one central burden of the algorithmic realist is to show how there can be a class of abstract objects $\mathcal{A}$ such that for all procedure-denoting terms t, there exists an $A \in \mathcal{A}$ such that t can be taken to denote A. But since it is the realist's view that the reference of t is identical to A, he must additionally hold that A possesses structural features which may be identified with properties which are predicated of t in practice – e.g. properties like having running time $O(\log_2(\min(n, m))$, using recursion, employing a stack, etc. In order to argue for this, he will have to present an argument that properties of these sorts can reasonably be identified with features of the members of $\mathcal{A}$, i.e. that a property like having running time $O(\log_2(\min(n, m))$ can be conceptually *analyzed* in terms of the structural properties possessed by certain members of $\mathcal{A}$. And in addition to this, an algorithmic realist will need to make sure that his assignment of denotations to procedure denoting terms is internally consistent in the sense of comporting with our pre-formal judgments of when distinct procedure-denoting terms t_1 and t_2 refer to the same algorithm, and when they refer to different algorithms.

As will become clear in subsequent chapters, the practice of even isolated fragments of mathematics and computer science already impose sufficiently many constraints on the properties that $\mathcal{A}$ must possess that even defining such a class will be a tall order for potential algorithmic realists. This said, our everyday discourse about algorithms has many of the hallmarks of what might be called "naive realism" about algorithms – i.e. we are willing to speak without hesitation as though algorithms were abstract objects of a certain definite sort. That such a view is indeed characteristic of our current practices is evident from the fact that statements such as the following can readily be found in computer science textbooks and journal articles:

(1.1) a) MERGESORT has running time $O(n \log_2(n))$.

b) There is an algorithm for deciding whether a sentence of propositional logic is valid.

c) There is a polynomial time algorithm for determining whether a natural number is prime.

d) There does not exist an algorithm for determining whether a sentence of predicate logic is valid.

e) There does not exist a comparison sort algorithm with running time less than $O(n \log_2(n))$.

f) The algorithm expressed by the `LISP` program π_1 is the same as the algorithm expressed by the `C` program π_2.

Without going into great detail, each of the foregoing sentences instantiates a linguistic form whose grammatical acceptability (potentially within a specialized practice) is traditionally thought to signal ontological commitment. For instance, sentence (1.1a) contains an (apparent) proper name (i.e., MERGESORT) which serves as the (apparent) subject of a subject predicate statement. Sentences (1.1b,c,d,e) all contain quantifiers whose (apparent) ranges are either the class of *all* algorithms (1.1b,d) or a subclass thereof (1.1c,e). And sentence (1.1e) has the (apparent) form of an identity between two complex singular terms which are claimed to denote the same item.

It goes almost without saying that the acceptability of statements analogous in form to those in (1.1) containing terms making (apparent) reference to or quantifying over entities like sets, numbers, propositions, persons, colors, etc. have attracted a great deal of attention among philosophers. It is, for instance, the acceptability of statements about natural numbers analogous in form to (1.1a) and e) (e.g., "17 is prime," "The number of Supreme Court justices is equal to the number of planets") which motivated Frege [39] to claim that natural numbers not only existed but had the logical type of objects. And the acceptability of sentences about propositions analogous in form to (1.1b) or d) (e.g. "Bush believes all the propositions Cheney believes," "There is something which Libby knew which Fitzgerald did not") has motivated a number of elaborate programmes which seek to develop the view that propositions must be treated as abstract objects (e.g. [119], [25], [6]).

As I will examine further in the next section, questions about the ontological status of algorithms bear a number of affinities to analogous questions about the status of both natural numbers and propositions. Such parallels aside, however, what is potentially most surprising is how *little* attention statements like (1.1) have received from not only mathematicians and computer scientists, but also logicians and philosophers of mathematics. For there are a variety of well known proposals which attempt to elucidate the nature of natural numbers and propositions, most famously by attempting to identify them with the members of classes $\mathcal{N}$ or $\mathcal{P}$ consisting of various sorts of pure or impure set theoretic structures. But as matters currently stand, very little thought has been put into how to define an analogous class $\mathcal{A}$ of abstract objects which can reasonably be taken to correspond to algorithms.[2] There is thus currently no consensus on how, or if, this can be accomplished. And consequently, questions about the ontological status of algorithms – and in particular about the sustainability of algorithmic realism itself – remain largely unsettled.

Even though my ultimate intention in this work will be to argue *against* algorithmic realism, the lack of a broad philosophical (or even foundationally oriented) literature about

[2]There are two partial exceptions to this in the form of programmes of Yainnis Moschovakis and Yuri Gurevich, both of which attempt to provide a definition of $\mathcal{A}$ which is grounded in foundational considerations about the general notion of algorithm. However neither of these proposals is widely known or accepted. And since a variety of conceptual and technical desiderata need to discussed before these proposals can be properly evaluated, I will postpone a detailed consideration of these proposal until Chapter 6.

algorithms means that part of my burden will be to illustrate why algorithmic realism is a significant view and also to scout out ways in which it might be developed which are sympathetic to its goals. Note, however, that while there is very little literature which directly addresses ontological concerns about algorithms, the general notion of effective mathematical procedure is invoked in a variety of different theoretical contexts for a variety of different purposes. This is obviously most apparent in branches of theoretical computer science such as the analysis of algorithms of which, as I have already observed, algorithms form the presumptive subject matter. But it is easy to see that an abstract notion of procedure possessing most or all of the properties described above is also an important part of the theoretical inventory of other fields such as constructive and intuitionistic mathematics, cognitive science, functionalism in philosophy of mind and language, and various subfields of economics and social choice theory such as game theory, mechanism design and voting theory.

A truly general survey which is sensitive to the detailed role played by mathematical procedures in each of these disciplines would be quite wide-ranging. For not only would such a survey need to classify and synthesize the role of procedures in the context of a variety of different theoretical developments, but it would also have to account for cases in which there was not unanimity within a given field about their status. Sorting out details of this sort would thus require not only that we enter into a detailed examination of the theoretical commitments of each of the fields just mentioned, but also in some cases that we sort out tensions between different views about their foundations. And since this would be a major undertaking in its own right, in the sequel I will for the most part treat questions pertaining to the ontological status of algorithms largely as *sui generis* problems arising in computer science and, to lesser extent, in classical mathematics.

But although this decision will help to streamline my presentation below, it has the unfortunate consequence of ignoring many of the theoretical applications of the notion of algorithm which are likely to be most familiar to philosophers. Among these are the use of a notion of abstract algorithm to define an intermediate level of psychological explanation in cognitive science and philosophy of mind (cf., e.g., [110], [83]), to give the meanings of the propositional connectives in intuitionistic logic (cf., e.g, [143]) and to state the truth

conditions of sentences in the context of verification theories of meaning (cf, e.g., [30]). As these are conceptually distinct applications of the general notion of algorithm, it is likely that these and other theoretical developments impose a variety of demands on a general theory which seeks to regard algorithms as abstract objects in their own right. And a detailed examination of these fields would be likely to yield a rich array of adequacy conditions which constrain any such theory that is adequate for their foundational purposes.

While I do not wish to downplay the significance of these constraints, there are reasons beyond exegetical expediency which advocate in favor of limiting the set of theoretical desiderata that will constrain my discussion of algorithmic realism. Primary amongst these is that while it is easy to find evidence of realism about algorithms in the foundational literature of a variety of different subjects other than computer science and mathematics, it is rare to find detailed discussion about the nature of algorithms themselves or what is entailed by regarding them as abstract objects. What is most common in cognitive science and philosophy of mind is to simply assume that theoretical computer science already provides a theoretically perspicuous notion of algorithm.[3] And what is most common in intuitionistic and constructive mathematics is to either treat the notion of algorithm as a conceptual primitive in terms of which other notions (such as that of intuitionistic proof) are defined, or alternately to replace it with one of several technical notions which are motivated by epistemic concerns which are largely extraneous to any conceptual analysis of the notion of algorithm itself.[4]

Although much more can be said about how algorithms function as a sort of theoretic posit in the practice of cognitive science and intuitionism, it is doubtful that a more careful examination of these fields will will shed substantial light on the sorts of basic ontological questions which directly bear on algorithmic realism. In the sequel, I will thus take classical

[3]This is, of course, something of a simplification. Among book length treatments, for instance, Pylyshyn [110] and Harnish [58] both discuss the general notion of algorithm in detail as well proposing constraints which are proposed by an independently motivated theory of so-called *cognitive architecture*. However both this notion and the general notion of algorithm they employ are explicitly motivated by a series of analogies to notions drawn from theoretical computer science, most notably those of a machine model and a formal programming language.

[4]The former route is adopted by theorists like Dummett [31] and Martin-Löf [84] who favor a linguistic or epistemic characterization of intuitionistic truth, while the latter is adopted both in the constructive mathematics practiced by the Markov school [82] or intuitionistic mathematics based on the well-known realizability semantics introduced by Kleene [65] (also cf., e.g., [142]).

mathematics and computer science as both the primary sources of conceptual and technical constraints which bear on the form a foundational theory of algorithms might take as well as the primary contributors of formal models of the sort which might inform such a theory. I have already noted the rationale for choosing to look on work in computer science in this manner: not only do individual algorithms form the presumptive matter of one of its major subfields (the analysis of algorithms), but the general notion of algorithm described above has informed the development of several of its other branches (computability theory, complexity theory, programming language semantics, formal verification, etc.). Theoretical computer science thus provides not only the more clearcut cases of apparent ontological commitment to algorithms, but also the precisely articulated bulwark of theoretical constraints that a general theory of algorithms would have to satisfy.

In comparison to the relatively straightforward theoretical role which algorithms appear to play in computer science, the status of algorithms within classical mathematics is more complex. For note that unlike intuitionistic mathematics (wherein mathematical objects are often identified with so-called *mental constructions* which have an overly procedural character), the subject matter of classical mathematics is often described in terms which explicitly forbear a procedural understanding of mathematical objects. This is to say that mathematical objects conceived classically correspond to structures which are held to be both abstract (in the sense of existing outside of space and time) and extensional (in the sense of existing independently of means by which we might grasp or describe them). But as I will discuss further below, algorithms are standardly understood as temporal (in the sense that they operate within time, at least in an abstract sense) and intensional (in the sense of corresponding to means by which extensional functions or properties are grasped or determined). It thus reasonable to think that whatever metaphysical view we adopt about mathematical objects in general, there will be no reason to say anything about the status of algorithms in particular for the simple reason that they do not appear to constitute part of the domain of classical analysis or number theory.

For this reason, the relationship between mathematical practice and algorithmic realism may initially seem obscure. But as I will discuss at greater length in Sections 3 and 4, the algorithm contributes to contemporary mathematics not by serving to fix a new domain of

abstract objects which are themselves the subject of mathematical investigation, but rather by serving as provably sound or correct methods by which we come to know various facts about other mathematical objects. In other words, although algorithms may not contribute to the ontology of contemporary mathematics, they form an important part of the epistemic apparatus which allows us to discover which mathematical statements are true.

The fact that algorithms serve this epistemic role in mathematics is also reflected prominently in computer science. For note the study of individual algorithms has historically been motivated primarily by a desire to discover new means of deriving novel mathematical statements in traditional subjects like graph theory or abstract algebra, not by a desire to discover or analyze the properties of procedures in their own right. For instance, it was a preexistent desire on the part of mathematicians to multiply large matrices which ultimately lead to the discovery of efficient (i.e. $< O(n^3)$) matrix multiplication algorithms such as that of Strassen [137]. And thus although a wide array of distinctive techniques and formal methods has been developed by computer scientists to study algorithms on their own, much, if not all of this research is aimed at developing and verifying computational methods which are ultimately employed in mathematics.

As consequence of such research, computational methods have come to play a significant role in areas of mathematics such as number theory or topology which have traditionally been conceived of as abstract and lacking in concrete computational content. The extent of this role started to be acknowledged by philosophers of mathematics somewhat belatedly as a consequence of the original Haken-Appel proof of the Four Color theorem [2], [3]. As is well-known, this proof involved an exhaustive case analysis of over 1400 distinct graph configurations which was originally conducted by the use of a graph theoretic algorithm – call it A – which attempted to reduce each of these configurations G to a simpler planar graph G' with the property that G' is four-colorable if and only if G is. Due to the large number of cases which had to be surveyed, Haken and Appel implemented the process of generating the configurations and applying A by writing a computer program π which was then implemented on a digital computer P. The output of the execution of π on P (which indicted that each of the 1400 configurations reduced to one which was four-colorable) was then cited as a step in the proof of the theorem.

As is also well-known, the publication of this proof was considered controversial by both the mathematical and philosophical communities for a number of reasons. Among them was the fact that an essential part of the Haken-Appel proof which consisted in the case analysis described above which was carried out by a physical computing device – namely, the computer P. On this basis Tymoczko [145] famously argued that in accepting the Four Color theorem on the basis of the Haken-Appel proof, we were essentially expanding the canons mathematical justification. In particular, Tymoczko argued that in accepting a proof partly consisting of a report on the behavior of the P, we were allowing that a variety of empirical statements concerning P could serve as premises to a legitimate mathematical proof. These premises would, among other things, describe the initial physical configuration of P (e.g. how the program π was stored in its registers) and also its physical evolution during the course of its operation. Tymoczko argued that as we cannot know that these premises are satisfied *a priori*, the Haken-Appel proof confers a different "quasi-empirical" sort of justification on its conclusion than do traditional mathematical proofs.

Tymoczko has subsequently been taken to task by a number of other commentators for misidentifying the moral of the Haken-Appel proof and its subsequent acceptance by the mathematical community.[5] For present purposes, however, I do not so much wish to take issue with the conclusion he draws about the status of Haken-Appel proof, but rather to point out that the attention paid to this case has the unfortunate consequence of causing many commentators on mathematical knowledge to either downplay or overlook the distinctive role played by algorithms in mathematical demonstrations at large. But it is precisely in the context of this proof which general questions about the nature and status of algorithms have come to the surface in the recent literature of philosophy of mathematics. In order to delimit yet further the range of issues with which we will have to be concerned in subsequent chapters, it will thus be useful to conclude this introductory section by briefly outlining why I believe that questions about the ontological status of algorithms typically supersede their physical implementation in the context of discussion about mathematical knowledge.

[5]See in particular Detlefsen and Luker [28] for a treatment which is largely in line with the interpretation adopted herein.

The fundamental observation in this regard is that before any algorithm A can be justifiably used in order to derive a purely mathematical conclusion, A must be proven *correct* within classical mathematics itself. I will have much more to say about what this means in Section 4, but for the time being the central point can be understood by considering a simple example concerning prime numbers. To this end, let p denote the characteristic function for the set $\{n \in \mathbb{N} \ : \ n \text{ is prime}\}$ (i.e. for all n, $p(n) = 1$ if n is prime and $p(n) = 0$ otherwise). Now consider the statement

(1.2) $p(170141183460469231731687303715884105727) = 1$

which expresses the fact that the natural number 170141183460469231731687303715884105727 (which is equal to $2^{31} - 1$) is prime. This statement was originally shown to be true by Lucas in 1879 without the use of any form of mechanical computing machinery. However Lucas' demonstration did make use of an algorithm which has come to be known as the *Lucas-Lehmer primality test*. This method allows us to determine whether a Mersenne number n (i.e. an integer of the form $2^p - 1$ for prime p) is prime by computing a sequence of numbers $S_0 = 4$, $S_{i+1} = S_1^2 - 2$ for $i \leq 2$. Upon computing this sequence, the algorithm tells us that we should output 1 (indicating that this number is prime) just in case S_{p-2} is divisible by $2^p - 1$ and 0 (indicating that it is not prime) otherwise. Call this algorithm L.

The fundamental observation mentioned above can be applied to this case as follows. Given that Lucas derived (1.2) on the basis of carrying out the algorithm just described for $p = 31$, he would not have been justified in claiming that this statement was *true* (or equivalently that he *knew* (1.2)) unless he had also been able to demonstrate that the algorithm L was reliable in the following sense:

(1.3) For all primes p, the result of applying L to p is 1 if and only if $2^p - 1$ is prime.[6]

[6]The "only if" direction of this biconditional expresses what is traditionally described as the *correctness* of L – i.e. the claim that L is sound in the sense that if it outputs 1 on input p, then $2^p - 1$ really is prime. The "if" direction, on the other hand, expresses that L is *complete* in the sense that if $2^p - 1$ really is prime, L will produce output 1 on input p. Since in this case the correctness of L is consistent with the case in which it always outputs 0 or produced no output at all, correctness and completeness not only are distinct requirements but both are required to ensure that L is usable in practice. In the typical case we will consider below, we will speak of proving an algorithm A correct with respect to a function f. In this case, correctness and completeness are both expressed by the statement "for all $x \in dom(f)$, the result of applying A to x is equal to $f(x)$."

The necessity of proving such a statement before concluding on the basis of the fact that L outputs 1 for input 31 as indicating that (1.2) is true ought to be apparent from the description of L alone. For note that since this description is not transparently related to the traditional definition of primality, we possess no warrant for using this procedure to test $2^p - 1$ for primality until such a result were to be demonstrated.

The foregoing observations appear to apply regardless of whether L is to be carried out by computer or "by hand" (as in the case of Lucas's original derivation). For note that even though the latter sort of derivation is transacted by a human mathematician (and hence, we may assume, will not depend on the same sort of extra-mathematical hypotheses which are required to ensure the correct outcome of a computer derivation), there will be no warrant for believing in its conclusion unless L has previously been proven to have the property expressed by (1.3). And as such, the availability of the same sort of correctness proof for the algorithm A employed in the Haken-Appel proof is also a prerequisite for demonstrating that our belief in its conclusion is also justified.

The general question of whether and how we can prove that an algorithm A is correct with respect to an antecedently-defined mathematical function f is thus the issue which arises when we wish to use A to derive a statement about the values of f. And since we have been using such methods in mathematics for millennia, the necessity of asking this question both predates Tymoczko's particular concerns about the status of the Haken-Appel proof and also has a considerably wider significance with respect to the epistemology of mathematics. In Section 4, I will argue in detail that the necessity of proving algorithms correct before they can be used in mathematical demonstration represents the single most significant respect in which classical mathematics appears to be invested in algorithmic realism. For as I will argue there, it is precisely in the context of constructing correctness proofs that we must construct abstract mathematical representations of individual algorithms. And this at least appears to suggest that mathematics is implicitly committing to recognizing that, like the items on which they operate, individual algorithms are mathematical objects in their own right.

However, in order to understand what is involved in such a claim, we must first get some impression of what it means to regard an individual algorithm as an object of any sort. The

relevant considerations here derive from a diverse range of sources, including traditional metaphysics, the philosophy of language and of mathematics, as well as the theoretical practices of mathematics and computer science themselves. A thorough exposition of this subject must hence await subsequent chapters. However as I hope to demonstrate in the next section, some purchase on what it means to regard an algorithm as an abstract object can be gained by recording a number of systematic affinities which discourse about algorithms bears to discourse about other traditionally recognized classes of abstract objects. And it is to this task that I now turn.

1.2 Algorithms and formal ontology

In order to best understand the claims of algorithmic realism and also to motivate the sort of theory which I believe has the best chance of vindicating such a view, it will be useful to begin by considering some reasons why we do not at present possess a foundational theory of how procedures "fit in" relative to the classes of abstract entities whose existence is traditionally recognized in philosophy and mathematics. One way to begin to get a handle on this is to note that it is traditional to compare algorithms to two different sorts of intensional entities: propositions, and functions in intension. Of course since these notions are themselves often taken to be unclear or otherwise problematic, it is doubtful whether prior work in philosophy of language or intensional logic can be of direct help to the algorithmic realist. But at the same time, it also turns out that there that are a number of operational affinities between algorithms and how these other entities are traditionally described. And since it will often be useful to allude to them at various points, it will be useful to go about making these parallels explicit.

I have already noted a number of similarities which are often drawn between algorithms and propositions: both sorts of entities are standardly described as being specifiable by another class of linguistic expressions which are taken to express them. In the case of propositions, this other class is taken to correspond to sentences of either a natural or a formal language. And in the case of algorithms, this class may include programs expressed in either a formal programming language such as `C` or `LISP` or, more broadly, sequences of instructions written in a natural language idiom known as *pseudocode* of which more will

be said below. There is thus canonically a many-one relationship between a proposition P and the class of sentences $\varphi_1, \varphi_2, \ldots$ which express it, just as we will see that is also typically a many-one relationship between an algorithm A and the class of programs $\pi_1, \pi_2, \ldots$ which express it. As noted above, it is for this reason that both propositions and algorithms are commonly taken to be *abstract*. For both sorts of entities are the kinds of things which not only must be referred to *indirectly* (i.e., by constructing a sentence or program which *expresses* them) and are also such that they maybe specified in different ways (i.e., by *distinct* sentences or programs). Thus in neither case does it seem reasonable to *identify* a proposition with a sentence or an algorithm with a program.

If we now attempt to explain further what it means for a proposition to be an object unto itself, we arrive at the customary characterization that such an entity is a language-independent entity which intrinsically carries the meaning of the sentences which express it. As such, propositions are typically assigned the theoretical role of being the primary bearers of truth and falsity (sentences having these properties only in the derivative sense of expressing propositions). However, they are also traditionally conceived as *intensional* entities in the sense that if we wish to speak of them (as did Frege [37]) as *denoting* a truth value, then we must recognize that many non-identical propositions – e.g., those expressed by "There are eight primes less than 20" and "There are eight planets" – denote the same truth value. For this reason, propositions are also standardly taken to correspond to objects of propositional attitude such as belief and knowledge (for note that we famously cannot infer from "S believes that there are eight primes less than 20" that "S believes that there are eight planets").

If we similarly try to think in abstract terms about what it means to treat algorithms as objects, we may note that they possess some, but not all, of the features traditionally associated with propositions. On the one hand, we will see in the next chapter that it is largely consistent with the methodology of computer science to characterize algorithms as being programming-language-independent entities which bear the meaning of the programs which express them. Consistently, however, with the comparison of programs to sequences of *imperative* statements in natural language, it is not customary to speak of a programming language as expressing statements which are either true or false. Thus there is no precedent

for claiming either that algorithms are bearers of truth or falsity, or that they denote truth values.

Further reflection reveals that the system of analogies between propositions and algorithms can be extended if we adopt the popular view that propositions are *structured* entities. For suppose that we regard a proposition P as being composed of constituents which are composed of components corresponding to the grammatical structure constituents of a sentence φ by which it is expressed. For instance, in the case that φ has the form $\psi(a)$ (where $\psi(x)$ is a predicate and a is a singular term), we may take P to correspond to a structured complex formed from an entity $\boldsymbol{a}$ which serves as the argument to (or *saturates*) an entity corresponding to ψ.

Although the view that propositions must be taken to be structured objects is itself quite standard, there is no equally standard view about the sorts of entities to which $\boldsymbol{a}$ and $\boldsymbol{\psi}$ must correspond. For present purposes, however, let us also adopt the Fregean view according to which for all linguistic expressions α, we must distinguish between their *reference* $\underline{\alpha}$ and their *sense* $\overline{\alpha}$ (where, of course, the reference of an expression is what it denotes – e.g. an object in the case of a name, a set (let's say) in the case of a predicate – and its sense is a mode of presentation of its reference). On the Fregean view, the components of a proposition (or what he referred to as a *thought*) are the sense of the grammatical constituents of the sentence which expresses it. So in the case under consideration, $\boldsymbol{a}$ should be taken to be $\overline{a}$ – i.e., a means of picking out $\underline{a}$, potentially, as the object satisfying a complex description customarily associated with a – and $\boldsymbol{P}$ should be taken to be $\overline{P}$, i.e., per Dummett [29], a means of deciding for an arbitrary object, whether it is a member of the set $\underline{P}$.

If we adopt this familiar (although largely schematic) view of structured propositions, it is now possible to extend our previously stalled discussion of the relationship between algorithms and propositions. For consider the logical-cum-grammatical role procedure-denoting terms typically play in sentences. As I will discuss at greater length in Chapter 2, a term like EUCLID (which purportedly denotes Euclid's algorithm) canonically appears in two grammatical frames, one of which is exemplified by a sentence of the form

(1.4) $\textsc{Euclid}(\texttt{6647}, \texttt{5491}) = \texttt{17}$.

Note that in this sentence, the term EUCLID does not function as a grammatically well-formed sentence on its own (either declarative or imperative). Rather it plays the role of a so-called *functional expression* – i.e. an expression like $\sin(x)$ or $x + y$, which denotes a function – and in (1.4) it acts as an (apparently proper) name. On the basis of this observation, there should be no expectation that the item denoted by the expression EUCLID itself either bears or denotes a truth value. Rather, the form of (1.4) suggests that the reference of EUCLID should be taken to be a function of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, i.e. the sort of entity which may be applied to pairs of natural numbers (such as those denoted by the numerals `6647`,`5491`) so as to return another natural number.

There is, however, a problem with this view unless we say something additional about the general notion of function. For note that on the standard modern understanding, a function $f : X \rightarrow Y$ is a set of ordered pairs which is a member of the set X^Y. On this view, the denotation of a functional expression of type $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, such as $x + y$, is simply the set $p = \{\langle n, m, q\rangle \ : \ n, m, q \in \mathbb{N} \wedge n + m = q\}$. And from this, it would follow that we ought to assign as the denotation of EUCLID the set $g = \langle\langle n, m, q\rangle\rangle n, m, q \in \mathbb{N} \wedge gcd(n, m) = q$ (where $gcd(x, y)$ denotes the greatest common divisor function). But now note that it is easy to define other algorithms A which also compute the function g, meaning that a statement of the form $A(\mathtt{n}, \mathtt{m}) = \mathtt{q}$ will be true just in case a statement of the form $\textsc{Euclid}(\mathtt{n}, \mathtt{m}) = \mathtt{q}$ is true. One such algorithm is the function NAIVEGCD which I will discuss in Section 4.

But now consider another grammatical frame in which the expression EUCLID may also appear:

(1.5) EUCLID has running time $O(\log_2(\min(n, m))$.

In this instance, it may appear as though EUCLID is functioning as a proper name instead of a functional expression. This corresponds to the view I will ultimately adopt in Chapter 2. However, the mere grammatical acceptability of (1.5) is not enough to yield this conclusion. For note that it is routine in mathematical practice for one and the same functional expression to appear *polymorphically* – i.e. as a grammatical constituent of two different logical types. This is witnessed, for instance, by the acceptability of both

(1.6) a) $\sin(\pi) = 0$,

b) $\sin(x)$ is continuous.

According to the standard analysis of these sentences, $\sin(x)$ would be taken to denote the same function (i.e., set) $s \in \mathbb{R}^{\mathbb{R}}$ in both cases. In the case of (1.6a), this expression is applied to the constant π, yielding a complex functional expression which denotes a number. This sentence is true because the value of this functional expression (as determined by applying s to the denotation of π) is equal to the denotation of the constant `0`. And in the case of (1.6b), $\sin(x)$ functions as an argument to the predicate "is continuous." This sentence is true because the set s is a member of the denotation of this predicate (i.e., the set of sets corresponding to continuous functions on $\mathbb{R}$).

This leads us to ask whether the same analysis can be applied in order to provide a uniform account of the role of EUCLID in statements (1.4) and (1.5). To see that it cannot, it suffices to consider the additional statement

(1.7) NAIVEGCD has running time $O(\log_2(\min(n, m))$.

As we will see in Section 4, this statement is *false*. Per my original characterization of the manner in which an algorithmic realist must treat a statement of this form, this must be taken to mean that whatever object is taken as the denotation of the term NAIVEGCD fails to have the property expressed by "has running time $O(\log_2(\min(n, m))$." But according to the current proposal, this object is just the function g which we are also taking as the denotation of EUCLID. It should follow, however, from this fact together with (EUCLIDGRAMMAR2) (which we will see to be true in Section 4), that (1.7) is *true*.

Philosophical sophisticates will immediately recognize the foregoing as a standard case of substitutivity failure. In somewhat more detail, if we assume that algorithm-denoting terms like EUCLID denote functions (in particular, the functions which we would standardly say are *computed* or *induced* by the algorithm denoted by the term), then we ought to be able to conclude that both EUCLID and NAIVEGCD denote the function g. But from this, we ought to be able to infer (1.7) from (1.5) in conjunction with the identity EUCLID = NAIVEGCD.) But since this latter statement is not true, something must have gone wrong with the argument.

On the model of many other well-known instances of substitutivity failure, the foregoing

consideration would standardly be taken to demonstrate that the denotations of procedure denoting terms – at least as they occur in statements like (1.5) and (1.7) – cannot be taken to be functions. Note, however, that these statements do not, at least on the face of things, appear to introduce the sort of semantic environments in which substitutivity failures are known to occur. In particular, they contain neither propositional attitude verbs nor modal operators of the sort which are standardly taken to flag so called *intensional contexts* – i.e. grammatical frames $\Phi(x)$ in which substitution of co-denoting expressions does not preserve truth value.

In treating cases of this sort it appears that we have two options. On the one hand, we can take expressions of the form "$\_\_is/has\Psi$", where Ψ denotes a property of procedures (such as having running time $O(\log_2(\min(n, m)))$, as introducing a new class of intensional contexts. And on the other, we can reject the claim that a procedure-denoting term like EUCLID denotes the function which this procedure computes in favor of assigning it some other form of denotation. On the basis of a combination of conceptual and technical considerations, I will argue in Chapter 2 that the latter option provides for a more uniform interpretation of procedural discourse of the sort we have been considering and is thus the preferable option.

But note that regardless of how we resolve this logical issue, we are still left with the problem of saying what sort of item must be assigned to the terms EUCLID and NAIVEGCD as they appear in (1.5) and (1.7) so that the former is true and the latter is false. If we adopt the former proposal and regard these statements as composed of names and matrices introducing intensional contexts, this questions amounts to inquiring after the *sense* of EUCLID and NAIVEGCD. However since we are going to proceed in the manner announced in the previous paragraph, this question amounts to inquiring after their *reference*.

As I suggested in Section 1, accounting for the reference of such expressions is the question which the algorithmic realist must ultimately answer. And while the foregoing considerations do not bring us any closer to answering it directly, they at least suggest another avenue to consider. For although we have seen that it is not viable to identify the reference of EUCLID and NAIVEGCD with *a function* when this notion is understood in the now standard set theoretic sense, there is another traditional way of explicating this

notion. In particular, while it is common to all understanding of the notion *function* that a function f with domain X and range Y is mapping that takes members of X as arguments and yields members of Y as values, the notion of such a mapping was not always understood *extensionally* – i.e. identified as a set theoretic correspondence given by a subset of $X \times Y$.

As I will discuss somewhat further in Section 3, it is generally said that the original notion of function employed in mathematics was *intensional*. On this understanding, a function $f : X \rightarrow Y$ is understood as being something closer to a *rule* which induced a mapping from X to Y – i.e. some means of taking members of X and constructing or deriving members of Y. In canonical cases, such rules were given by expressions of certain privileged forms – originally polynomials, but later expressions formed transcendental functions, differentials and infinite sums and products. For note that, at least to a first approximation, an expression such as a polynomial $a_1x^n + a_2x^{n-1} + \ldots a_{n-1}x + a_n$ can be seen as a rule for mapping real numbers to other real numbers by applying the operations of multiplication and addition.

Although a characterization does not amount to a definition of what it is to be a so-called *function in intension*, there at least exists a tradition (which has now largely been taken over by constructive and intuitionistic mathematics) for regarding functions as something other than sets. And it may also be noted that there exists an independent motivation for acknowledging such a distinction between extensional and intensional notions of functions which is grounded in classical substitutivity failure arguments of the sort just considered. For note that we cannot conclude from the premises "S believes that n^2 is monotonic" and $n^2 = \sum_{i=1}^{n} 2i - 1$ that "S believes that $\sum_{i=1}^{n} 2i - 1$ is monotonic." We are forced to conclude that in intensional contexts, functional expressions like n^2 and $\sum_{i=1}^{n} 2i - 1$ must denote something other than the sets which they are now taken to denote. Following Frege, we should take these entities as the senses of n^2 and $\sum_{i=1}^{n} 2i - 1$. And although Frege mentions the necessity of extending the distinction between sense and reference to functions only in passing (cf. [40]), it has generally been thought (cf., e.g., [150]) that the appropriate candidate for the role of sense of a functional expression is the notion of function in intension as explicated above.

Since there is already a precedent for acknowledging the difference between the sense

and reference of a functional expression in this manner, it seems reasonable to consider the possibility that the denotations of procedure terms like EUCLID and NAIVEGCD be taken to be functions in intension. In certain respects this may seem like a useful proposal. For it at least seems possible that we may explicate the notion of an intensional function in a manner which allows for the well-formedness of both (1.4) and (1.5) and for the possibility of analyzing intensional function application in a manner which results in intuitively reasonable truth conditions for the latter. If all this were possible, we could then claim to have vindicated algorithmic realism by showing how procedure-denoting terms could be taken to denote items in some previously recognized class of intensional objects.

But although setting things up in this manner provides a certain degree of insight into how the general notion of algorithm may be taken to relate to traditional treatments of meaning and reference, the assimilation of algorithms to functions in intension makes little real progress towards providing a systematic basis for understanding discourse about procedures in contemporary computer science. The fundamental problem with this proposal is, needless to say, that as presented above the notion of function in intension is even more schematic than the informal notion of algorithm with which we began. For as we will see in section 3, the class of expressions which were considered acceptable means of denoting functions grew considerably over the course of the eighteenth and nineteenth centuries. And thus not only was no consensus even reached on the definition of this class of functions, but at various points there was disagreement as to whether a certain correspondence ought to be accepted as a genuine function.

While there is broad and systematic basis for thinking that the theoretical role which algorithms play in our practices can be aligned with that of traditionally recognized classes of intensional entities, I will henceforth assume that prior studies of these entities shed little light on the concerns specific to algorithmic realism. This situation arises not only because of our failure to develop a formal theory of functions in intension but also because the formal study of algorithms has been conducted as a technical subfield of computer science with its own goals and methods. As a consequence, there now exists a variety of technical classifications and results pertaining to algorithms for which there exist no analogues in regard to our reasoning about propositions. This means that the task of constructing

a foundational theory of algorithms which is consistent with our theoretical practices is almost certainly going to be harder than the task of constructing an analogous theory about propositions or Fregean senses. For note that once we have decided what theoretical role we wish entities like propositions to serve (e.g. that of serving as the correlates of sentence meaning or the objects of propositional attitudes) then the rest of the data which such a theory must satisfy is determined by intuitions about meaning and entailment. But such intuitions are both inevitably partial (in the sense of not necessarily extending to an entire language) and also potentially defeasible (in the sense there can be disagreement between theorists as to whether a given inference is an entailment or an implicature).

On the other hand, the data constraining a general theory of algorithms comes from a number of well developed subfields of computer science which are standardly regarded as issuing mathematical *theorems* concerning algorithms as typified by (1.1a-e). As I have already noted, for instance, one important dimension along which different algorithms for computing the same function are routinely compared is that of asymptotic (or "big O") running time complexity – i.e. the rate of growth of the number of computational steps they require to return an output as a function of the size of their input. This feature of individual procedures is treated as fundamental metric for distinguishing procedures which compute the same function. For instance it is because MERGESORT has asymptotic running time $O(n \log_2(n))$ and INSERTIONSORT has asymptotic running time $O(n^2)$ that the former procedure is generally considered to be a more efficient algorithm. For this reason, asymptotic running time is standardly considered to be a fundamental invariant of individual algorithms. This means that however an algorithmic realist might propose to assign a denotation to the term MERGESORT it must turn out that the object associated with the former has whatever property is taken to correspond to the predicate "has running time $O(n \log_2(n))$" to ensure that (1.1a) turns out to be true. And similarly, with respect to the term INSERTIONSORT and the predicate "has running time $O(n \log_2(n))$". And from this it follows *indefeasibly* that no adequate foundational theory of algorithms can treat the terms MERGESORT and INSERTIONSORT as denoting *the same* algorithm.

This is just one of many observations derived from the practice of computer science which places a substantial constraint on the sort of detailed formal theory of algorithms

which would be required to defend algorithmic realism. In the next chapter I will attempt to characterize, at least in broad outlines, a class of statements similar to those given in (1.1) which represent the theoretical role played by algorithms in computer science. Taken in conjunctions with statements such as (1.4) which express mathematically significant results about applying an algorithm to certain set of arguments, these sentences can be taken to comprise a logical theory which I will refer to as T_p. I will then explore the logical features of this theory in detail and propose the task of vindicating algorithmic realism can be reduced to that of arguing that 1) it successfully codifies our computational practices in a manner which assigns procedures the logical type of objects and 2) may be proven to be consistent.

Note in regard to the first desideratum that there are a number different criteria relative to which we might take an axiomatic theory as successfully codifying our discourse involving algorithm. One important dimension is whether such a theory treats the sorts of terms appearing in our informal discourse which we took to denote procedures as denoting *objects*, as opposed to say, properties or functions. For as I have already noted, part of what is involved with realism about algorithms is a willingness to treat occurrences of names like MERGESORT or EUCLID as they appear in sentences like (1.1a) or (1.5) as denoting objects. In particular, since these sentences have the appearance of being subject predicate statements, it is traditionally taken to follow that we must regard terms like MERGESORT and EUCLID as denoting an objects. For as Frege [39] is famously said to have argued with respect to sentences of the form $P(a)$ (where in the case relevant to him, a was a numerical term like 17 and $P(x)$ a numerical predicate like $Prime(x)$), we could not account for their truth or falsity if it were not possible to assign reference of a so that the item it denoted either fell under or failed to fall under the concept denoted by $P(x)$.

One problem which we will encounter in subsequent chapters, is that even if it is possible to formulate a candidate theory which codifies our practices involving algorithms in both mathematics and computer science, it is not at all clear that such a theory would ensure that procedure-denoting terms functioned in this manner. We have already encountered one prima facie problem of this sort. For note that we have already seen that although a term like EUCLID may appear as a name (as in (1.4), it may also appear as a functional

expression (as in (1.7). The apparent necessity of accommodating both forms might lead one to worry about whether it is possible to construct T_p in a manner which preserves the apparent logical forms of these statements without leading to some sort of paradox arising from the fact that EUCLID is used as a term denoting entities at two different type theoretic levels.

In Chapter 2, I will argue that this is not the case. In particular, I will suggest that there is a straightforward way of analyzing occurrences of algorithm denoting terms such as the functional expression appearing in (1.7) proper names which function as arguments to a higher order application functional. I will argue that doing so both allows for a more consistent semantic analysis of procedural discourse and is foreseen by a number of technical developments in type theory and the semantics of programming languages.

The issue which I believe represents a much more substantial hurdle for the construction of T_p – and in fact corresponds to the Achilles heel of algorithmic realism – turns on a much more mundane kind of worry about the logical consequences of taking procedure denoting expressions to refer to objects. Simply put, this worry boils down to the fact that while our informal discourse about procedures contains apparent singular terms which we wish to treat as denoting abstract objects, our use of these terms does not in itself guarantee that we use them consistently with respect to stating equalities and inequalities between algorithms and also predicating properties of algorithms.

Note in particular that as we have already observed on the basis of examples like (1.5) and (1.7), the entities assigned as the references of EUCLID and NAIVEGCD must be finer grained than the functions which we take these procedures to compute. But once we have realized that algorithms are individuated more finely than functions in extension, there immediately arises the question of how algorithms themselves are to be individuated. And failing an obvious answer, we may note that there is substantial precedent for worrying about the consequences of discourse which attempts to treats algorithms as objects prior to fixing their identity conditions.

Several substantial components of our computational practices bear directly on this issue. For note that there are in fact a wide variety of different ways by which a procedure may be specified in practice. I have thus far concentrated on the use of proper names

like EUCLID and MERGESORT. But as we will see in Chapters 2 and 3, these expressions function in much the same way as constants like π or e in arithmetic or analysis – i.e. they are introduced by explicit definitions which are taken to fix their meanings in terms of complex descriptions. In the case of procedures, such descriptions standardly take the form of two classes of formalisms which I will refer to respectively as *programs* and *machines*. Roughly speaking, a program is a complex linguistic entity build up by using individual statements (which in the general case may have either an imperative, descriptive of recursive syntax) defined over a formal grammar. As discussed above, programs are standardly said to *express* algorithms. Even more roughly speaking, a machine is an abstract combinatorial entity (akin, say to a group or graph) consisting of a class of computational states together with an abstract definition of state transition in terms of which a notion of *execution* may be defined. Machines are typically said to *implement* algorithms in something very roughly like the sense in which an electronic device might be said to implement a schematic diagram.

Unlike algorithms, we know that individual programs and machines correspond to mathematical objects from the outset. We may thus assume that they are members of well defined mathematical classes $\mathcal{P}$ and $\mathcal{M}$ which we can assume for the moment are defined in the language $\mathcal{L}_p$ of T_p. And since we also see that programs and machines are finite combinatorial entities, we may further assume that there are classes of $\mathcal{T}_P$ and $\mathcal{T}_M$ which uniquely denote each member of $\mathcal{P}$ and $\mathcal{M}$. The foregoing observations suggest that for every algorithm A, there will be non-empty classes of terms $\mathcal{T}_P^A$ and $\mathcal{T}_M^A$ such that each term $\pi \in \mathcal{T}_P^A$ corresponds to a program (or a description of a program) which we take in practice to express A and that similarly, each term $m \in \mathcal{T}_P^M$ correspond to a machine (or a description of a machine) which we take in practice to implement A. For this reason, I will argue in Chapter 3 that T_p will have to contain classes of sentences formalizing statements of the forms

(1.8) a) A is the algorithm expressed by program π.

b) A is the algorithm implemented by machine m.

for each term $\pi \in \mathcal{T}_P^A$ and $m \in \mathcal{T}_P^M$. As a consequence, this means that in any mathematical interpretation of T_p, the terms which we take to formalize the complex descriptions "the

algorithm expressed by π" and the "the algorithm implemented by m" must denote the same object (i.e. the mathematical correspondent of A in the interpretation).

On the basis of these and related observations concerning how reference to algorithms is mediated, I will argue in Chapters 2 and 3 that any attempt to defend algorithmic realism by directly identifying individual algorithms with individual mathematical objects is doomed to fail. For note that if A is any algorithm, then if either of the classes of terms $\mathcal{T}_P^A$ and $\mathcal{T}_M^A$ contains more than a single term (or even if they both contain a single term), then there will be more than one way of referring to A – say as both the "the algorithm implemented by m_1" and "the algorithm implemented by m_2" where $m_1, m_2 \in \mathcal{T}_M^A$ denote distinct machines. Thus if we proposed to identify A directly with the denotation M_1 of m_1 in some model of T_p, then the question would inevitably arise how it is that A can be identical with M_1 without being identical to M_2, the interpretation of m_2 in this model. But clearly A cannot be identical to both by virtue of the fact that they are distinct mathematical objects.

I will suggest that instead of pursuing a so-called *reductionist* strategy of this sort, an algorithmic realist will be much better off pursuing what I refer to as an *abstractionist* strategy. Such a proposal attempts to understand algorithms as having the logical type of objects by the use of so-called *abstraction principles*, similar to those which figure in Frege's well known analysis of classes and natural numbers. Skipping over many details, if we let $prog(\pi)$ be the $\mathcal{L}_p$ term formalizing "the algorithm expressed by π" and $imp(m)$ be the $\mathcal{L}_p$ term formalizing "the algorithm implemented by m", then these principles will have the form

(1.9) a) $prog(\pi_1) = prog(\pi_2) \leftrightarrow \pi_1 \approx \pi_2$

b) $imp(m_1) = imp(m_2) \leftrightarrow m_1 \underline{\leftrightarrow} m_2$

Here $\approx$ and $\underline{\leftrightarrow}$ correspond to binary predicates intended to denote equivalence relations over $\mathcal{P}$ and $\mathcal{M}$ which hold respectively just in case the programs denoted by π_1 and π_2 express the same algorithm and the machine denoted by m_1 and m_2 implement the same algorithm. In the context of T_p, the statements (1.9a,b) are intended to serve the same function Frege claimed was served the statement of Frege arithmetic now know as Hume's

principle (i.e. "the number of F equals the numbers of G if and only if the Fs and the Gs can be put into 1-1 correspondence") – i.e. it serves to i) give the identity condition of algorithms in terms of that of programs and machines and ii) to implicitly define the functions *prog* and *imp* which map these entities into the domain of algorithms.

These remarks concerning the details of how algorithmic realism works are, of course, extremely schematic. In Chapter 2 I will present a uniform development of T_p which will include axioms of the form (1.8) and (1.9) together with an argument that the structure of discourse about algorithms suggests that this is by far the most promising means by which to develop a formal theory in which they are treated as abstract objects. Readers familiar with the recent development of Frege's philosophy of mathematics will, however, realize that this proposal also raises a number of familiar conceptual and technical problems which will ultimately have to be addressed by an algorithmic realist who wished to adopt it. In particular, one may inquire i) whether (1.9a,b) can plausibly be taken as analytic of our background notion of algorithm, ii) whether the adoption of such principles gives determinate truth conditions to so-called *mixed identity statements* (e.g. ones of the form $prog(\pi) = imp(m)$) and iii) whether it can be shown that that these statements are logically consistent together with the rest of T_p.

Analogues of each of these questions are generally taken to pose serious threats to the ontology Frege proposed for natural numbers and sets. And thus if the ontological strategy which I am proposing is indeed the unique best way of developing algorithmic realism, one might think that these sorts of challenges must be met head on. But note that the ontological aims of the algorithmic realist are somewhat different from those of Fregean (or neo-Fregean) logicism. In particular, the the latter sort of theorist wishes to show that mathematics is reducible to logic, where (at least roughly speaking) the former sort merely wishes to show that computer science is reducible to mathematics. This means that the algorithmic realist has a much greater range of technical and rhetorical options available to him in replying to questions i)-iii). However a detailed discussion of what these are will also have to wait until Chapter 2.

Announcing my contention that an abstractionist theory is the most promising framework in which to attempt to seriously develop algorithmic realism does, however, allow me

to lay out in advance where I think the central difficulty for this view lies. For note that another substantial set of constraints which would be imposed on the theory T_p concern the codification of results in the analysis of algorithms and complexity theory which reports on the properties of individual algorithms. As noted above, this will include statements like "MERGESORT has running time $O(n \log_2(n))$" which the realist is inclined to analyze as subject predicate statements of the form $\varphi(a)$. As noted earlier in this section, many of these predicates are intensional in the sense $\varphi(a_1)$ may be true but $\varphi(a_2)$ may be false even though a_1 and a_2 denote algorithms which determine the same (extensional) function.

Note, however, that we have also observed that there will generally be multiple ways of denoting the same algorithm A. For instance A can be denoted as the algorithm expressed by the distinct programs π_1 and π_2 or machines m_1 and m_2. These facts will also have to codified in T_p by statements of the form $prog(\pi_1) = prog(\pi_2)$ and $imp(m_1) = imp(m_2)$ for all such pairs of terms denoting programs and implementations. But because such statements will be *theorems* of T_p, it follows in conjunction with (1.9a,b) that the predicates $\approx$ and $\underline{\leftrightarrow}$ must not only denote equivalence relations (respectively over the classes of programs and machines), but also function as a *congruence* with respect to predicates which may be proved to hold of the members of $\mathfrak{T}_P^A$ and $\mathfrak{T}_P^M$ in T_p. This is to say that whenever T_p proves a statement of the form $t_1 = t_2$ (for $t_i = prog(\pi)$ or $t_i = imp(m_i)$, $i \in \{1, 2\}$), and also $\varphi(t_1)$, it must also prove $\varphi(t_2)$ and conversely. Put into words, this means that T_p must not only have the resources to track our intuitions about the circumstances under which different programs or machines may be used to denote the same algorithm, but it must also ensure that exactly the same computational properties are provable of the algorithms specified in these different manners.

But merely pointing out that the theory T_p must have these properties in order to fulfill the algorithmic realist's purposes in no way entails that such a theory actually exists. In particular, I have thus far said nothing about how the relations $\approx$ and $\underline{\leftrightarrow}$ are to be defined or even what sort of relationship among programs or machines them are intended to formalize. And as such, it seems that the realist can offer no *a priori* guarantee that relations which simultaneously satisfy the constraints described in the previous paragraph, let alone also have appropriate conceptual properties which allow to view (1.9a,b) as implicitly giving

identity conditions for algorithms.

In Chapters 3, 4 and 5, I will offer a concerted argument that no such relations can exist. But many of the considerations which enter here are of a technical nature which pertains to the way in which the notions of program and machine have been developed in various subfields of computer science. In order to motivate the general style of argument I will employ, I will devote Chapter 3 to a detailed case study. In particular, I will introduce a pair of intuitively distinct algorithms PAL1 and PAL2 for deciding whether a finite string is a palindrome. I will then introduce classes of Turing mahcines $\boldsymbol{S}$ and $\boldsymbol{U}$ such that all the machines in $\boldsymbol{S}$ may plausibly be claimed to implement PAL1 and all those in $\boldsymbol{U}$ PAL2. On the basis of the abstraction principles (1.9a,b), it follows that for all $S, S' \in \boldsymbol{S}$, we must have $S \underline{\leftrightarrow} S'$ and for all $U, U' \in \boldsymbol{U}$ that $U \underline{\leftrightarrow} U'$. But since PAL1 and PAL2 are distinct, it should also follow that for no pair of machines $S \in \boldsymbol{S}$ and $U \in \boldsymbol{U}$ do we have $S \underline{\leftrightarrow} U$. I will argue that it is impossible to define $\underline{\leftrightarrow}$ so that it not only satisfies these constraints but also serves as a congruence with respect to complexity theoretic properties as discussed above.

Chapters 4 and 5 will be devoted to an argument that the problem encountered in Chapter 3 is not an isolated artifact of the choice of algorithms or machine model considered therein. I will set out in Chapter 5 to present a systematic analysis of the concept of *model of computation*. In the course of this study, I will identify three general classes of models which I will refer to respectively as *transition based*, *register based* and *recursion based* models. I will argue along the way that it is possible to assimilate all of these models to a more general definition of *transition system* in a sense which generalizes on the definition which will be given in section 4 of this chapter.

Chapter 5 will contain my direct argument against algorithmic realism. It will be in two parts. In the first, I will summarize and compare the two positive programmes mentioned above which have been put forth as systematic defenses of algorithmic realism – the first owing to Yainnis Moschovakis (cf. [92], [93],[95], [96]) and the second to Yuri Gurevich (cf.c [51], [9], [10]). I will first argue that these programmes correspond respectively to the views that algorithms may be identified with individual instances of a certain class of recursion based model $\mathcal{R}$ and with individual instances of a certain class $\mathcal{ASM}$ of register

based model. As such, I will argue that both views suffer from essentially the same defect. In particular, they turn out to be variants of what I refer to above as *reductionism* – i.e. the view that algorithms may be identified with particular programs or machines. Such views may be compared to forms of logicism about natural numbers which hold that particular natural numbers are to be identified with particular sets. In particular, I will argue that this view falls victim to a version Benacerraf's [7] well known critique according to which a natural number (e.g. 2) cannot be held to be identical a set s_1 (e.g. $\{\emptyset, \{\emptyset\}\}$) if there is another set s_2 (e.g. $\{\{\emptyset\}\}$) which does an "equally good" job at reflecting its arithmetic properties.

The other half of Chapter 5 will be aimed at a hypothetical (but to my mind better positioned) theorist who proposes to adopt an abstactionist strategy with respect to one of the sophisticated models of computational models – call it $\mathcal{M}$ – considered in Chapter 5. For the reasons we just discussed, the onus will be on such a theorist to propose a definition of $\underline{\leftrightarrow}$ over $\mathcal{M}$ which meets the same technical and conceptual conditions which will be in force in Chapter 3. In particular, he must define $\underline{\leftrightarrow}$ so that whenever $M, M' \in \mathcal{M}$ are machines which we would take to implement the same algorithm $M \underline{\leftrightarrow} M'$ holds and whenever they are machines which we would take to implement different algorithms $M \underline{\nleftrightarrow} M'$ holds. Based on considerations pertaining to the relationship between recursion based models and register based ones, as discussed in Chapter 4, I will argue that this can never be achieved. The detail of this will be grounded both in my prior argument that recursion based models should be assimilated to transition systems in order to be counted as legitimate machine models and also on the precise manner in which this must be carried out. And since these are matters of substantial technical detail, further description of the argument must await Chapter 5 itself.

1.3 Mathematical procedures and mathematical practice

The role of algorithms in mathematics is complex. It appears uncontroversial that a general notion of a procedure as an abstract, repeatable method for constructing a mathematical object or structure is among the set of informal background notions in terms of which we may explain or motivate various mathematical ideas. One way in which this is

evident is that there have been periods in the history of mathematics during which certain mathematical concepts have been *defined* as corresponding to forms of procedures. And it is also evident from the fact that a large of number of computational methods which were originally discovered in classical mathematics have subsequently been explicitly recognized as algorithms worthy of study.

It is thus common in the informal discourse of both mathematics and computer science to speak of algorithms *as if* they were abstract objects in much the same sense as natural numbers, groups or graphs. But some care must be exercised before we can reach the conclusion that algorithms constitute part of the *subject matter* of contemporary mathematics. For on the one hand there appears to be no prohibition against *using* procedures to derive certain forms of mathematical results and more generally in heuristic description of mathematical proofs. Many popular presentations of analysis and algebra will thus not hesitate to refer explicitly not only to mathematical constructions, methods, procedures, etc., but give examples which involve *executing* or *carrying out* such techniques. But it is also notable that as common as such language may be in the informal exposition of mathematical proofs and results, it is much more difficult to find instances in which explicit reference to procedures or their application is made in the statement of mathematical results. This is to say that there are few (if any) *theorems* of classical mathematics which are standardly expressed in a manner which contains either a singular term or quantifier which appears to denote an algorithm.

This observation may lead one to suspect that while mathematicians may feel perfectly at home using procedural language in giving informal demonstrations, the role assigned to such entities is purely heuristic or instrumental. However, as I mentioned in Section 1, the view for which I ultimately want to argue is that classical mathematics *is* genuinely committed not only to regarding procedures as abstract objects about which we can reason formally, but also to the view that they are genuine mathematical objects. However I believe this conclusion must be based on a examination of a special class of cases where procedures are not simply mentioned in the course of a demonstrating general propositions, but are rather used to derive singular (or "concrete") propositions concerning explicitly presented objects like natural numbers.

My plan of attack for elucidating the way in which classical mathematics is committed to an ontology of procedures will thus be indirect. In this section I will outline a number of ways in which procedures appear to play a role in mathematical discourse which I believe are *not* indicative of a genuine ontological commitment. This will help to single out the narrower classes of cases of the sort just alluded to whose significance I will then examine in detail in Section 4.

A reasonable place to begin our investigation of the role of procedures in classical mathematics is with an instance in which a commitment to procedural entities appears to accompany the manner in which a particular mathematical concept is defined. Perhaps the best known instance of this sort concerns the definition of the concept *function*. According to the accepted history, an abstract notion of a function as a correspondence between quantities was not employed in mathematics until the late 17th century when it was introduced by Leibniz to describe a relationship between purely geometric quantities.[7] In its original setting, the notion of function was closely tied to geometric properties of curves which were given either physically, by tables, or by polynomial equations. Euler is generally credited with having been the first to provide a formal definition of this notion relative to which functions were identified with so-called *analytic expressions*.[8] The paradigmatic example of such an expression at this time would have been a polynomial, although Euler explicitly allowed that functions could also correspond to expressions formed from (terms denoting) transcendental functions and power series.

Most scholars agree that Euler's definition initiated a period during which the notion of function was understood as being connected to certain classes of linguistic expressions. The development of the calculus and discoveries in mathematical physics led to gradual expansions in this class to include, e.g., expressions containing differential and integral terms, case definitions which (as we would now put it) describe discontinuous functions, etc.

---

[7] There are, of course, many important precursors to the introduction of the function concept. For instance, numerous examples of verbal or tabular descriptions of functions can be found in classical and Arabic mathematics. However, no general notion appears to have emerged from examples of this sort until the first applications of algebra to geometry by Descartes in the mid-17th century. For more on the history of the notion of function, cf. [159] on which the following discussion is based.

[8] "A function of a variable quantity is an analytic expression composed in any way from this variable quantity and numbers or constant expression." [159], p. 61

However, most of these expansions in usage were accompanied by a corresponding limitative doctrine according to which the only functions which existed were those corresponding to expressions in the sanctioned class. Often such views were based on the supposition that functions corresponded to dependencies between physical or geometric quantities. And since it was also held that only continuous dependencies of this sort existed in nature, this led to a limitative conception of function according to which only those functions which are expressible by analytic terms were held to exist. For instance, a view along these lines led Fourier [15] to claim, as late as 1822, that all functions must be given by (what we would now call) Fourier series.

This view about the nature of functions stands in sharp contrast to our modern understanding in two significant respects. First, we now typically distinguish between what I refer to as *functional expression* and the functions which they are used to denote. The former are linguistics items such as x^2, $x^17 + 3x^2 + 2$, $\sin(x)$, $\sum_{i=1}^{n} i$, etc., which may or may not be drawn from some class of expressions which we have chosen to label as "analytic." We would now say that such expressions refer to functions in much the same way that singular terms refer to objects, and predicates refer to properties or sets. From this it follows that whatever functions are taken to be, the relationship which functional terms bear to functions is not that of identity but rather analogous to that borne by numerals to natural numbers.

The first contrast between the contemporary view of functions and that which prevailed in the 18th and 19th centuries thus turns on how we are to distinguish between the mathematical expression and what this expression denotes. Of course, in order to clearly frame the matter in this way, we must have some candidate in mind for what the denotation of a functional term should be. The modern view is, of course, that a function f with domain X and range Y is a single-valued relation $R_f \subseteq X \times Y$, i.e., a relation such that if $R_f(x, y_1)$ and $R_f(x, y_2)$, then $y_1 = y_2$. But relations are sets and set theory wasn't firmly established as a universal foundation for mathematics until at least the 1920s. Thus, throughout the period in question, there would not have been a precise means of distinguishing between functional expressions and what they denote.

One consequence of this is that although authors like Leibniz, Euler, Dirichlet, and

Fourier occasionally wrote as though they thought of functions as linguistic items, there are other instances in which they appear to acknowledge a distinction between functional expressions and (as it was often described) the functions which these expressions *represent* or *determine*.[9] But of course, once such a distinction is recognized, it also becomes legitimate to ask what sort of entities these authors would have taken functions to be if (contrary to their own pronouncements) they may not be assimilated to linguistic terms.

In this regard, there are at least two classes of historically plausible alternatives: 1) functions could have been regarded (as we now put it) *extensionally* – i.e., as collections of ordered pairs corresponding to a particular correspondence between domain and range; 2) they could have been regarded *intensionally* – i.e., as abstract rules or procedures which give rise to such correspondences. It may at first appear that the former alternative represents the anachronistic proposal that functions were understood set theoretically before the advent of axiomatic theory. But this would only be so if we were unable to distinguish between the class of *all* (as we now describe it) combinatorially possible functions between two sets and various subclasses of this class consisting of those correspondences which met some further conditions. Of course, some degree of set theoretic understanding is required to conceive of a particular function $f : X \to Y$ as a collection of ordered pairs $\langle x, y\rangle \in X \times Y$. However, equating an individual function with such a collection does not require that we recognize that the class Y^X of *all* functions from X to Y is a set or even that even conceive of it as a well-defined totality. Once the former step has been taken, it becomes possible both to systematically distinguish between a functional expression $f(x)$ and the function $R_f \subseteq X \times Y$ which it denotes, and also to make sense of various limitative doctrines about function existence. For if it were maintained that all real valued functions had to be representable by a particular class of functional expressions – say polynomials or power series – such a limitation would correspond to a proper subclass of $\subseteq \mathbb{R}^{\mathbb{R}}$.

Following Maddy [81], we might call this view of function existence *definablism*. According to this view, functions are identified with correspondences between sets and the

[9]For instance, in stating the result which we now phrase as "Every periodic function of one real variable can be written as a Fourier series," Fourier writes "[E]ven discontinuous arbitrary functions can always be *represented* by expansions into sines or cosines of multiple arcs ... A conclusion that the celebrated Euler always rejected" [15], italics mine.

notion of correspondence can be understood as essentially corresponding to the modern notion of a set of ordered pairs. However, the central tenet of this view is that correspondence existence is governed by a linguistic principle akin to a modern comprehension axiom – i.e., that a correspondence f between sets X and Y exists only if there is a term $f(x)$ in some pre-identified class $\mathcal{F}$ of functional expression which "represents" or "determines" f. On this way of looking at things, the general definition of function retains a linguistic component, even though individual functions are taken to be items of essentially the same sort as they are on the modern conception. And from this latter fact it follows that even though the definablist holds that for every function, there exists a functional expression of the appropriate sort which represents it, identity between functions is determined extensionally in the sense that functional expressions which determine the same correspondence (e.g. $\sin^2(x)$ and $1 - \cos^2(x)$) must be taken to denote the same function.

The other option by which we might attempt to understand the pre-twentieth century notion of function is one on which functional expressions are not taken to denote directly any sort of set-like correspondence, but rather are taken to denote some other sort of non-linguistic entity which determines such a correspondence. Given a functional expression $f(x)$, one possibility for what such an entity might be is a procedure A_f which, given any value $a \in X$, returns the value $b \in Y$ such that $f(a) = b$. This view provides another means of explaining how someone could simultaneously maintain that functions are distinct from functional expressions qua mathematical objects and also that all functions are determined by functional expressions drawn from a certain class $\mathcal{F}$. For if the procedures determining correspondences between X and Y in the sense just described were treated as abstract mathematical entities in their own right, then functions could be identified with these objects which in turn could be held to be determined by expression contained in $\mathcal{F}$. One consequence of this view which distinguishes it from the former alternative is that it allows for functional identity to remain intensional since it is possible that there are pairs of distinct procedures which determined the extensional correspondence.

Making sense of this second view as a genuine alternative to the modern extensional definition of function would require two things. First, it would require that it be possible to give a general definition of a mathematical procedure which rivaled in clarity and precision

the notion of set on which the extensional definition is based. And second, in order to understand this proposal in its historical context, we would also have to develop an account of what it means for a functional expression to denote or otherwise determine a procedure which could replace the way in which we now speak of functional expressions denoting sets. But there appears to have been no systematic attempt to work out this possibility prior to the the development of the lambda calculus in the 1930s. And there does not appear to have been discussion of the distinction between intensional and extensional identity conditions for functions which would have decided between the two views prior to Frege (e.g. [36]), and Russell and Whitehead (e.g [152]) both of whom favored variants of the modern set theoretic treatment. It would thus be at best speculative to conclude that there was any point in the history of pre-twenthieth century mathematics at which functions were explicitly held to be procedures.

But at the same time, the developments which led to the final demise of the view that functions can be equated with any sort of linguistic entity may be naturally understood relative to the view that functional expressions determine procedures rather than definable correspondences. For the developments which ultimately led to the acceptance of the modern set theoretic definition were initiated by definitions of correspondences by both Baire and Lesbegue which could only be shown to exist by appealing to the Axiom of Choice. The claim that such correspondences were genuine functions was originally regarded skeptically, not merely on the basis that there was no linguistic means of describing them, but rather because there was no evident way of explicitly computing their values.[10]

---

[10]The gravity with which this concern was treated is evident from the manner in which Baire and Lesbegue described the competing view in their defenses of the set theoretic conception. For instance Baire distinguishes his view from what he took to be the prevailing one as follows:

> There is a *function* when there is a correspondence between some numbers .... On this definition, one doesn't ask after the means by which the correspondence might be effectively established; one doesn't even ask if it is possible to establish it.

And similarly,Lesbegue writes as follows:

> Although ... it is generally agreed that there is a function when there is a correspondence ... without concern for the procedure that serve to establish this correspondence, many mathematicians seems to consider only those established by analytic correspondence as true functions.

These passages provide some indication that while there may never have been a completely unitary pre-set theoretic notion of function, one set of intuitions bearing on how this notion was understood in practice of

But as we now know, the computability of a function's values certainly does *not* follow immediately from its representablity as a certain sort of functional expression. There are, for instance, functions definable over the natural numbers using a single universal quantifier which (as we would now put it) are not effectively computable. However it also bears noting that virtually all of the forms of expression which were recognized as analytic in the 19th century had associated with them various procedural means of computing their values. For instance, the representablity of f as a polynomial $p(x)$ provides an immediate means of computing the values of $f(a)$ in terms of the structure of $p(x)$. The same observation obviously applies to cases in which f is represented as a power series or other form of summation. And when we take into account the existence of numerical methods for computing roots, transcendental functions, and differentials and integrals – all of which were well developed by the end of the 19th century – it is clear that there would have existed procedures for explicitly computing the values of most functions denoted by most expressions which were recognized as analytic.

It should be borne in mind, however, that the foregoing observations belong exclusively to the history of mathematics and not to its subject matter as currently conceived. Thus, even if an historical case can be made that functions were, at least in some inchoate sense, understood as procedures in pre-20th century mathematics, this would not in itself represent something an algorithmic realist could legitimately cite in favor of the conclusion that our present mathematical practices are committed to an ontology of procedures. Nonetheless, the historical development of the understanding of functions does attest to the fact that some notion of procedure has traditionally been part of the conceptual background in terms of which it has been thought possible to explicate mathematical notions. The question thus arises whether there remain any instances in contemporary mathematics in which a concept is explicitly defined relative to a prior understanding of mathematical procedure.

I believe that this question ultimately ought to be answered in the negative. But I have already discussed one apparent exception to this claim in the form of the modern theory of computability which grew out of the foundational analyses of effectivity mentioned in the

---

nineteenth century mathematics drew upon the view that for a function f to exist, it had to be determined in some that manner would allow its values to computed.

previous section. As discussed there, the goal of the original research in this field was to provide a mathematical definition of what it means for a given function from $f : \mathbb{N}^m \to \mathbb{N}$ to be effectively computable. The informal notion of effectivity which was employed at this time was partially grounded in the understanding that simple arithmetic functions like addition, subtraction, multiplication, and exponentiation for which there were known procedures allowing their values to be explicitly computed ought to be regarded as effective. Although the term "algorithm" was not as widely used at this time, we may anachronistically characterize the class of functions of which a mathematical description was wanted as those *computable by an algorithm*.

One might reasonably think that the most straightforward way to analyze this latter notion is first to provide a formalization of the notion of *algorithm* itself. On this basis, one could then provide a further analysis of what it meant to carry out or execute an algorithm which would in turn provide an analysis of the notion *function computable by algorithm*. It is thus also reasonable to presume because the analysis of effective computability, equivalents of which were given by Church, Turing, Gödel, Kleene, and Post, is now widely considered to have been *successful*, this must be because one or more of these theorists also succeeded in giving a mathematical analysis of the concept of *algorithm*. And for this reason, it is also commonly assumed that not only was one or more successful analysis of this given during the 1930s, but also that the subsequent development of computability theory has been concerned, at least in part, with studying the properties of individual algorithms and how they may be mathematically represented.

I will argue in Chapter 2 that all of these suppositions turn out to be false. In particular, the classical analysis of effective computability did not proceed by first attempting to formalize individual effective procedures and then generalizing with the hope of obtaining a universal analysis. As a matter of historical fact, it was rather accomplished by presenting several general models of finitary computation which were based on an analysis of locally effective steps as typified by the basic read/write/move operations of a Turing machine or the rewriting of a lambda term under β-conversion. But as it turns out, these analyses give rise to very general classes of formal models which only in retrospect may be argued to contain members which induce intuitively effective functions. And as I will argue, this

means that not only were these models not intended by their expositors to formalize the notion of an *individual* computation procedure, but also that we may see in hindsight that they do a relatively poor job of doing so.

One upshot of this is that even though the modern theory of effective computability which grew out of the foundational analyses of the 1930s seems like a natural setting in which to find mathematical discourse whose interpretation relies on a realistic understand of algorithm, this turns out not to be the case. This subject, which until recently has been referred to as *recursive function theory* (as developed in textbooks such as those of Rogers [113], Odifreddi [103], and Soare [135]), is most aptly described as being concerned with the relative definablity of subsets of the natural numbers. And although such sets which are studied must be specifiable intensionally (i.e., as the sets of natural numbers which may be enumerated by an effective procedure, possibly with the addition of a non-effective "oracle"), the vast majority of its results are extensional in the sense that they are invariant with respect to *how* these set are presented. There is hence a straightforward and generally acknowledged sense in which computability theory is not concerned with the properties of individual algorithms but whether with what can be computed by *some* effective algorithm.

The foregoing considerations suggest that if we wish to identify some way in which mathematical practice is committed to the existence of algorithms, such a commitment cannot arise because any of its modern subfields takes mathematical procedures themselves to be its official subject matter.[11] But as I noted initially, it is also clear that reasoning which involves procedures has traditionally played an important part in the practice of mathematics. In this regard, we may identify two substantial roles played by reference to procedures in the context of mathematical proof: 1) a procedure may serve as a component

[11] One obvious exception to this generalization is intuitionistic mathematics. In this setting a commitment to an abstract mathematical procedure occurs already at the level of propositional logic. For note that on the standard Brouwer-Heyting-Kolmogorov interpretation of the logical connectives, a proof of an implication $\varphi \to \psi$ is defined to be a procedure for transforming a proof of φ into a proof of ψ. This idea is carried over into intuitionistic number theory and analysis wherein effectiveness is not only standardly imposed as a requirement on function existence, but functions themselves are identified with effective procedures. But as I noted above, however, not only does it remain controversial how these idea should be formalized when providing a semantics for such theories, most of the leading proposals (like realizability semantics) simply take over formalisms which were developed independently in proof theory or theoretical computer science. Thus while intuitionism does serve as a clear example of a theoretical development in which procedures are treated as mathematical objects, I will argue that it is not a source of constraints on the notion of algorithm which cannot be derived from other sources.

of a proof in the sense of either being specified as one of its components or being referenced so as to ensure the existence of a particular object or class of objects; 2) a procedure may be applied within a proof so as to derive an explicit solution to an instance of a numerical, algrebraic, or combinatorial problem.

These two roles are quite distinct operationally. But taken together, they make a significant contribution to fixing the class of mathematical statements which mathematicians would conventionally describe as having been acceptably demonstrated at any given time. And for this reason, it will be difficult to deny that reasoning involving procedures plays a central role in the acquisition of mathematical knowledge through proof. The question I will attempt to address over the course of the rest of this Section and the next is whether acknowledging the epistemic significance of procedures in mathematical practice in any way commits us to regarding them as mathematical objects in their own right.

Speaking first of the role of procedures in mathematical proofs, classical theorems are often demonstrated in a manner which specifies either implicitly or explicitly, a method for constructing a mathematical object or structure given other objects or structures. A simple example of this kind occurs in the elementary proof of König's Lemma – i.e., every infinite, finitely branching tree has an infinite branch – found in most set theory textbooks (cf., e.g, [71]).[12] This proof proceeds by showing that for every infinite, finitely branching tree $\mathcal{T}$ there exists a branch b such that $b(0)$ corresponds to the root v of $\mathcal{T}$. This existence claim is demonstrated by specifying a so-called *construction* by which b is "built up" by showing how the values $b(i)$ may be chosen in stages parameterized in $i = 0, 1, \ldots$.

This construction would typically be specified in the following way:

(1.10) KONIG(T)

- At stage 0, let $b(0) = v$.
- At stage i, choose $b(i+1) = u$ such that $u < b(i)$ and $\{v \in T \ : \ v < u\}$ is infinite.

Note that since $v \in T$, the set of nodes $\{u \in T \ : \ u < v\}$ below $b(0)$ must be infinite. Also

[12]Recall that a tree $\mathcal{T}$ is a structure $\langle T, \leq \rangle$ such that $\leq$ is an anti-symmetric partial order and the sets $\{u \in T\} u < v$ are well ordered by $<$. We may additionally define a branch through $\mathcal{T}$ to be a function $b : \mathbb{N} \to T$ such that for each $i \in \mathbb{N}$, $b(0) < b(1) < \ldots < b(i-1) < b(i)$.

note that at stage i there must always exist a u satisfying the condition in the second part of (1.10), as otherwise the set of nodes accessible from $b(i)$ would finite, and thus the set $\{v \in T \ : \ b(i) < v\}$ would be a finite union of finite sets, contra to our choice of $b(i)$. It thus follows that since stage i can always be carried out for all $i \in \mathbb{N}$, we may therefore look upon (1.10) as specifying a means by which b may be built up in an inductive or step-by-step manner.

We would typically call such a specification a means of *building* or *constructing* the branch b given $\mathcal{T}$. It is common in such contexts for verbs like "choose," "pick," and "assign" to be used in the active voice. This suggests that such a specification is most naturally interpreted as describing a process by which a mathematical object is actively being constructed in time. However, it is already unclear in the case under consideration how an object like b may be literally formed during or as a result of the exposition of the proof in question. For note that relative to the standard realistic understanding of mathematical objects, it makes no sense to describe the act of carrying out a procedure as bringing a mathematical object satisfying a condition Φ into existence. For on this understanding, such an object either exists or fails to exist independently of our mathematical activity. On this interpretation, for instance, a given tree $\mathcal{T}$ is a static abstract structure and as such either contains or fails to contain an infinite branch. And thus, the procedure described in (1.10) cannot be taken as describing a process whereby a branch b is literally brought into being in anything like the sense in which a complex physical object is created by a process of assembling its phyiscal constituents.

One way in which to reconcile the use of operative language (as exemplified by the use of "choose" in (1.10) with a realistic understanding of mathematical existence is to interpret talk of constructing mathematical objects not as describing a process by which they are literally constructed, but rather as specifying procedures which, were they to be carried out, would result in certain kinds of specifications of these objects. For instance, in the case under consideration, we can view (1.10) as specifying a method of constructing a branch b through $\mathcal{T}$ not in the sense of literally generating b, but rather in the sense of describing a sequence which is assumed to already exist. Suppose we agree to call this procedure "König's algorithm" or A_k for short. We could now speak of the proof as presenting a

method which, were it to carried for a particular tree $\mathcal{T}$, would result in assignment of nodes to nodes as the value arguments of $b(i)$ corresponding to an infinite branch. The argument sketched above proves that A_k is *correct* in the sense that as long as $\mathcal{T}$ satisfies the hypotheses of König's Lemma, then it may always be carried out to fruition, i.e., that $b(i)$ may be assigned a value for each $i \in \mathbb{N}$.

Treating (1.10) as specifying an abstract mathematical procedure has several interpretative advantages with respect to explaining its role in the proof of König's Lemma. For note that part of what is involved with treating A_k as a procedure is the ability to treat it as a method which can be uniformly applied to different trees on different occasions as opposed to treating it as a description of a single construction by which a single branch is "built up" from scratch. Once we have taken this step, it is then possible to reason about this procedure abstractly that does not entail that it has been, or even could be, carried out in practice. In particular, statements like " choose $b(i+1) = u$ such that $u < b(i)$ and $\{v \in T \ : \ v < u\}$ is infinite" may now be interpreted as instructions stating what a hypothetical agent *would* do were he to carry out the specified procedure as opposed to commands to an actual agent (e.g. the reader of the proof) to perform various "mathematical actions." Thus the problem of interpreting the language used in proofs which contain so-called constructions is reduced to that of providing a general account of procedural execution as opposed to that of directly assigning meaning to statements appearing in such proofs (such as those in (1.10)) which appear to be commands to carry out mathematical operations.

In Chapter 2, I will present a general theory of exactly this sort. For the time being, however, we may understand a procedure as corresponding to a set of instructions which can be carried out by an appropriately idealized mathematical agent by following the instructions corresponding to its constituent steps. For instance, in the case of A_k, the procedure specifies that the agent must choose v as the value of $b(0)$, and given that the value of $b(i)$ is u, it specifies a method by which the value of $b(i+1)$ may be selected. Since the proof described above guarantees that this method may be carried out for all i, the existence of A_k turns out to be sufficient to demonstrate the existence of the object b called for by König's Lemma. However, on the realistic understanding of mathematical

existence which we are presuming, the reverse implication is not taken to be true. In other words, on the assumption that quantification over procedures makes sense, the existence of a procedure like A_k for constructing b is in no way asserted to be equivalent in meaning to the existence of such a sequence.

Returning to the question of accounting for the significance of procedures like A_k to the practice of classical mathematics, it seems reasonable to draw the following conclusions. First, the foregoing considerations suggest that if we wish to think of mathematical objects realistically, then the existence or non-existence of a procedure for constructing a certain object cannot be taken to have any bearing on mathematical existence claims. From this it follows that although it is often useful to interpret talk of mathematical constructions in terms of the existence of mathematical procedures, the role of these procedures must be explained solely in terms of the role which they play in the specification of proofs. But since the study of mathematical proofs and their properties is generally taken to be a topic in the analysis of mathematical *knowledge* as opposed to mathematical ontology, this suggests that the significance of mathematical procedures to mathematical practice must be understood in terms of their epistemic contribution to the proofs in which they are employed.

This is, of course, a broad and somewhat schematic conclusion. Nonetheless, I will suggest below that not only are there well-known examples of mathematical procedures whose epistemic contribution to mathematical proof can be analyzed quite precisely, but also that accounting for this role indirectly raises questions about mathematical ontology which bear on the status of algorithmic realism. But before discussing these matters, I also want to briefly describe a general problem which arises, when we attempt to account in more precise terms for the contribution of a procedure like A_k, to a result like König's Lemma.

I have thus far described A_k as a procedure which can, at least in principle, be applied by a mathematical agent so as to produce a branch through any infinite, finitely branching tree. It is important to realize, however, that A_k is traditionally taken to have this significance despite the fact that it is not *effective* in either a formal or informal use of this term. For note that it is not even clear what it means to say that a mathematical agent could carry

out A_k for an arbitrary infinite tree $\mathcal{T}$. One simple reason for this is that even though this procedure specifies how to construct b by determining the value of $b(i)$ for each $i \in \mathbb{N}$, there will be no finite stage at which this process is complete. But note that it unclear whether the specification of how the value $b(i+1)$ is to be chosen on the basis of the value of $b(i)$ corresponds to a method which can actually be carried out in any reasonable sense that we might assign to the act of "choosing" a value.

One way of precisely characterizing these problems is to specify the manner in which $\mathcal{T}$ may be given as an input to A_k and in which b must be specified as its output. For instance, we might require that $\mathcal{T}$ be presented in a manner such that for all $u, v \in T$, the relation $u < v$ may be effectively determined. And we might allow that it is sufficient to specify b by providing a finite description which allow us to construct the initial segment $\langle b(0), \ldots, b(1) \rangle$ for each $i \in \mathbb{N}$. These requirements can be formalized even further by requiring that a) $T = \mathbb{N}$, that b) $\leq_{\subset} N \times \mathbb{N}$ be a recursive relation, and finally that c) the branch b be specified as an index to a recursive function which on input i returns a code for a finite sequence corresponding to a path of length i through $\mathcal{T}$ starting at v. Even relative to these liberal assumptions for making precise the sense in which A_k might be taken to describe a procedure for constructing b, it still turns out that this procedure cannot be accepted as effective. For it is a theorem of elementary recursion theory that there are recursive trees in the sense of b) through which there exist no recursive branches in the sense of c). It thus follows that not only cannot A_k be taken to be an effective procedure for finding an infinite branch through an arbitrary infinite tree, but that this remains true when we consider only trees which themselves are effectively presented.[13]

In this respect A_k is similar to many other components of mathematical proofs which are typically referred to as constructions. In fact, constructions very similar to (1.10) occur in the standard proofs of a number of several well known results including the Bolzano-Weiertrauss Theorem in analysis ("Every bounded subset of $\mathbb{R}$ has an accumulation point"), Lindenbaum's Lemma in logic ("Every consistent set of sentences can be extended to a

[13]The above cited result is usually summarized by the statement that recursive König's Lemma is false for recursive trees – cf., e.g., [103]. This is related to the fact that when presented in full detail, the proof of König's Lemma not only requires a weak form of the Axiom of Choice (know as the Axiom of Dependent Choice), but also that it may be proven to be equivalent to this statement over a weak base theory.

maximally consistent set of sentences") and the existence of prime ideals in commutative rings. In each case, the central component of the proof involves the specification of a construction which can be interpreted as an abstract procedure in almost exactly the same sense as (1.10). And in each case, the method so described can be shown to be non-effective in essentially the same manner as A_k.[14]

Given the non-constructive nature of the procedures which often occur in standard mathematical proofs, it is difficult to provide a uniform account of their epistemic significance. I have already noted that recasting so-called mathematical constructions occurring in informal proofs as specifications of mathematical procedures allows for a somewhat smoother interpretation of the operational language which often occurs in such proofs. But it is also notable that statements like König's Lemma may easily be formulated in the language of set theory and then supplied with formal proofs in an axiom system like ZFC (or ZF plus Axiom of Dependent Choice). Formal proofs obviously contain no mention of mathematical constructions and no operational language, and yet such proofs are also standardly assumed to provide the same sort of epistemic justification of their conclusions as do informal ones.

On this basis, it is tempting to conclude that even the epistemic role of mathematical procedures is exhausted by their heuristic significance in allowing us to understand what would otherwise be bare existence claims (such as the existence of an infinite branch through an infinite tree) in terms of the existence of a general sort of non-effective procedure. There are, however, a number of other well-known existence theorems whose proofs describe constructions which are effective in both in formal and informal sense. A well-known example of such a result is Gentzen's *Hauptsatz* or Cut Elimination Theorem for first-order logic which states that any formula provable in the classical or intuitionistic sequent calculus using the so-called Cut Rule can also be derived without using this rule. Gentzen's proof contains an explicit specification of a method for transforming proofs containing the cut rule into proofs of the same formula without the cut rule. But in this case, the procedure

[14]In fact, when these results are translated in the language of second-order arithmetic, all three of the results may be shown be equivalent to be König's Lemma restricted to binary trees – cf. [131]. This allows us to see that, in this setting, all four statements are equivalent to the same set existence principle.

itself turns out not only to be effective, but also to be of substantial practical utility in various branches of formal logic. For instance the cut elimination procedure may be used to construct proofs with desirable structure features (most notably the subformula property) and to aid in the automation of proof search for a variety of different logical calculi.

A number of other classical theorems have the same property of possessing proofs which specify, either implicitly or explicitly, effective procedures which are sufficiently useful in their own right to have earned an independent recognition within mathematics. Such theorems lead a sort of double life in mathematical practice. On the one hand they announce a general fact – e.g. that a certain object or class of objects exists or that a given function takes on a certain value or range of values. And on the other, they indicate how these objects or values may effectively be determined by a method which is contained in their proofs. Since the interplay between these dual applications can be complex, it will be useful to consider an example in detail. To this end, consider the statement of Sturm's theorem from abstract algebra:

Theorem 1. Let $f(x)$ be a polynomial with real valued coefficients. Then the number of distinct real roots of $f(x)$ over the interval (a, b) (where $f(a) \neq 0$ and $f(b) \neq 0$) is equal to $\sigma(a) - \sigma(b)$ where $\sigma(x)$ is equal to the number of changes of the members of the elements of the Sturm function chain for $f(x)$ evaluated at x.

The statement of this theorem relies on the definition of the Sturm function chain for $f(x)$ which is defined to be a sequence of polynomials $r_1(x), \ldots, r_n(x)$ defined as follows: $r_0(x) = f(x)$, $r_1(x) = f'(x)$ (the derivative of $f(x)$) and $r_i(x) = -(r_{i-2}(x) \bmod r_{i-1}(x))$.

It will be noted that the Sturm function sequence is composed of the polynomials which serve as the intermediate steps in the application of Euclid's algorithm to $f(x)$ and $f'(x)$ over the field $\mathbb{R}[x]$.[15] This is by no means a coincidence. In fact general reasoning involving this algorithm serves as a central part of the standard proof of Sturm's theorem.

[15] As we have seen, in its most familiar form, Euclid's algorithm is a method for computing the greatest common divisor of two natural numbers n and m. However it was observed in the nineteenth century that essentially the same procedure could be applied over any so-called *Euclidean domain* – i.e., a ring R for which there exists a map $N : R \to \mathbb{N}$ (known as a *norm*) such that for any $a, b \in R$ such that $b \neq 0$, there exists unique q and r such that $a = qb + r$ and $N(r) < N(b)$. $\mathbb{N}$ is clearly such a structure where $N(n) = n$ as is $\mathbb{R}[x]$ where $N(f(x))$ is taken to be the degree of $f(x)$. It is, of course, a matter of some delicacy whether Euclid's algorithm applied over these two structures is really *the same* procedure. A detailed treatment of this question will have to await the development of a general approach to algorithmic identity in Chapter

In particular, this proof relies on the fact that Euclid's algorithm is both *total* (i.e., that it halts after a finite number of steps, meaning that the Sturm function sequence for $f(x)$ will always be finite) and *correct* (i.e., that if the result of applying Euclid's algorithm to $g(x)$ and $h(x)$ is $d(x)$, then $d(x)$ is in fact the greatest common divisor of $g(x)$ and $h(x)$).

And as should be evident from the statement of Sturm's theorem itself, a concrete application of this algorithm will also be required in order to apply the theorem to derive a statement of the form "The number of roots of $f(x)$ in the interval (a, b) is n." This is to say that we want to apply Sturm's theorem to derive a statement like

(1.11) $x^5 - 3x - 1$ has three roots in $(-2, 2)$.

we must actually calculate the Sturm function chain for $f(x) = x^5 - 3x - 1$, and this requires carrying out Euclid's algorithm. There are, of course, any of number of ways to derive (1.11) without carrying out this process, e.g., we might use Descartes' test or factor $f(x)$ explicitly. However, both the original discovery of Sturm's theorem (as detailed in Sturm's own account [139]) and the way it is motivated in contemporary textbooks (e.g. [19]) suggest that its significance derives precisely from its practical utility in deriving statements like (1.11) without the need to apply other methods of this sort.

The proof of Sturm's theorem is thus similar to that of König's Lemma in that, when written out in full, it would contain a specification of a mathematical procedure. In this case, however, the procedure so specified not only *can* be carried out in the sense of being effective, but in fact *must* be carried out in order to apply the theorem to specific instances. This observation again motivates us to inquire whether either of these features requires that we reevaluate whether Euclid's algorithm must be acknowledged to be a mathematical entity itself in order to account for the significance conventionally attached to Sturm's theorem.

Although matters are more complicated in this case here than in the case of König's Lemma, I believe that this question must again be answered in the negative. This conclusion appears to contradict the impression – which is likely to be reported by working

3. I will return to consider the original numerical form of Euclid's algorithm in greater detail in the next section.

mathematicians – that Sturm's theorem is somehow *about* how Euclid's algorithm may be applied to $\mathbb{R}[x]$. However, in attempting to make this point precise, we encounter the immediate problem that the *statement* of Sturm's theorem does not contain any term or quantifier which obviously is intended to refer to Euclid's algorithm. And there is also no reason to suppose that a procedure-denoting term would be introduced where we to render Theorem 1 into whatever format might be taken to be its logical form.[16] And as such, if we continue to apply traditional criteria of ontological commitment, there thus appears no way in which a commitment to recognizing Euclid's algorithm can simply be "read off" from Sturm's theorem itself.

To this it might replied that although no relationship between Sturm's theorem and Euclid's algorithm can be extracted from the linguistic form in which the former is stated, criteria of commitment in mathematics should be applied not only to the statement of its theorem, but also to the techniques which are required in their proofs. As I have already indicated, the conventional proof of Sturm's theorem *does* explicitly refer to Euclid's algorithm. For this reason it seems reasonable to expect that the most straightforward formalization of the proof of Sturm's theorem would require us to represent Euclid's algorithm as a mathematical object about which a property (such as correctness or totality) could be formally established. Such a formalized proof would thus contain either a term referring to an algorithm or an explicit description of this procedure.

In the next section I will propose that our acceptance of certain mathematical statements whose demonstration depends on computational methods *does* bring with it a commitment to recognizing that mathematical procedures are mathematical objects. However, it should

[16] To be absolutely secure in this, we would, of course, need to possess not only a well-developed theory of logical form applicable to complex algebraic statements, but also a means of regimenting mathematical prose appearing such as that appearing in (1) as a statement of first- or potentially higher-order logic. As long as we are willing to quantify over quasi-linguistic items like polynomials, there seems to be no particular obstacle impeding the performance of the second task. However, there is room for debate about whether algebraic results like Sturm's theorem are to be interpreted as corresponding to statements about extensional functions on $\mathbb{R}^n$ or about the polynomials by which such functions are represented. The former alternative is, of course, more in keeping with the standard extensional ideology of classical mathematics. But it should also be clear that analyzing Sturm's theorem in this manner would require the introduction of quantifiers over functions which make for a poor fit between the grammatical structure of Theorem 1 as stated and the formula assigned as its logical form. It seems safe to say, however, that no divergence tolerated in service of interpreting this statement extensionally would introduce an expression denoting Euclid's algorithm or any other procedure.

also be fairly obvious that nothing we have thus far seen about how Sturm's theorem is demonstrated provides a reasonable basis for reaching such a conclusion. One reason for this is that despite the fact that there is a well known proof of Sturm's theorem which employs Euclid's algorithm in the manner just discussed, this is by no means the only acceptable proof of this theorem. And a novel proof may, of course, fail to refer to this or any other mathematical procedure. Thus despite the conventional sentiment that there is some relation in content between Sturm's theorem and Euclid's algorithm based on the proof that we currently accept, there seems to be little reason to think that this connection is anything more than heuristic.

As things stands, we have thus yet to see any considerations which suggest that a realistic interpretation of classical mathematics requires an ontological commitment to algorithms. But in this section, I have concentrated on instances in which reference to procedures appears to be involved with the proof of what might be called *general* mathematical propositions – i.e. statements which express either that an object with a particular property exists or that every member of a class of mathematics objects has a given property and which thereby would be most naturally formalized as starting with an initial quantifier. But as we saw at the end of section 1, algorithms are standardly used to derive statements that might be called *singular* mathematical propositions – i.e. statements which have either the simple subject predicate form $P(\overline{a})$ or the form $f(\overline{a}) = b$ which state that a function has a given value.

Now although (1) obviously has the former form, (1.11) has the latter. A typical demonstration of the latter statement (as might appear as an example or an exercise in a textbook) might proceed by simply citing Sturm's theorem and then instantiated $f(x)$ as $x^5 - 3x - 1$. What is of most interest is the part of such a demonstration which would follow this step. For in order to apply Sturm's theorem to $f(x)$, the Sturm function for $x^5 - 3x - 1$ must be calculated so that we may compute the values of $\sigma(2)$ and $\sigma(-2)$. And as mentioned above this would typically be accomplished by applying Euclid's algorithm to this polynomial.

A fully explicit but informal derivation of (1.11) would thus feature not one but two

references Euclid's algorithm. In the first of these, the algorithm would have to be presented in a manner which allows us to reason about it formally so as to prove that it is total and correct. And in the second, the algorithm would have to be presented in a manner that allowed it to be carried out with respect to input $f(x)$. I suggested above that the first invocation of Euclid's algorithm is potentially eliminable in the sense that it is possible that we might provide a proof of Sturm's theorem which did not proceed by citing Euclid's algorithm in order to guarantee the function chain associated with an arbitrary polynomial has the appropriate properties. We can, for instance, envision an indirect or non-constructive proof that such a chain existed without providing any indication of how it might be constructed. But on the other hand, it is also clear that without access to *some* method for explicitly calculating the members of this sequence, we would be unable to apply the theorem to derive singular statements like (1.11).

This observation raises the following question: are there mathematical propositions which we currently take ourselves to be justified in believing but whose demonstrations ineliminably depend on actually having carried out the steps of an algorithm? While we have seen that this is probably not the case for general propositions like Sturm's theorem, it is as least plausible that certain simpler propositions like (1.11) do have this property. In order to see why this is so, we will have to examine an aspect of the role of computational methods in mathematics which I have thus far mentioned only in passing – i.e. that not only may a procedure A be justifiably used to determine the values of a function $h : A \to B$, but it may also be the case that there are values $a \in A$ such that using A is the only practical way we have to determine the value of $h(a)$. This is typically the case when calculations which employ A turn out to be *shorter* than other known means of determining the values of f. In the next Section I will argue that there are choices for h, a and A such that it is uncontentious that we know the value of $h(a)$, but for which we would be unable to account if we did not acknowledge that A was a genuine mathematical object.

1.4 A problem about mathematical knowledge

My goal in this section is to demonstrate the existence of mathematical statements which we currently take ourselves to be justified in believing but whose justification we cannot

account for unless we are are willing to acknowledge that the algorithms by which they have been derived are mathematical objects in their own right. Although such statements are to be found throughout contemporary mathematics, my paradigm example will have the form of identities of the form $h(x, y) = z$, where $h(x, y)$ is an explicitly defined arithmetic function. My contention will be that there are such functions such that at any stage in the development of our mathematical knowledge, there will be particular values n, m with the following properties: 1) it will be uncontentious within the mathematical community that we know a statement of the form $h(n, m) = q$; 2) the only justification for this fact that can be cited at the time in question will involve the fact that an algorithm A has been applied to n, m to yield q. It thus follows that if we are to accept that our belief that $h(n, m) = q$ is actually justified, the process of applying A to pairs of natural numbers to yield other numbers as output must itself be subsumed to a conventional mathematical derivation. Over the course of the section I will argue that this means that A itself must be regarded as a mathematical object.

The necessity of proving an algorithm correct before it can be used to learn the values of a function which it has been introduced to compute is a widely recognized requirement in theoretical computer science. This is reflected prominently both in informal algorithmic analysis (in the sense of [72]) and also in the field known as *program verification* (in the sense of [4]). In the first instance, informal mathematical techniques are used to prove that the execution of an algorithm A, generally presented in pseudocode, determines a prespecified mathematical function $f(x)$. Although often routine, such proofs can occasionally be quite involved in the sense that the claim that A is indeed correct with respect to $f(x)$ can both be non-obvious and remain open for some time. In the context of formal program verification, such informal reasoning is eliminated by using formal programs to represent algorithms and reasoning axiomatically about the results of their execution.

A number of interesting issues arise due to the interplay of these two approaches to proving correctness. On the one hand, there appears to be a consensus that axiomatic proofs of correctness are to be preferred to formal ones because they replace informal procedural notions (e.g. the notion of one computational step being performed *before* another in the course of carrying out an algorithm) with precise mathematical ones (e.g. descending

in a well-ordering of computational states). This sentiment is explicitly expressed in the well-known papers of Hoare [62] and Floyd [116] which are credited as founding program verification as field. But such proofs are often impractical to construct in practice. This is due in large part to the fact that the process of expressing an informally stated algorithm of even moderate complexity into a formal programming language often complicates the task of reasoning about it. In particular, since formal programs often involve reference to programming language-specific features like pointers, formal proofs of correctness often obscure the central mathematical ideas occurring in informal proofs. For this reason, it has occasionally been argued (cf, e.g., [33]) that the formal verification of an algorithm adds little to the sort of justification for its application which is conferred by a formal correctness proof.

Two topics which are particularly germane to the topic of this Chapter tend to be overlooked in technical and non-technical literature on verification. The first of these is whether we need to acknowledge that computational methods are anything more than time saving or heuristic expedients in the conduct of mathematical practice. In particular, it is often efficient to determine the value of a function f at an argument a by applying an algorithm A to a than by explicitly deriving a statement of the form $f(a) = b$ in the appropriate axiom system. But this observation leaves open whether the our knowledge of $f(a) = b$ depends on having access to A, or whether all such knowledge must ultimately be justified on the basis of traditional deductive methods.

The second overlooked issue concerns the status which must assigned to A in order to accept that a given form of correctness licenses its application in concluding that $f(a) = b$. For as we will see in detail below, in order to prove that an algorithm A is correct with respect to f requires that a mathematical model M_A of A be constructed which can be reasoned about within mathematics. But even it can be shown classically that the application of M_A induces the function f, this does not in and of itself address why we are justified in using the algorithm A to compute f. In particular, A may been specified informally in a manner quite distinct from that in which M_A is defined. And as such, an auxiliary argument to the effect that M_A accurately reflects the computational properties of A must be given if a mathematical proof involving the former is to be accepted as part

of a justification for the use of A in computing values of f.

Over the course of this section, I will highlight several complexities involved with constructing mathematical representations of informally specified algorithms so as to prove that they are correct. In general, however, not only is it difficult to assign a precise sense to what it means to say that M_A represents A, but it is unclear what it means to claim that one such object more accurately represent a given algorithm than another. At least in paradigmatic cases, there is thus little basis on which to block the suggestion that algorithms should simply be identified with the mathematical objects we construct to represent them in course of constructing correctness proofs. And since it appears that such proofs are needed to explain the extent of our mathematical knowledge, it appears that operational and explanatory forces may be combined in order to make a case in favor of algorithmic realism.

The argument just outlined is both indirect and also turns on a number of potentially contentious assumptions about the nature and extent of mathematical knowledge. Nonetheless, I believe that it corresponds not only to the strongest case which can be given for algorithmic realism but that it also touches most directly on why this view is of importance to contemporary mathematics. Below, I will examine a particular case of this argument involving an explicitly defined function f and several algorithms which might be used to compute it. But before entering into this degree of detail, it will be worthwhile to first briefly layout the steps in the overall argument.

1) Let f be any total function on an infinite domain X – for convenience we can assume $X = \mathbb{N}$ – and let I be any group of human mathematical agents who we will suppose are interested in computing the values of f. Since the domain of X is infinite, and the agents in I are of finitary cognitive capacity, it follows that any time t in the complete history of the mathematical activity of the members of I, these agents will only know the value of f on a finite set of arguments $D_0 \subseteq \mathbb{N}$. The exact extent of D_0 will be determined by a number of practical exigencies both about the members of I, the methods of mathematical proof which they accept and the way in which f has been defined.

2) Suppose that the members of I share a common mathematical language $\mathcal{L}$ and theory T and that the function f has been introduced into $\mathcal{L}$ by an explicit definition θ_f. A traditional deductive derivation of a statement of the form $f(n) = m$ by which the members of I might learn about the values of f will correspond to a proof in $T + \theta_f$ in which the definition of f is first expanded as θ_f and then it is shown that m is the unique number such that $\theta_f(n, m)$ holds. Clearly the construction of such a derivation by a member of I is sufficient for the inclusion of n in D_0. But at the same time, the set of numbers $D_1 \subseteq D_0$ for which the members of I can actually construct such a proof of the sort just described in $T + \theta_f$ will be limited by the length of these proofs. In particular, suppose that the length of the shortest derivation of $f(n) = m$ of the sort just described in $T + \theta_f$ is given by a function $\ell_1(n)$. We may also assume that there exists an upper bound on the length of the derivations the members of I could have constructed up to time t which given by a function $b(t)$. For all times t, we should thus have $D_0 \subseteq \{n \ : \ \ell_1(n) < b(t)\}$.

3) Now suppose that there also exists an algorithm A which is believed by the members of I to compute f. (They might believe this, for instance, either because of the way in which A was specified or inductively on the basis of comparing the values produced by A with those for which they can prove $f(n) = m$ for $n \in D_1$.) Suppose also that the number of computational steps which it takes for A to produce an output on n is given by $\ell_2(n)$ and that for some $n_0 < b(t)$, we have that $\forall n > n_0[\ell_2(n) < \ell_1(n)]$. In this case, it is possible that the set D_2 of values n for which some member of I has used A to compute the value of $f(n)$ at time t is a *proper* superset of D_1. In this case the numbers $n \in D_2 - D_1$ will be ones for which the members of I believe that a statement of the form $f(n) = m$ is true, but for which they have not constructed a standard deductive proof in $T + \theta_f$ of this statement (in virtue of its prohibitive length), but rather have derived m as the output of A.

4) Suppose we take t to be the present day and I to be the set of all working mathematicians. A standard necessary condition on the proper analysis of "i knows that φ" is that i knows (or with sufficient effort could construct) a proof of φ. But in the case

$\varphi \equiv f(n) = m$ for $n \in D_2 - D_1$, we have supposed that "direct" proofs of φ in $T + \theta_f$ (i.e. those that proceed by directly demonstrating that $\theta_f(n, m)$) are too long to grasp. If we want to understand how (if at all) we are justified in believing that $f(n) = m$, then we must do so by providing a mathematical argument that justifies the use of A to compute the values of f. However, it we suppose that T represents our entire mathematical background theory, then this argument must itself be formalizable within $T + \theta_f$.

5) The mathematical argument we must give should have the conclusion

   (*) For all $n \in \mathbb{N}$, the value obtained by applying A to n is equal to $f(n)$.

   But note that as written, (*) cannot be taken to abbreviate any sentence in $\mathcal{L} + \{f\}$. For based on what has been said thus far, we have no idea whether how (or even if) the procedure A and the notion of applying A to input n can be formalized in $T+\theta_f$. In order to do so, we must present an independent argument that there exists a $T + \theta_f$-definable structure M_A which can be taken to represent A and a $T + \theta_f$-definable functional $App(M_A, x)$ which can be taken to represent the operation of applying A to $x \in \mathbb{N}$. If we are able to do so, then (*) may be formalized in $T + \theta_f$ as

   (**) $\forall n[App(M_A, n) = f(n)]$

   Such a statement is standardly described as a formal statement of the *correctness* of A with respect to f as defined by θ_f.

6) The provability of (**) in $T + \theta_f$ constitutes a *necessary* condition on our ability to justifiably learn the values of f through the application of A. But the relationship between M_A and A (and also between the informally defined function given by the expression "the value obtained by applying A to n" and the formally defined function $App(M_A, n)$) must be further analyzed before it can also be accepted as a sufficient condition. It particular, we must show that M_A is what we might colloquially call a *faithful representation* of A. The exact significance of this requirement will vary depending on what constraints our informal practices determine are the essential or computationally invariant properties of A. (I will argue in subsequent chapters that paradigmatic examples of such properties include running time complexity and the use

of certain data structures.) For note that it is only if such additional facts about A may be demonstrated that the sort of warrant which we wish to secure for using using A to learn compute the values f is adequately formalized by (**).

7) If we take as fixed the assumption that we do in fact have *knowledge* of the statements $f(n) = m$ for $n \in D_2 - D_1$ (and this knowledge is on an epistemic par with that we have of mathematical propositions which we have derived through traditional deductive proofs), then from 1)-6) it follows that we must be able to demonstrate both (**) and these additional facts about M_A required by 6) within $T + \theta_f$. If we are able to justify the application of A in the manner just described, we will have thus also shown that discourse about A and it properties is systematically replaceable with talk about the subject matter of $T+\theta_f$ (which we can assume is exhausted by traditional mathematical objects like numbers and sets). In this case we would thereby have developed the basis of what is standardly referred as an *ontological reduction* of algorithms to mathematical objects – i.e a means of showing that individual algorithms may be uniformly associated with individual mathematical objects. This, however, is precisely the view which has been identified as algorithmic realism.

Before the foregoing argument can be accepted as an argument for algorithmic realism, a number of gaps need to be filled in. In addition to providing concrete examples of f, θ_f, T and A, we must, for instance provide an abstract characterization of the difference between what I have been referring to as a deductive derivation of a mathematical statement and a derivation carried out by applying an algorithm. Another related problem concerns how we may say definitively of a particular number that it falls in the set $D_2 - D_1$ (i.e. that it is such that we take ourselves to know the value of $f(n)$ on the basis of having applied A but not by virtue of having derived the statement $f(n) = m$ in $T + \theta_f$). However, gaps of this sort become somewhat easier to fill once examples for the various parameters have been specified.

To this end, consider the greatest common divisor [gcd] function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ – i.e. the function which for the arguments n, m returns the largest natural number dividing both. In what follows, I will use g to denote the gcd function in extension – i.e. the set of

pairs $\langle (n, m), q \rangle$ such that q is the gcd of n and m. Of course this function is sufficiently important in elementary number theory to be assigned is own symbol – i.e. $gcd(x, y)$. Our subsequent discussion will be largely focused on statements of the schematic form

(1.12) $gcd(n, m) = q$

We will be primarily interested in epistemic of statements like

(1.13) $gcd(43928, 27149) = 17$

which may be derived from (1.12) by the substitution of numerals (of various forms) for n, m and q – i.e. whether such statements are known to mathematical agents resembling ourselves and if so, how they might be justified. But before we can begin to explain what it means to say that a mathematical agent knows a statement such as (1.13), we must first examine how functional symbols like $gcd(x, y)$ are treated in conventional mathematical discourse. We must decide, for instance, whether $gcd(x, y)$ should be treated as a basic functional expression or be taken to be defined in terms of other arithmetic expressions.

Although this is a fundamental distinction from the standpoint of constructing a logical system in which to reason about statements like (1.12), it is often unclear in the practice of mathematics when a given function expression is being treated as primitive notion and when it is being treated as defined in terms of other symbols in the language. In the case of elementary number theory, however, there is a tradition to regard only the successor function, addition and multiplication as primitive functions and regard other functions such as subtraction and exponentiation as defined. And since our informal understanding of the function g itself would normally be stated in terms of these functions, it seems reasonable to start out by regarding $gcd(x, y)$ as a defined expression.

In the context of a formal theory of arithmetic, such a definition will take the form $gcd(x, y) =_d ft(x, y)$ where $t(x, y)$ is an expression in the language containing symbols denoting the successor, addition, and multiplication functions. Before deciding on the language in which such a definition should be given, however, it will be useful for the moment to remain at the level of informal number theory wherein $gcd(x, y)$ might be defined as

(1.14) $gcd(x, y) =_{df} \max\{z \in \mathbb{N} \ : \ z|x \wedge z|y\}$

While the right-hand side of this definition relies on the non-arithmetic notion of set comprehension, the use of this and related set theoretic notations is common throughout informal mathematics. Taken in this context, (1.14) appears to do as good a job as any formula at directly expressing our original definition of g.

Another advantage (1.14) has over many standard definitions of otherwise simple arithmetic and analytic functions is that is that an intuitive understanding of the terms it employs allows us to see immediately how it is possible to compute the value of $gcd(n, m)$. For note that since $z|x$, $z|y$ implies $z \leq \min(x, y)$, then it can easily be seen that the set over which the maximum needs to be taken in (1.14) is finite. Hence in order to compute $gcd(n, m)$, it suffices to explicitly check for each of the numbers $z = 2, 3, \ldots, min(n, m)$ whether $z|n$ and $z|m$.

This procedure is likely to correspond to the manner in which we would proceed were we to find the definition (1.14) written in a textbook and were then asked to compute the value of $gcd(12, 5)$. For while this procedure is not itself directly expressed by (1.14), it may be readily extracted by operationalizing our understanding of the terms it contains. One way we might describe the outcome of this process of operationalization is via the following procedural specification:

(1.15) NaiveGCD(n, m)

Step 1: Let q = 1;

Step 2: Let i = 2;

Step 3: If i $\geq$ min(n,m) then output q and halt;

Step 4: If i | n and i | m, then let q = i;

Step 5: Let i = i + 1;

Step 6: Goto to step 2.

(1.15) is our first example of a procedure specified in the quasi-formal idiom of *pseudocode* – i.e. a hybrid of mathematical English and the syntax of formal programming languages which is most often employed in mathematics and computer science to specify

procedures more precisely than is customary in standard prose. I will discuss the conventions by which such specifications are standardly interpreted at greater length in Chapters 2. For the time being, however, we may understand (1.15) as a set of imperative statements $\alpha, \beta, \ldots$ which specify that certain mathematical operations are to be performed by an agent who elects to carry the procedure so denoted. These operations are joined together by groups of constructs known as *control structures* which serve the grammatical role of conjoining instructions and function semantically as directing an agent carrying out the procedure to perform the conjoined instructions either in sequence (as denoted by "$\alpha;\beta$"), conditionally (as denoted by $\mathsf{If}\ \varphi\ \mathsf{then}\ \alpha$) or to switch control to another instruction (as denoted by $\mathsf{Goto\ to\ step}\ \ell$).

Following the conventions of modern computer science, I will also begin referring to procedures specified in this manner via the proper name NaiveGCD. It will be my ultimate contention that in nearly all instances in which a human mathematical agent believes a statement like (1.13), this belief is grounded in carrying out either NaiveGcd or another more efficient procedure to be described below. As a first step in this argument, the following two questions may now be posed directly: 1) what does it mean for an agent i to "carry out" a procedure like NaiveGCD?; and 2) given that i has carried out such a procedure with input n, m and obtained output q, how can this result serve to justify i's belief that $gcd(n, m) = q$. The force of the second question becomes apparent as soon as we recognize that as close as we may think the relation between the procedure NaiveGCD and the definition (1.14) to be, the sequence of corresponding steps which would arise by carrying out the former is not likely to resemble a deductive proof of $gcd(n, m) = q$ based on the latter. If we are going to accept applications of NaiveGCD as forming part of the evidentiary basis for belief in such statements, an additional argument must be given which shows that the value determined by applying this procedure to n, m will coincide with that determined by the definition of $gcd(n, m)$.

Although it is ultimately straightforward to provide such an argument, doing so requires that we take several steps towards formalizing both what it means to carry out NaiveGCD and also what it means to derive a statement of the form $gcd(n, m) = q$ via conventional mathematical proof. To this end, first note that what we might informally

describe as a *calculation* via this procedure can be taken to correspond to sequences of tuples of $\langle n, m, i, q, \ell\rangle$ where n and m correspond to the input variables n, m, i to the current value of i, q to the current value of q and ℓ to the current line number being carried out. I will refer to such tuples as the *computational states* of NaiveGCD. As I will discuss in more detail below, a complete calculation carried out in accordance with this procedure may then be taken to correspond to a sequence of tuples of the form $\langle\langle n, m, i_0, q_0, \ell_0\rangle, \langle n, m, i_1, q_1, \ell_1\rangle, \ldots, \langle n, m, i_{k-1}, q_{k-1}, \ell_{k-1}\rangle\rangle$ where the transition between computational states is mediated by carrying out one of the component instructions of NaiveGCD and $k-1$ corresponds to the first tuple in such a sequence for which no such transition is defined.

Suppose we let $exec_{\text{NaiveGCD}}(n, m)$ denote the sequence of states induced by the operation of NaiveGCD on n, m. Stated in these terms the fact that this procedure may be used to compute the value of $gcd(n, m)$ may now be taken to be equivalent to the two following assertions:

(1.16) i) For all $n, m \in \mathbb{N}$, the sequence of computational states $exec_{\text{NaiveGCD}}(n, m)$ whose first component is $\langle n, m, 1, 1, 1\rangle$ is finite.

ii) For all $n, m \in \mathbb{N}$, if $\langle n, m, i_{k-1}, q_{k-1}, \ell_{k-1}\rangle$ is the last member of $exec_{\text{NaiveGCD}}(n, m)$, then $q_{k-1} = gcd(n, m)$.

Take together, these statements express that NaiveGCD is correct with respect to the function denoted by $gcd(x, y)$. I have already noted the specification of this procedure is so closely related to definition (1.14) as to give the appearance that it would be trivial to prove (1.16i,ii). But of course corresponding claims for some other algorithm need not have this property. And since there are a number of subtleties which arise only when we attempt to say in precise terms what is required to *prove* such statements from more basis assumptions, it will be useful to examine further what is involved in proving the correctness of NaiveGCD.

The first thing we must do is to take a step backwards with respect to the manner in which we have decided to introduced $gcd(x, y)$ to our mathematical language. For note that any formal proof of (1.16ii) will require that this functional expression be treated as defined

in some definite manner. And although (1.14) reflects how this term might be introduced in everyday mathematical practice, it is difficult to translate this definition directly into a formal language. It will thus be useful to choose as our "official" definition of $gcd(x, y)$ a statement in a traditional first-order language which reflects as much as possible the mathematical content of (1.14). A standard choice in this regard would be something like

$$(1.17)\quad gcd(x,y) \;=_{df}\; \iota z.[\exists u \exists v(z \times u = x \;\wedge\; z \times v = y) \wedge \forall w[\exists u \exists v(w \times u = x \;\wedge\; w \times v = y) \rightarrow w \leq z]]$$

I will treat this statement as a definition of a new function symbol given over the language $\mathcal{L}_A = \{\mathtt{0}, \mathtt{1}, <, s, +, \times\}$ of first-order arithmetic. Such a statement should be viewed as an axiom which may be adjoined to a background arithmetic theory T which is sufficient to formalize elementary number theoretic reasoning.

I will start out by assuming that T extends first-order Peano arithmetic $[PA]$. So as to facilitate the introduction of mathematical natural function definitions, it will also be useful to assume that the e $\mathcal{L}_A^+$ over which T is formulated contains a definite description operator axiomatized in the standard manner – i.e. by taking as logical axioms all instances of $\varphi(\iota y.\psi(\overline{x}, y)) \leftrightarrow \exists y[\psi(\overline{x}, y) \wedge \forall z[\psi(\overline{x}, z) \rightarrow z = y] \wedge \varphi(y)]$. Using this device, a functional symbol f_φ can then be added to $\mathcal{L}_A$ by the axiom $\forall \overline{x} \forall y(\mathtt{f}_\varphi(x) = \iota y.\varphi(\overline{x}, y))$ whenever $T \vdash \forall \overline{x} \exists! y \varphi(\overline{x}, y)$. If we let $\theta(x, y, z)$ correspond to the formula on the right-hand of (1.17), then we may easily prove the corresponding statement for this formula in T. I will thus assume that $\mathcal{L}_A$ contains a functional expression $\mathtt{gcd}(x, y)$ whose interpretation is governed by the axiom taking (1.17) as an axiom.

I have now presented a basic mathematical language $\mathcal{L}_A$ and theory T in which the functional expression $gcd(x, y)$ is a formal term. We can thus begin the process of formalizing statements like (1.13) and (1.16) so as to begin to answer three of the questions have been posed previously: 1) what does it mean to formally prove the correctness of NaiveGCD with respect to a particular definition of $gcd(x, y)$?; 2) what does it mean to provide a formal deductive derivation of a statement of the form $gcd(n, m) = q$?; and 3) how does a proof of this sort differ from a calculation carried out by a procedure like NaiveGCD? Since answering the second and third questions will lead us to expand $\mathcal{L}_A$ and T in various

ways which facilitate answering the first, I will begin there.

The first obstacle we face in answering 2) and 3) is that we cannot yet regard (1.13) as a statement of $\mathcal{L}_A$. For note that we would conventionally say that (1.13) contains numerals denoting each of the numbers 43928, 27149 and 17 and in the context of this statement these numerals are treated as grammatically primitive expressions. $\mathcal{L}_A$, however, only contain constants $\mathtt{0}$ and $\mathtt{1}$ corresponding to the natural numbers 0 and 1. This means that in order to refer to the numbers 43928, 27149 and 17 in $\mathcal{L}_A$, we must employ complex terms to denote them. While there are obviously many $\mathcal{L}_A$ which perform this function, the canonical choice is to use the unary numeral $s^n(\mathtt{0})$ to refer to the natural number n.

Relative to this convention (1.13) would rendered as the $\mathcal{L}_A$ statement

(1.18) $\mathtt{gcd}(s^{43928}(\mathtt{0}), s^{27149}(\mathtt{0})) = s^{17}(\mathtt{0})$

It may appear that (1.18) does a reasonable job of expressing (1.13). But it must be kept in mind that the statement (1.18) is not *itself* a sentence in the official syntax of syntax $\mathcal{L}_A$, but rather a metalinguistic abbreviation for such a sentence. And since the sentence which (1.18) abbreviates will contain the terms of the form $s(s(\ldots(\mathtt{0})\ldots))$ denoting each of the numbers 43928, 27149 and 17, it will be over $200,000$ symbols in length. And although this will be a perfectly well-formed sentence of $\mathcal{L}_A$, it is doubtful that any human mathematical agent could even survey this sentence let alone dependably grasp that it expressed (1.13). Thus if our desire is to formalize this statement in some manner so that its formal role in T reflects its epistemic significance in our informal mathematical practices, we need to refine $\mathcal{L}_A$ so that (1.13) can be expressed more compactly.

Accomplishing this in an intuitively satisfactory manner turns out to be a surprising elusive goal. It is not sufficient, for instance, to simply introduce three new constant symbols – say $\mathtt{43928}$, $\mathtt{27149}$ and $\mathtt{17}$ – into $\mathcal{L}_A$. For although this would allow us to write a sentence

(1.19) $\mathtt{gcd}(\mathtt{43928}, \mathtt{27149}) = \mathtt{17}$

which contains a surveyable number of primitive symbols, the mere presence of such a statement in an arithmetic language is of little use unless T is also amended in some way to ensure its derivablity. This can, of course, be achieved by simply taking (1.19) to be an

axiom of T. But in this case, the sentence which we would have chosen to express (1.13) would have a single line derivation in T. T thus fails to accurately reflect the fact that (1.13) is not only likely to be informative to a human mathematical agent (in the sense that prior to deriving it, he may fail to realize that the term $gcd(43923, 27149)$ denotes 17), but also that it may require substantial mathematical effort on the part of such an agent to demonstrate that it is true.

If we wish derivations in T to reflect the epistemic significance of derivations in informal number theory, then some other means of codifying the contribution of numerical expressions to the meaning of $\mathcal{L}_A$ sentences must be found. One desirable refinement would be a means of reflecting the distinctive role which numerals play in our informal understanding of arithmetic identities like (1.13). Note, for instance, that our understanding of expressions like 43928 is not exhausted by the recognition that they are names for natural numbers. Rather we recognize 43928 as being a certain kind of numeral, in particular a member of the *denary* (i.e., base ten or Arabic) *numeral system*. And with this recognition comes an understanding that such a system includes both recursive rules which allow us to see this symbol as having internal structure as well as a means of using this structure to effectively determine what number it denotes. The treatment of numerals as constant expressions in conventional first-order languages overlooks both of these aspects of their conventional use.

If we wish to construct T and $\mathcal{L}_A$ so as to be maximally faithful to our informal modes of number theoretic reasoning, then a number of emendations must be made to both. At minimum, the basic inductive definition of $\mathcal{L}_A$ terms would have to modified so as to recognize both primitive symbols (i.e. the digits $\mathtt{0}, \ldots, \mathtt{9}$) and concatenations of such symbols as singular terms. And in addition to the familiar PA axioms governing the base cases for addition and multiplication (i.e. $\forall x[x + \mathtt{0} = x]$, $\forall x[x \times \mathtt{0} = \mathtt{0}]$), we would also need to have axioms summarizing the addition and multiplication tables for the digits $\mathtt{0}, \ldots, \mathtt{9}$. And in addition to this, we would also have to emend T with axioms which codify how we perform addition and multiplication on arbitrary denary numerals. Such axioms should somehow mirror the way in which the standard PA axioms governing addition and subtraction reflect a means of computing on unary numerals. For instance the axiom governing addition (i.e. (i.e. $\forall x \forall y[x + s(y) = s(x + y)]$) allows us to proof-theoretically

"work out" the value of arbitrary terms of the form $s^n(\mathtt{0})+s^m(\mathtt{0})$ by successive applications of this axiom along the and first-order equality rules.

In order to capture analogous forms of computation involving arbitrary numerals in T, we might attempt to express various familiar calculating techniques such as the familiar "grade school" algorithms for performing addition and multiplication via carrying as axioms. But note that although we may think of these methods as the canonical means of computing sums and products, this view reflects a certain narrow-mindedness with respect to other traditional computational techniques. For instance, the methods of *lattice* multiplication and *Russian peasant* multiplication (c.f. [18]) have many of the concrete advantages of the grade school algorithm and were widely used in Europe well into the nineteenth century. The existence of such alternative computational techniques makes it difficult to provide sharp criteria for deciding whether a given method is sufficiently elementary to warrant direct axiomatization as opposed to the sort of indirect formalization of the sort we hope to provide for NaiveGCD. As such if we wish T to reflect even some of the practical expedients embodied by our informal number theoretic practices, we must decide between a maximalist theory which enshrines all such methods as axioms and an arbitrary version which makes unprincipled distinctions between them.

For present purposes, I will circumvent this problems by defining $\mathfrak{L}_A$ and T in a manner which idealizes certain aspects of how we perform addition and multiplication but which still attempts to reflect the difficulties we would face in deriving a statement like (1.13) in practice. In particular, I will henceforth assume that $\mathfrak{L}_A^+$ contains a constant symbol $\mathtt{n}$ corresponding to every n natural number. *Pace* the foregoing, I will also assume that these symbols are treated as conventional constants in the sense of being unstructured. However I will additionally assume that T contains as axioms all true atomic statements involving these symbols and the basic function symbols denoted by $s, +$ and $\times$ – i.e., all true sentences of the form $s(t_1) = t_2$, $t_1 + t_2 = t_3$ and $t_2 \times t_2 = t_3$ for all terms t_1, t_2, t_3 over this expanded language, and similarly for inequalities.[17]

Having defined $\mathfrak{L}_A$ and T in the manner just described, not only is

[17] In model theoretic terms, this corresponds to augmenting T with the elementary diagram of the standard model of PA in which constants are introduced to stand for all members of $\mathbb{N}$.

(1.20) $\texttt{gcd}(\texttt{43928}, \texttt{27149}) = \texttt{17}$

now a well-formed sentence of $\mathcal{L}_A$, but T now also contains appropriate axioms governing the behavior of all of the symbols it contains. Since we have agreed to take (1.17) as the sole axiom pertaining to the functional symbol $\texttt{gcd}(x, y)$, it thus follows that to derive a statement of the $\texttt{gcd}(\texttt{n}, \texttt{m}) = \texttt{q}$ we must proceed by demonstrating that the terms $\texttt{n}, \texttt{m}$ and $\texttt{q}$ satisfy the right hand side of this definition. To simplify this task, let $\theta(x, y, z)$ denote the $\mathcal{L}_A$ predicate $\exists u \exists v[z \times u = x \wedge z \times v = y]$. Such a proof must then proceed by demonstrating a statement of the form

(1.21) $\theta(\texttt{n}, \texttt{m}, \texttt{q}) \wedge \forall w[\theta(\texttt{n}, \texttt{m}, w) \rightarrow w \leq \texttt{q}]$

As is true of all well-known proof systems for arithmetic, T has the property that a derivable formula φ will have infinitely many distinct proofs in the standard sense of line-by-line identity. It is thus impossible to use T itself to uniquely characterize the manner in which we might go about deriving a statement like (1.13). What we can do, however, is to ask whether the manner in which we would go about calculating the value of $gcd(43928, 27149)$ based on an understanding of (1.14) alone (i.e. without the use of shortcuts based on auxiliary lemmas about the properties of $gcd(x, y)$) can be naturally formulated as a derivation of T.

The answer is clearly affirmative since we may readily see that the sort of calculation via NaiveGCD (which was itself grounded in an informal understanding (1.14) may be readily mimicked in T. The first step in such a derivation would be to prove as a lemma the statement

(1.22) $\forall x \forall y \forall z[\exists u \exists v(z \times u = x \wedge z \times v = y) \rightarrow z \leq \texttt{min}(x, y)]$

where $\texttt{min}(x, y)$ is the function introduced by the axiom $\texttt{min}(x, y) = \iota z.((x < y \wedge z = x) \vee (z = y))$. Once this statement was derived, it is then straightforward to demonstrate in T that (1.21) is implied by the statement

(1.23) $\theta'(\texttt{n}, \texttt{m}, \texttt{q}) \wedge \forall w \leq \texttt{min}(n, m)[\theta'(\texttt{n}, \texttt{m}, w) \rightarrow w \leq \texttt{q}]$

where $\theta(x, y, z)$ corresponds to the formula derived from $\theta(x, y, z)$ by bounding its initial existential quantifies by $\texttt{min}(x, y)$. Such a derivation will take a constant number of steps

k_0, independent of n, m. Note also that because T contains axioms corresponding to the complete multiplication table on $\mathbb{N}$, the first conjunct of this statement may be demonstrated by simply presenting a number q such that $\theta'(\mathtt{n}, \mathtt{m}, \mathtt{q})$. Such a derivation will thus have constant length k_1, also independent of the values of n, m.

The only real work in demonstrating (1.23) is thus that of verifying that the second conjunct holds. But now note that if we attempt to derive the second conjunct directly (i.e. in accordance with its logical structure), then we must proceed by demonstrating the finite conjunction

$$(1.24)\ \ \theta'(\mathtt{n}, \mathtt{m}, \mathtt{2}) \rightarrow \mathtt{2} \leq \mathtt{q} \wedge \ldots \wedge \theta'(\mathtt{n}, \mathtt{m}, \mathtt{i}) \rightarrow \mathtt{i} \leq \mathtt{q} \wedge \ldots \wedge \theta'(\mathtt{n}, \mathtt{m}, \mathtt{r}) \rightarrow \mathtt{r} \leq \mathtt{q}$$

where $\mathtt{r}$ corresponds to the value of $\min(\mathtt{n}, \mathtt{m})$. Since each of these conjuncts corresponds to a statement of the form $\exists u < \mathtt{min}(\mathtt{n}, \mathtt{m}) \exists v < \mathtt{min}(\mathtt{n}, \mathtt{m}) \ldots$, in the case that $i \nmid n$ and $i \nmid m$ they may themselves take as many $k_2 \cdot min(n, m)$ steps to derive in accordance with their logical structure (where k_2 is a constant independent of n and m). And thus a complete derivation of $\gcd(\mathtt{n}, \mathtt{m}) = \mathtt{q}$ in T of the indicted form will have overall length $k_0 + k_1 + \min(n, m)(k_2 \cdot \min(n, m)) = O(\min(n, m)^2)$.

Let $\mathcal{D}_T(n, m)$ correspond to the sequence of statements comprising a "naive" derivation of a statement of the form $\mathtt{gcd}(\mathtt{n}, \mathtt{m}) = \mathtt{q}$ of the sort just described. While we will see below that such a derivation is by no means the optimal means of deriving such a statement in T in the sense of length, $\mathcal{D}_T(n, m)$ is a reasonable reflection of how an agent might go about computing the value of $gcd(n, m)$ based on the definition (1.17) alone. And at the same time $\mathcal{D}_T(n, m)$ should also clearly qualify as what I referred to at the beginning of this section as a "traditional deductive derivation" in the sense that it corresponds to a deductive proof in an axiom system which arises as a minor extensions by definition of Peano arithmetic. As such, if an agent were to succeed in deriving (1.20) in this manner, there should be no further question as to whether his belief in this statement is justified for the simple reason that we accept that the axioms of T are true and that its rules of inference are valid.

As I mentioned above, it is at best unclear whether the same can be said about a calculation performed via NaiveGCD. For as noted, such a calculation is not composed

of statements either in English or in a formal language, but rather of structures which encode the present values of the variables occurring in the pseudocode specification of this procedure.[18] The components of such a sequence are thus not interpretable as having truth values and consequently do not bear inferential relationship to one another. As such, there is no clear sense in which we can speak of a calculation as being deductively valid or invalid.

But having now described the structure of the derivations of $\mathcal{D}_T(n, m)$, it is also apparent how narrow the gulf between derivations in T and calculations via NaiveGCD really is. For note that if an agent were to attempt to compute the value of $gcd(n, m)$ by constructing a derivation in T, he would naturally proceed by successively deriving each of the conjuncts in (1.24). Each such sub-derivation naturally corresponds to one iteration through the main loop of NaiveGCD consisting of steps 3 through 6. Thus in order to justify the use of this procedure for computing $gcd(n, m)$, all that is required is that we formalize the definition of $exec_{\text{NaiveGCD}}(n, m)$ in T. Upon doing this, we can proceed to both prove the formal correctness claims (1.16) and also formally demonstrate that the sort of structural relationship just described holds between $\mathcal{D}_T(n, m)$ and $exec_{\text{NaiveGCD}}(n, m)$.

Both tasks can most easily be accomplished in a background theory which allows us to reason directly about tuples and finite sequences. One way to facilitate the construction of correctness proofs is thus to expand T so that incorporates a portion of axiomatic set theory sufficient for reasoning directly about hereditarily finite sets. However, this leads to a variety of minor logical complications since we have to adopt a multi-sorted language to distinguish between sets and natural numbers. So for present purposes, it will be most convenient to work in a informal set theory and suppose that the constructions described may then be formalized in T through the use of standard techniques for arithmetically encoding finite sequences of numbers. This will allow us to concentrate on what is of

[18] If we were to attempt to analyze the intuitive notion of a calculation more generally, we would also have to take into account two additional features of such sequences: 1) in the paradigm case, calculations may be comprised of arbitrary (finite, discrete) configurations of symbols; 2) the constituents of such configurations may to be assigned a mathematical interpretation before the calculations in which they appear may be viewed as determined mathematical functions. Both points are raised already in Turing's [144] original paper on the analysis of the notion "finite mechanical procedure." While Turing himself ends up adopting a very narrow analysis of calculation, subsequent authors (e.g. Gandy [42], Seig and Byrnes [130]) have generalized along the lines manner just described. In Chapter 5 I will return to the question of characterizing these generalizations. However, for purposes of analyzing the simple procedures which are consider in this Chapter, the analysis given in the text will suffice.

most significance to out current discussion, namely showing that it possible to construct an explicit mathematical representation of NaiveGCD relative to which correctness may be demonstrated.

Although we will see in subsequent chapters that there is no unique or canonical form of representation which must be used for this purpose, it is straightforward to specify a finite combinatorial structure which is sufficient for proving correctness. One convenient choice for is what is known as a *transition system*. Although I will present a somewhat more general definition in the next Chapter, for present purposes we can take such a system to correspond to a pair $M_A = \langle \Sigma_A, \delta_A \rangle$ where Σ_A correspond to the class of computations associated with a given algorithm A and δ_A (the so-called *transition function* of M_A) is a function of type $\Sigma_A \to \Sigma_A$ which specifies how the state of A evolves in course of its execution. In particular, if σ corresponds to the state of M_A before a given instruction is executed, then $\delta_A(\sigma)$ will denote the state after this instruction has been executed.

We will see in the next chapter that in cases where A is specified by a formal program π, δ_A can be defined directly from the structure π by an operational semantics for the programming language over which this program is defined. In the case at hand, however, it easy to see we ought to have $\Sigma_{\text{NaiveGCD}} = \{\langle n, m, i, q, \ell \rangle \ : \ n, m, i, q \in \mathbb{N}, i \leq \min(m, n), \ell \leq 7\}$. And it also straightforward to see from (1.15) that δ_{Euclid} should be defined follows:

$$(1.25)\quad \delta_{\text{NaiveGCD}}(\langle n, m, t, l \rangle) = \begin{cases} \langle n, m, i, q, 2 \rangle & \text{if } \ell = 1 \\ \langle n, m, 2, q, 3 \rangle & \text{if } \ell = 2 \\ \langle n, m, i, q, 4 \rangle & \text{if } \ell = 3 \ \wedge \ i < \min(n, m) \\ \langle m, m, i, q, 7 \rangle & \text{if } \ell = 3 \ \wedge \ i = \min(n, m) \\ \langle n, t, i, q, 5 \rangle & \text{if } \ell = 4 \ \wedge \ (i \nmid n \ \vee \ i \nmid m) \\ \langle n, t, i, i, 5 \rangle & \text{if } \ell = 4 \ \wedge \ (i | n \ \wedge \ i | m) \\ \langle n, m, i + 1, q, 6 \rangle & \text{if } \ell = 5 \\ \langle n, m, i, q, 3 \rangle & \text{if } \ell = 6 \end{cases}$$

Having now seen an example of a transition system, it will now be useful to define several auxiliary notions which are applicable to such structures in general. To this end, let A be an

algorithm which takes inputs in a set X and produces outputs in a set Y. Let $M_A = \langle \Sigma_A, \delta_A \rangle$ be the associated transition system. For present purposes, we may assume that states $\sigma \in \Sigma_A$ have the form $\langle x_1, \ldots, z_n \rangle$ and that inputs $x =\in X$ correspond to sub-vectors of σ of the form $\langle x_1, \ldots, x_{in} \rangle$ ($in \leq n$). We may further assume that an output of A is given by a single component x_{out} ($out \leq n$) of σ. An *execution* of A on input $x \in \Sigma_A$ may now be defined to be a sequence of computational state $exec_A(x) = \langle \sigma_0(x), \sigma_1(x), \ldots, \sigma_{len_A}(x) \rangle$ where $\sigma_0(x)$ corresponds to the initial state of A with the first in values set to $x_1, \ldots, x_n$ and all $i > 0$, $\sigma_{i+1}(x) = \delta_A(\sigma_i(x))$ if this value is defined and $\sigma_{i+1}(x) = \sigma_i(x)$ otherwise. $len_A(x)$ is defined to be $k+1$ where k is least such that $\delta_A(\sigma_k(x)) = \sigma_k(x)$. We finally define the function computed by A on X to be given by $App_A(x) =_{df} \pi_n^{out}(\sigma(x)_{len_A(x)})$.

On the basis of these definitions, it is now straightforward to demonstrate the two claims (1.16i,ii) which comprise the formal statement of the correctness of NaiveGCD relative to the definition of (1.17). In order to demonstrate (1.16i), it suffice to show that $len_{\text{NaiveGCD}}(n, m)$ if defined for all $n, m \in \mathbb{N}$. This may be seen by observing i incremented in the states $5, 9, 13, \ldots, (4i+1), \ldots (4\min(n, m) + 1)$ of $exec_{\text{NaiveGCD}}$ (with from initial state $\sigma_0(n, m) = \langle n, m, 1, 1, 1 \rangle$). And from this it follows that we have

(1.26) $\forall n \forall m [len_{\text{NaiveGCD}}(n, m) = 4\min(n, m) + 3]$

In order to demonstrate (1.16ii), it suffices to show that if we set $out = 4$, then for all $n, m \in \mathbb{N}$, if $App_{\text{NaiveGCD}}(n, m) = q$ then $gcd(n, m) = q$. This can be shown by inductively by noting that for all $i \leq \min(n, m)$, the value of the parameter q in the state $\sigma_{4i+1}(n, m)$ is equal to the largest $j \leq i$ such that $j|n$ and $j|m$. And thus for $i = \min(n, m)$, it follows that $q = gcd(n, m)$.

Relative to conventional standards of mathematical rigor, the foregoing correctness proof would suffice to justify the use of NaiveGCD for computing the values of $gcd(x, y)$. But as already noted, this proof involved only finite combinatory reasoning, and as such, it may readily be formalized in the theory $T + \theta_f$. It is, moreover, possible to carry out this formalization in a manner such that the definitions of M_{NaiveGCD}, $exec_{\text{NaiveGCD}}$ and App_{NaiveGCD} are translated in definitions M^*_{NaiveGCD}, $exec^*_{\text{NaiveGCD}}$ and App^*_{NaiveGCD}. And on this basis we may then prove the translates of (1.16i,ii) directly in $T + \theta_f$.

Both of these are significant findings from the standpoint of algorithmic realism as they lend credence to the overarching claim that statements about algorithms may be systematically interpreted as referring to ordinary mathematical objects. But in addition to this, the manner in which the latter result is derived is significant because it also shows that we may demonstrate in T that we are justified in using NaiveGCD to determine the values of g. This is true not just in the global sense that the proof shows that $App_{\text{NaiveGCD}}(x, y)$ determines the same function as is denoted by the term $gcd(x, y)$, but also in the local or intensional sense that we can prove in T that there is a correspondence between sequences of statements in the deductive derivation $\mathcal{D}_T(n, m)$ and subsequences of states in $exec_{\text{NaiveGCD}}(n, m)$ of the sort described above.[19]

The example of $gcd(x, y)$ and NaiveGCD we have been considering answers the three questions posed above about the relationship between proofs of identities like (1.13) involving the former and computations of the latter which may also be taken to justify our belief in them. If we generalize away from the details of this example, we arrive at the conclusion that in order to justifiably employ an algorithm A to compute the values of a mathematical function f, we must typically construct a mathematical representation M_A of A by which A can be proven correct in the context of our mathematical background theory S. Once we have done so, we can also go on to provide a explanation of the relationship between A and the definition under which we have introduced f by reasoning about the operation of A internally in S using M_A. This leads us naturally to ask about the ontological relationship between M_A and A. For if we have succeeded in formalizing not only a definition of A but also in formalizing important procedural notions such as the definitions of application and running time, and shown that these definition have the correct properties in S, then this provides motivation for advancing the thesis that A may be simply identified with M_A.

It seems, however, that our views about the role which algorithms play in the practice of mathematics are more settled than any intuitions we might have about their ontological

[19] In particular, suppose that the order of statements in $\mathcal{D}_T(n, m)$ is fixed in the manner described above – i.e. a subproof of (1.22), followed by a subproof of (1.23) followed by a subproof of each of the conjuncts of the form $\theta'(\mathtt{n}, \mathtt{m}, \mathtt{i})$ appearing in (1.24) for $i <\leq \min(n, m)$. The latter class of derivations can now be correlated with the sub-sequences of $exec_{\text{NaiveGCD}}(n, m)$ of the form $\sigma_{3+4(i-1)}(n, m), \ldots, \sigma_{5+4(i-1)}(n, m)$ which perform the same computational labor (i.e. checking n and m for divisibility by i).

status. And for this reason it is doubtful that a clear cut case can be made for the thesis that A should be identified with M_A based only on considerations discussed thus far. Nonetheless, I do think an argument to this effect can be adduced which is grounded directly in the need to prove its correctness. In order to see this, we need to return to the schematic argument with which I began this. The central premise of this argument corresponds to the observation that if we refuse to accept that algorithms are mathematical objects in their own right, then we will be left without a means of justifying our belief in certain identities of the form $f(x) = y$ which, in the course of our mathematical practices, we take to be true. The paradigmatic example of such a statement is one whose conventional deductive proof is prohibitively long where the notion of "conventional deductive proof" may now be understood to resemble $\mathcal{D}_T(n, m)$ – i.e a derivation eventuating in $f(x) = y$ which proceeds by directly decomposing a fixed mathematical definition of $f(x)$.

Although I did not originally propose an example of such a statement, it should not be surprising that one is close at hand. In particular, as I will now attempt to demonstrate, (1.13) has exactly this property. To see this, it suffices to recall our estimate on the length of the derivation $\mathcal{D}_T(n, m)$. Although the exact values of the constants k_0, k_1 and k_2 appearing in this estimate will depend on the properties of the particular proof system on which T is based, each of these values must obviously be at least 1. It thus follows that the $\mathcal{D}_T(n, m)$ will be at least $\min(43923, 27149)^2 = 737,068,201$ lines long. I will assume that it is infeasible that any human agent could possible even survey or check such a derivation, let alone explicitly construct it "from scratch."

But in order for (1.13) to be a legitimate contender for the status of a verifiably true statement which is not known on account of such a proof, it must also the be case that we are able to calculate the value of $gcd(43923, 27149)$ by some other means. But note that it is obviously also possible to compute this value by applying the procedure NaiveGCD. As we now know, the length of the execution of this procedure on the input $43923, 27149$ will be given by $len_{\text{NaiveGCD}}(43923, 27149)$ (which in turn corresponds to the running time complexity of this algorithm as discussed in Section 1). By calculation given above, we have that $len_{\text{NaiveGCD}}(43923, 27149) = 108599$.[20] Such a calculation is obviously far shorter than

[20]Note also that the savings incurred by using NaiveGCD over $\mathcal{D}_T(n, m)$ is largely illusory. For note that

the length of $\mathcal{D}_T(n, m)$, but it too is presumably still far longer than any computation a human mathematical agent could reliably carry out by hand.[21]

This observation should not come as a surprise, since we have already noted that NaiveGCD is truly naive in the sense that it can be directly "read off" from the definition (1.14). And for this reason, it is also not surprising that calculating via this procedure is, at heart, no more efficient than explicitly deriving a statement of the form $gcd(\mathtt{n}, \mathtt{m}) = \mathtt{q}$ in an appropriately formulated axiom system. But as the reader has surely anticipated, there are considerably more efficient algorithms for computing the gcd function. One example of such a procedure is, of course, Euclid's algorithm. As described in the previous section, this procedure corresponds to a general method for computing greatest common divisors in an arbitrary Euclidean domain. When applied to $\mathbb{N}$, this procedure can thus be informally described as computing $gcd(n, m)$ by successively calculating the values $m_0 = n \bmod m, m_1 = m \bmod m_0, m_2 = m_0 \bmod m_1, \ldots$ until the point at which $m_{i-1} = 0$ and then returning m_i. A pseudocode specification of Euclid's algorithm would thus be as follows:

(1.27) Euclid(n, m)

Step 1: If m = 0, then output n and halt;

Step 2: Let t = n mod m;

Step 3: Let n = m;

Step 4: Let m = t;

Step 5: Goto to Step 1.

---

while the running time of NaiveGCD is $O(\min(n, m))$ as opposed to the $O(\min(n, m)^2)$ length of $\mathcal{D}_T(n, m)$, this improvement is rooted in the fact that we have elected to treat divisibility check as a primitive operation in our specification of NaiveGCD while also electing to leave such a predicate out of $\mathcal{L}_A^+$. The length of $\mathcal{D}_T(n, m)$ can thus be shortened by a multiplicative factor of $O(min(n, m))$ by introducing a modulus function $\mathtt{mod}$ into $\mathcal{L}_A$ and adding the axiom $x \bmod y = \iota r.[\exists q(q \times x + r = y) \wedge r < y]$ to T along with all true sentences of the form $\mathtt{n} \bmod \mathtt{m} = \mathtt{q}$ and $\neg(\mathtt{n} \bmod \mathtt{m} = \mathtt{q})$.

[21] If the reader doubts that it is truly infeasible to derive (1.13) by applying NaiveGCD, then this example can be replaced with one involving numbers which are several orders of magnitude higher – e.g. $gcd(560171761683, 346205188258) = 17$. Since such a statement would require over a trillion steps to verify using NaiveGCD and is thus even less likely to be verifiable by a human carrying out this procedure. Pursuant to the proviso issued in Section 1, I have thus have refrained from mentioning the potential use of a computer to carry out NaiveGCD in this section. But it should able be clear that for any physical computing device P, it will also be possible to choose values of n and m, such that P will be unable to carry out the corresponding calculation, due either to physical limitations on the numbers it can store or to the prohibitive number of steps it would require to carry out to completion.

I will assume that this specification succeeds in denoting a determinate procedure which I will subsequently refer to as EUCLID.[22] We are now in a position to demonstrate both of these facts mathematically. In particular, we may prove the following:

(1.28) a) $\forall m \forall n[App_{\text{EUCLID}}(n, m) = gcd(n, m)]$

b) $\forall m \forall n[len_{\text{EUCLID}}(n, m) \leq 5\lceil \log_2(\min(n, m))\rceil] + 6$

The first of these statements is standardly proven with the aid of the following number theoretic lemma:

(1.29) **Lemma** For all $n, m \in \mathbb{N}$ such that $0 < m$, $gcd(n, m) = gcd(m, n \bmod m)$.

On this basis, (1.28a) may now be proven as follows. Define sequences of natural numbers $n_0, n_1, n_2 \ldots$ and $m_0, m_1, m_2 \ldots$ as follows: i) for $i = 0$, let $n_0 = n$, $m_0 = m$ and ii) for $i > 0$, let m_i and n_i be the values assumed by the variables n and m immediately after Step 5 of EUCLID is carried out in successive iterations of the loop expressed by Steps 1-5. By the Lemma, we have that for all i, $gcd(n_{i+1}, m_{i+1}) = gcd(n_i, m_i)$. Note also that since if $n > m$ then $n \bmod m < \min(m, n)$, and thus we have $m_{i+1} < m_i$ for all i. Since the natural numbers are well-ordered, this sequence must thus terminate after a finite number of steps k. But due to the form of the conditional in Step 1, this can only happen if $m_k = 0$, at which point the procedure outputs n_k. Note, however, that $gcd(n_k, m_k) = gcd(n_k, 0) = n_k = gcd(n, m)$. And thus at the beginning of the kth pass through the loop, EUCLID will output $gcd(n, m)$.

---

[22] A close variant of EUCLID appears in Book VII of Euclid's *Elements*, although it is now thought to have been known at least two hundred years earlier (i.e. around 700 BCE, c.f. [18]) The procedure described by Euclid differed from EUCLID in two respects on which it is worth briefly commenting. First, Euclid described this procedure as operating on magnitudes as opposed to natural numbers. And second, the procedure was defined so that in the second step, the value m (described algebraically) was set equal to $t - m$ as opposed to $t \bmod m$. The first difference required Euclid to limit the application to magnitudes AB and BC which we were not relatively prime to one another (i.e., for which there existed a magnitude measuring both), as he did not take the unit magnitude as corresponding to a number. However, since the procedure works even where AB and BC possess no common measure, he could have stated the procedure so as to apply to the general case. The other difference between Euclid's presentation and (1.27) is more significant. In particular, the subtractive form of the Euclid's algorithm turns out to have worst case running time complexity $O(\min(n, m))$, as opposed to the $O(\log_2(\min(n, m)))$ complexity of the modular form. Computations performed by this procedure may thus be substantially shorter than those performed via NAIVEGCD.

The degree of formality employed in the argument just given is typical of that employed in correctness proofs given in modern textbooks on the analysis of algorithms such as [24], [125], [46]. But note that foregoing proof relies on an informal understanding of what it means to carry out the individual steps of EUCLID and also on a informal understanding of their modes of composition – i.e. what it means to execute Steps 1-5 in sequence, to pass control from one instruction to another the a Goto statement, etc.. If we wish to view the correctness proof outlined above as serving to justify the use of EUCLID to derive the values of $gcd(x, y)$, then we must show how it may be transformed into a purely mathematical argument which does not depend on an intuitive understanding of how EUCLID is specified.

Although somewhat less obvious than the corresponding correctness proof for NAIVEGCD, this may readily be accomplished using the definition of M_{EUCLID} given above.[23] And it should again be clear that this new correctness proof can be straightforwardly formalized within a mathematical theory like T which is capable of formalizing finite combinatorial reasoning. Thus despite the fact that the relationship between the specification of EUCLID and the definition of $gcd(x, y)$ is much less direct than in the case of NAIVEGCD, the epistemic warrant which we possess for using the former procedure to compute the values of this function is as every bit as strong as that we possess for the latter.

The dimension along which NAIVEGCD and EUCLID clearly differ is that of running time complexity. For as we may see from results (1.26) and (1.28b), the complexity of these procedures is respectively $O(\min(n, m))$ and $O(log_2(\min(n, m)))$.[24] While both of these functions are relatively slow growing compared to the complexities hierarchies we will

---

[23] Consider the partial ordering $<_E$ on $\Sigma_{\text{EUCLID}} \times \Sigma_{\text{EUCLID}}$ given by

$$\langle n, m, t, \ell\rangle <_E \langle n', m', t', \ell'\rangle \Longleftrightarrow (m < m' \vee (m = m' \wedge (\ell + 3) \bmod 5 > (\ell' + 3) \bmod 5))$$

We may proceed by showing that for all $\sigma \in \Sigma_{\text{EUCLID}}$, if $\delta_{Euclid}(\sigma) = \sigma'$, then $\sigma' <_E \sigma$. And since it may be readily seen that $<_E$ is well-founded, it follows that for all $n, m \in \mathbb{N}$, the execution $exec_{\text{EUCLID}}(n, m)$ is finite. Now applying (1.29) we again have that if $\delta_{Euclid}(\sigma) = \sigma'$, then $gcd(\pi^1_4(\sigma), \pi^2_4(\sigma)) = gcd(\pi^1_4(\sigma'), \pi^4_2(\sigma'))$. And from we may conclude that if $k = len_{Euclid}(n, m)$, then we must have $\sigma_k = \langle q, 0, 0, 6\rangle$ and that $App_{\text{EUCLID}}(n, m) = q = gcd(n, m)$ as desired.

[24] (1.28b) is derivable from a result known as Lamé's theorem which states that if $fib(i)$ denotes the ith Fibonacci number, $n > m$, and $m < fib(i + 1)$, then the sequence $m_0 = n \bmod m, m_1 = m \bmod m_0, m_2 = m_0 \bmod m_1, \ldots, 0$ is of length less than i. Since it may be shown that $fib(i) \leq \varphi^i/\sqrt{5}$ (where φ is golden ratio), this means that if $i = \min(n, m)$, then such a sequence will be of length $O(log_2(i))$. It may also shown that the exact upper bound $len_{\text{EUCLID}}(n, m) \leq 5\lceil \log_2(\min(n, m))\rceil + 6$ is achieved in cases where $n = fib(i + 1)$ and $m = fib(i)$ – e.g. $n = 34, m = 21$. For a proof of Lamé's theorem and related discussion of the historical development of analyses of Euclid's algorithm, see Knuth [72], I.1.2.8.

consider in subsequent Chapters, the difference of between logarithmic growth and linear growth should not be underestimated. This is already manifest for the values $n = 43923$ and $m = 27149$, for as we have seen, in this case $len_{\text{EUCLID}}(n, m) = 83$ and $len_{\text{NAIVEGCD}}(n, m) = 108599$.

I have already argued that the latter result shows that (1.13) has the property of being unknowable through any "direct" means (i.e. either by deductive derivation or a corresponding calculation which are based directly on the definition of $gcd(x, y)$). But it should be equally clear that the former result can be taken as evidence that this statement *is* knowable by the application of the "indirect" procedure EUCLID. In particular, it presumably *is* within the reach of arithmetic ability to carry out the 83 steps of EUCLID (which requires performing 15 long divisions) in order to calculate that the value $gcd(43923, 27149)$ is 17.[25] And for this reason if we wish to maintain that we do have genuine mathematical knowledge of statements like (1.13), it seems that we must be explicitly acknowledging the role of a procedure like EUCLID in its fixation.

As I have already argued above, this observation may already be taken as an indication that in accepting that we know (1.13), we are already committed to regarding algorithms as mathematical objects. This follows simply by virtue of the fact that we need to construct M_{EUCLID} in order to rigorously show that EUCLID is correct. But what remains to be seen is whether there are any features of our use of this or other procedures which further elucidate the nature of the relationship between EUCLID and M_{EUCLID}.

As we will see in subsequent chapters, there are a number of factors which complicate the discussion of this issue. Not the least of these is that given an informal specification of an algorithm A, the precise structure of the object M_A which must be constructed in order to prove that A is correct with respect to a given function f will generally not be uniquely determined. And thus although it may seem that given the pseudocode specifications of NAIVEGCD and EUCLID, M_{NAIVEGCD} and M_{EUCLID} are indeed "canonical" representations of these procedures, this is at least in part an artifact of the simplicity of these procedures

[25] If, again, the reader doubts either of the feasibility or infeasibility claims, the same argument can be recycled with different values of n and m in (1.13) so as to magnify the difference even further. For instance, computing $gcd(560171761683, 346205188258)$ via NAIVEGCD takes requires over a trillion long divisions, but only 19 via EUCLID.

and the relative precision of their specifications. Since there generally will not be as strong a connection between A and M_A, one might reasonably wonder whether they must actually be held to be identical in order for a correctness proof given for M_A to accepted as establishing that A computes f.

In order to frame this question more precisely, we may distinguish between two broad alternatives. On the one hand, one might hold that although we must construct *some* mathematical object M_A in order to prove that A is correct with respect to f, M_A must merely represent A "well enough" to allow such a proof to go through. On this alternative, the relationship between A and M_A would be something like which must hold between a physical system Ψ (such as a pendulum or spring) and its representation as a mathematical dynamical system M_Ψ in order for us to be able to reliably use the latter to predict the behavior of the former. Since Ψ is composed of physical constituents and M_Ψ of mathematical ones, certainly no one would claim that they are *identical*. Nonetheless, M_Ψ may represent Ψ with sufficient fidelity that it still useful to speak of the former as *mathematical model* of the latter.

To a first approximation, it may seem as if this sort of faithful representation is all that is required of the object M_A. For note that in order in order to prove that A is correct with respect to f, we will generally not need to assume that all of A's computational properties (whatever they may be) are accurately reflected in the structure of M_A. After M_A is defined, all that is required is that it may be employed as a constitutent in a conventional mathematical proof with conclusion $\forall \overline{x}[App_A(\overline{x}) = f(\overline{x})]$. And if this is all that is needed in order for a correctness proof to sanction the use of A in order to compute the values of f, then it seems that there is no route from the considerations adduced above to the conclusion that A itself must be regarded as identical to M_A.

There are, however, reasons to doubt that this interpretation of the relationship between A and M_A is sufficient to explain why we are justified in using A to compute the values of f. The fundamental observation in this regard is that while the mathematical role of M_A may be exhausted by its contribution to the correctness proof for A with respect to f, such a proof should not be taken to *justify* the use of A to compute the values of f if we did not also take M_A to resemble A in a variety of other respects. This point can be seen most vividly

by considering the position in which we would find ourselves were we to be presented with the specification (1.27) without having any prior knowledge of Euclid's algorithm. In such a situation it seems reasonable to suppose that although we might comprehend that (1.27) specified a mathematical procedure, neither this understanding nor whatever additional practical ability is require to carry out the specified procedure is sufficient to conclude that the procedure so specified determines a given mathematical function.

It is, of course, precisely in this kind of context where the necessity of constructing a mathematical correctness proof before applying EUCLID would be felt most strongly. But if we had no antecedent knowledge of what function EUCLID computed, then such a proof could only serve to justify the use of this procedure to compute $gcd(x, y)$ (or whatever other function we took EUCLID to compute), if we also understood the mathematical model M which it employed to reflect our intuitive understanding of the operation of the algorithm which we understood to be expressed by (1.27). For note that it is only in such a situation that such a correctness proof could have the intuitive significance of proving EUCLID (as opposed to some other algorithm) correct.

By way of example, it presumably would *not* be sufficient in such a context to take M to be M_{NAIVEGCD}. For despite the fact that we have seen that this structure can be used to prove a correctness result of the appropriate form, M_{NAIVEGCD} does a bad job at representing the operation of EUCLID as it is expressed by (1.27). This observation can be made more precise by noting a number of "intensional" features of EUCLID which would be preserved if we were to represent this procedure as M_{EUCLID} but which would not preserved if we were to represent it as M_{NAIVEGCD}. Among this are the fact that EUCLID employs the modulus function (as opposed to a divisiblity predicate), that it halts when one its parameters has been decremented to 0 (instead of incremented to $\min(n, m)$) and that it has running time $O(\log_2(\min(n, m)))$ (and not $O(\min(n, m))$). Thus although it is a possible to prove a sort of ersatz correctness result for EUCLID by taking M_{NAIVEGCD} as its mathematical representation, were we to proceed in this manner, we would then have no basis for claiming that the algorithm which we took to be expressed by (1.27) was itself correct.

These considerations appear to point to the necessity of imposing some kind of intensional adequacy condition on the mathematical representations of algorithms which are employed in correctness proofs. To reiterate, the example just considered suggests that before a correctness proof for A with respect to f can be endowed with the operational significance of allowing us to use A to compute the values of f, the object M_A which we use to represent A in the course of proving its correctness must reflect at least certain of the properties which we would attribute to A based on our prior understanding of its mode of operation. If this is indeed so, then our willingness to accept mathematical statements like (1.13) demonstrates not only that the algorithms A which we employ to derive them are mathematically representable in some way, but also that for each such procedure there must exist what we might refer to as a *faithful representative* of A – i.e. a mathematical object M_A whose structure accurately reflects our informal understanding of A's mode of operation.

This latter interpretation of what is required of a correctness proof obvious comes closer to entailing the conclusion announced above that in accepting that we know statements like (1.13) we are indirectly committed to algorithmic realism. For if what is required to accept such a proof as justifying the use of A to compute f is not only that M_A induces the same function as A but also reflects in its definition our intuitive understanding of A's mode of operation, then we are one step closer to justifying the realists proposal to identify A with M_A. However, at least three significant questions remain open: 1) are the operational constraints imposed by either the necessity of proving correctness or our other theoretical practices involving algorithms sufficient to require that M_A represent A faithfully in all respects?; 2) if so, is it possible to uniformly construct mathematical representations with the correct properties; and 3) what relationship ought we to take these representations to bear to A? I will take up each of these questions starting in Chapter 2.

Chapter 2

Algorithms as objects

2.1 Introduction

Algorithmic realism was introduced in Chapter 1 as consisting of two claims: i) that individual algorithms are mathematical objects; and ii) that the computational properties of algorithms are structural properties of the mathematical objects to which they correspond. A proponent of algorithmic realism will thus be likely to hold that the individual algorithms studied in a field like the analysis of algorithms or complexity theory are like the structured or "algrebraic" objects studied in graph theory, algebra or topology. And consequently, he is also likely to hold that computational properties like running time complexity, which are typically ascribed to algorithms directly, are structural properties of these objects in much the same way that we standardly speak of commutativity being a property of groups, connectedness being a property of graphs, or compactness a property of topological spaces.

Neither of these claims is likely to sound particularly revolutionary to readers familiar with contemporary computer science. For as noted in Chapter 1, at least in their theoretical idiom, computer scientists treat algorithms in a highly realistic manner. This is reflected in a variety of ways which I will examine over the course of this chapter. One of the most prominent is the widely employed practice of referring to individual algorithms by using expressions which have all of the hallmarks of proper names. These terms correspond to expressions like Euclid's algorithm, MERGESORT or HEAPSORT which are conventionally used as shorthand to refer to specific procedures which have been introduced by a variety of different means. As such, these terms function in much the same way that constants and other mathematical "names" are used in conventional mathematical discourse.

I will take this latter category to include not only numerals (e.g. $|||$, 2, II) but also terms denoting well-known set theoretic or combinatorial structures – e.g. ω, ϵ_0, C_6, K_5, $\mathbb{S}^1$

etc. Terms of this sort are generally introduced to our mathematical language by explicit definitions and are conventionally taken to refer to particular structured objects (i.e. sets, groups, graphs, topological spaces, etc.). As I will consider in detail below, it is possible to locate many systematic parallels between our introduction and use of these terms and procedural names. Perhaps the most apparent of these is our willingness to use such names to form what appear grammatically to be subject-predicate sentences such as "MERGESORT has running time $O(n \log_2(n))$." Such procedural statements can be compared to purely mathematical ones such as "2 is prime", "C_6 is abelian", "K_5 is connected" or "$\mathbb{S}^1$ is compact" which are also standardly treated as having the subject-predicate form $P(t)$.

In the mathematical case, our willingness to treat such statements as having this logical form has traditionally been taken to go along with a willingness to regard apparent singular terms like 2, C_6, and K_5 as denoting mathematical objects, and predicates like "is prime," "is abelian," and "is connected," as denoting mathematical properties which are true or false of these objects. In fact, the observation that numerical terms can serve as grammatical subjects is one of the observations which motivated Frege [39] to argue that natural numbers are abstract objects. Of course, many theorists have found this sort of argument, which jumps directly from observations about the surface syntax of a class of statements to ontological conclusions about their presumptive subject matter, far too quick to the punch. And in fact, when applied to simple arithmetic statements like "2 is prime" there are at least two well worked-out ways in which: 1) we can deny that such statements express propositions which are of subject-predicate form (as did Russell, who held that natural numbers were not objects, but rather classes of classes); 2) we can acknowledge that such statements express subject-predicate propositions but deny either that they are true or that their truth requires that numerical terms be taken to refer to abstract objects (as have a variety of mathematical nominalists and fictionalists, ranging from Quine and Goodman [112], to Field [34] to Yablo [157]).

Our present project is, of course, not to evaluate the prospects for algorithmic realism in general, but rather those of algorithmic realism. And to iterate, it is the algorithmic realist's central contention that algorithms are mathematical objects. Since algorithmic realists presumably take the fact that we name and quantify over algorithms as good

evidence that such entities exist, they also hold (presumably *ipso facto*) that mathematical objects exist.[1] And for this reason I will accept (for the most part naively) not only the received platonistic interpretation of classical mathematics on which, e.g., numerals denote individual natural numbers, numerical quantifiers range over a well-defined class of objects, etc. but also that one of the reasons why it ought to be accepted is on the basis of the standard sort of linguistic or (largely) Fregean argument gestured at in the previous paragraph.

The central task of the present Chapter will be to consider to what extent an analogous sort of argument can be mounted in favor of regarding algorithms as abstract objects on the basis of routine but largely unconsidered observations about how we speak of them in the course of our computational practices. As an entree into this topic, I will begin by recording the standard observation that even if we agree that the fact that terms such as 2, π or ω denote abstract objects, the mere fact that they can appear in the same sorts of linguistic contexts as names for people, cities, and cats tells us essentially nothing about which such objects they denote. And for this reason, virtually all contemporary theorists who describe themselves as mathematical realists also accept some degree of indeterminacy in the reference of mathematical expressions.

With this said, however, there is also a near universal consensus within mathematical logic that the *structure* of all (or virtually all) of the objects studied in classical mathematics can be represented as pure sets in a manner which preserves the truth values of all (or virtually all) propositions in which they figure (the exceptions concern those structures which are "too big" to be represented as sets.). Of course this discovery on its own is by

[1] Of course this argument may also be seen as too quick to the punch. For it is at least open to a theorist to simultaneously hold that algorithms are mathematical objects and also that mathematical objects either don't exist or must be identified with concrete ones via the application of some form of nominalistic reduction. However, I propose to not take this view seriously here for two reasons. First, to equate the algorithms with mathematical objects and then deny existence to this these objects runs counter to the spirit of the realist's desire to view algorithms as themselves abstract in the sense discussed in Chapter 1 – i.e. existing outside of space and time, possessing computational properties which are independent of our linguistic modes of specifying them, etc. And second, I will argue below that a large part of the motivation for adopting algorithmic realism is reconstructive in the sense that it attempts to account for our use of procedural methods within classical mathematics. As such, many of the technical challenges an algorithmic realist will face will be the same regardless of whether he takes the mathematical objects with which he seeks to identify algorithms as themselves possessing genuine existence, or merely serving as ontological way stations within a larger nomimalistic programme.

no means a cure for referential indeterminacy. For it is well known that there are many different ways of assigning sets as the denotations of numerical terms so as to preserve the truth values of mathematical statements. This difficulty is paradigmatically framed by Benacerraf's [7] famous observation that it is difficult to see how a case can be made for identifying the natural numbers with finite von Neumann ordinals as opposed to finite Zermelo ordinals. For under both means of "reducing" numbers to sets, all number theoretic statements receive the same truth value. And for this reason it is generally conceded that there can be no non-stipulative answer to the question "What set is denoted by '2'?"

Somewhat less attention has been devoted by philosophers to the use of apparent singular terms to denote mathematical objects other than natural numbers. But as I have already indicated, such symbols are routinely used in mathematical practice to denote not only real, rational and complex numbers, but also used to denote so-called "combinatorial" or "algrebraic" objects such as graphs (e.g. K_5), groups (e.g C_6), or topological spaces (e.g. $\mathbb{S}^1$). This is significant because I have already indicated in Chapter 1 that common mathematical representations of algorithms such as transition systems are structurally similar to these objects. It is, of course, well known that these structures may also be represented by sets by the use of standard "coding" techniques. For instance, rational numbers can be represented as pairs of natural numbers, and real numbers can be represented as sets of rational numbers. Similarly, graphs and groups can be represented as relations or functions over the natural numbers which can then be represented as sets of ordered pairs of numbers. But since at every step in such a procedure the variety of unconstrained encoding choices grows exponentially (e.g. are real numbers to be represented by Dedekind cuts versus Cauchy sequences; ordered pairs a la Weiner or a la Kuratowski, etc.), the degree of referential indeterminacy of expressions like π, C_6, K_5 or $\mathbb{S}^1$ is presumably at least as great as it is for 2.

This is just one of several well-known problems involved with extending standard doctrines about reference and truth to mathematical discourse. However in attempting to forge a link between discourse about structured objects like graphs and groups with discourse about algorithms, it will not matter so much whether we can take a term like C_6 or K_5 to denote a specific set or other sort of abstract object. What will matter, however, is that

standard mathematical theories such as those of analysis, graph theory or group theory are *axiomatizable*, say by a theory T over a (possibly second-order) language $\mathcal{L}$. And in such cases, there are well-known ways of systematically interpreting $\mathcal{L}$ into the language of set theory so that the terms intended to denote these structures (i.e. individual real numbers, graphs, groups, etc.) are mapped to terms denoting sets.

Given the broad success we have experienced in constructing such set theoretic reductions of this sort for a variety of mathematical disciplines, it seems natural to suppose that discourse involving algorithms could also be treated in a similar fashion. For although I argued in Chapter 1 that algorithms are not properly conceived of as being part of the *subject matter* of any traditionally recognized branch of mathematics, they presumably are part of the subject matter of several subfields of computer science – most notably the analysis of algorithms and complexity theory. These fields are generally thought of as being formal not only in the sense of applying mathematical methods, but also in the sense of yielding non-trivial theorems which admit to formal proof. However, as we will see below, the conventional statement of such results often refers to or quantifies over algorithms. And thus it seems natural to suppose that the traditional methods of set theoretic interpretation could be extended to cover these subject matters in much the same way we assume that they could be applied to a novel branch of mathematics.

As things currently stand, however, no truly systematic attempt has been made to carry through a programme which shows how the discourse of theoretical computer science can be uniformly interpreted in set theory or any other axiomatically presented mathematical theory. This is problematic in two respects. First, it leaves us without a precise compositional analysis of what it means for a statement like "MERGESORT has running time $O(n \log(n))$" to be true – i.e. a means of semantic interpretation which tells us what object is denoted by MERGESORT and what property is denoted by "has running time $O(n \log(n))$." This is especially significant because such statements are often taken to express non-obvious theorems of algorithmic analysis which are standardly proven by non-trivial arguments. One would thus expect it to be possible to say in precise terms not only what it means for this argument to be valid, but also explain why it should be taken to prove whatever proposition is expressed by "MERGESORT has running time $O(n \log(n))$." But as we will

see below, terms like MERGESORT are not (at least in any straightforward way) taken to abbreviate descriptions of particular mathematical structures. And thus there is no means of using an "off the shelf" set theoretic reduction in order to associate MERGESORT with a mathematical object.

It should be clear how the algorithmic realist hopes to remedy this situation. In particular, in holding that algorithms are mathematical objects, he maintains that there is some way of carrying out the reduction just described, relative to which MERGESORT can be taken to denote a particular set. Relative to such a reduction, this would clearly be a boon for theoretical computer science in at least two respects. For on the one hand, it would show that statements like "MERGESORT has running time $O(n \log(n))$" can themselves be taken to express mathematical theorems. And on the other, it would show that this can be achieved without having to interpret them via a potentially elaborate paraphrase which treated MERGESORT in some manner other than as a singular term.

Having re-issued these advertisements for algorithmic realism, my perspective on this view remains that of a skeptic. The basis of my skepticism lies in the suspicion that it is inappropriate to view individual algorithms as mathematical objects not because they can be shown to lack one or more of the properties traditionally associated with such entities (e.g. lack of spatial or temporal location, causal inertness, etc.) but rather I fear that they cannot legitimately be taken to correspond to objects in the first place. Having said this, it must be acknowledged that questions about the general notions of objecthood (e.g. "What is an object?", "What objects are there?", "What does it mean for an object to 'fall under' a property?", etc.) are among the most vexed in analytic philosophy. And as such, one might reasonably wonder what it even means to deny that algorithms are objects.

I hope to illustrate over the course of this chapter that the source of my concern does not arise from the details of any particular doctrine about ontology or objecthood. Rather, it originates with the otherwise banal observation that it turns out to be impossible to state identity conditions for algorithms which track the way we speak about them in the course of our theoretical practices. An analogous observation can, of course, be made about concrete items like sand dunes, ocean waves, and ice cubes, which we may want to conceive of as "entities" but for which it seems likely that no such criterion can be given (cf., e.g.,

Lowe [77]). This claim is also sometimes generalized to apply to everyday items like tables and chairs (cf., e.g., Unger [147]) and to persons (cf., e.g. Unger [146]).

However, things stand quite differently with respect to mathematics. Items like natural numbers or groups are distinct from items like sand dunes and chairs not only in being abstract, but also in the sorts of terms we use to refer to them and the characteristic modes of inference into which these terms enter. In general, once we have fixed a language $\mathfrak{L}$ for speaking about the members of a mathematical domain $\mathcal{D}$, not only are all statements of the form $t_1 = t_2$ expected to be meaningful and potentially informative for all $\mathfrak{L}$-terms t_1, t_2, but, above all, they are expected to all have truth values. In standard case, these values can be taken to be fixed either axiomatically relative to an $\mathfrak{L}$-theory T, or by reference to a standard model M of $\mathfrak{L}$. In such cases, T (or T together with M) can thus be taken to supply identity conditions for the Ds. And in a variety of other familiar cases, the definition of $\mathcal{D}$ itself will come along with a definition $=_{\mathcal{D}}$ of identity for this class (e.g., in the case of sets, $=_{\mathcal{D}}$ is the relation of co-extensivity, in the case of finite groups, $=_{\mathcal{D}}$ is group isomorphism, in the case of topological spaces $=_{\mathcal{D}}$ is homeomorphism, etc.). Thus while it may at least be coherent to claim that some everyday "entities" lack identity conditions, the thought of a class of mathematical objects for which no such criterion can be given is almost inconceivable.

In claiming that algorithms correspond to a class of mathematical objects $\mathcal{M}$, the onus thus lies squarely with the algorithmic realist to provide a criteria of identity for these objects as identified with members of $\mathcal{M}$. In particular, although the realist may have some latitude as to how $\mathcal{M}$ itself is defined, once defined, it is not open to him to assign denotations to procedural terms like "Euclid's algorithm" and MERGESORT in an arbitrary or stipulative manner. Rather, he must do so in a manner which respects our preexisting conventions and results for regarding the procedures denoted by terms t_1 and t_2 as the same or different. Thus whatever formal criterion of identity the realist might attempt to give over $\mathcal{M}$ must accord with certain fixed data about algorithmic sameness and difference.

I have already acknowledged that it will be my ultimate contention that algorithms fail to be mathematical objects precisely because such a criterion cannot be formulated. However this amounts to a claim with the following logical form: for any class of mathematical

objects $\mathcal{M}$ and means of uniformly associating algorithms with the members of $\mathcal{M}$, there does not exist a equivalence definable relation $=_{\mathcal{M}}$ on $\mathcal{M}$ such that for all algorithms A_1, A_2, we are willing to accept that $A_1 = A_2$ if and only if the associated objects $M_1, M_2 \in \mathcal{M}$ are such that $M_1 =_{\mathcal{M}} M_2$. Note that this statement has negative existential form – i.e. it states that given $\mathcal{M}$, no definition of equality over $\mathcal{M}$ can be given which has the appropriate properties. And since it is up to the realist himself to define both $\mathcal{M}$ and $=_{\mathcal{M}}$, one might reasonably expect that such a claim would be difficult to demonstrate, essentially because one would have to show that no such relation exists uniformly in the definition of $\mathcal{M}$.

In arguing against algorithmic realism, it will thus be useful to start out by delimiting the choices of $\mathcal{M}$ and $=_{\mathcal{M}}$ as sharply as possible. The purpose of this chapter is to initiate this task by reviewing a variety of conceptual considerations which point to constraints on how these items must be defined so as to meet certain general adequacy conditions on the sort of general theory of algorithms which a realist will need to construct. I will proceed in two phases. First, in Section 2, I will consider various ways in which we refer and reason about algorithms in the course of both our theoretical and everyday practices. These considerations both serve as the fundamental data for which the realist hopes to account. And also, as we will see, they impose a variety of constraints on how the domain of algorithms must be characterized by help us locate notion of algorithm in the context of traditional discussion about language and ontological commitment.

In Section 3, I then will attempt to parlay the conclusions of Section 2 into a more focused discuss of how the algorithm realist might go about the task of reducing algorithmic discourse to mathematical discourse. In this context, it possible to note a number of systematic affinities between the status which the algorithmic realist wishes to assign to algorithms and that which a mathematical nominalist wishes to assign to mathematical entities. Most prominently, the algorithmic realist can be understood as claiming either that algorithms "just are" mathematical objects in the sense that discourse about algorithms is just discourse about mathematical objects "deep down" or that at least such discourse can be systematically reconstructed in terms of mathematical discourse. These correspond to rhetorical positions which Burgess and Rosen [16] label *hermenutic* and *reconstructive* nominalism which a mathematical nominalist might adopt about the status of mathematical

objects relative to the non-mathematical domain to which he claims they can be reduced.

Appellation notwithstanding, I will suggest below that the view I have been calling algorithmic realism can thus in fact be understood as a form of nominalism in the following sense. If we wish to understand what it means for statements appearing in our discourse about algorithms to be true, it might at first glance appear that we need to posit a novel domain $\mathcal{A}$ of free-standing abstract objects to which terms like MERGESORT refer. The algorithmic realist, on the other hand, claims that such statement can be interpreted in a manner such that all terms of this sort are assigned denotations in a class $\mathcal{M}$ of mathematical objects. Such a theorist can thus be understood as hoping to show that we can interpret procedural discourse in a manner that does not force us to enlarge our ontological commitments beyond those already entailed by our practices involving quantification over $\mathcal{M}$.

If we adopt this understanding of the realist's aims, it is also possible to usefully employ the framework of potential strategies for nominalistic reduction developed by Burgess and Rosen. The bulk of Section 3 will be taken up by my argument to the effect that one of these strategies is better suited to dealing with the conceptual constraints about algorithms in Section 2 than the other. In particular, I will first suggest that in order to understand the role of algorithms in mathematical practice, it is not sufficient to simply seek to reduce discourse about algorithms to discourse about mathematical objects in the familiar sense of interpreting one theory within another (this corresponds to the alternative which Burgess and Rosen refer to as *Tarskian reduction*). Rather I will argue that the realist must in fact show that algorithms need to be correlated with mathematical objects in some manner which explains how they can be employed in mathematical practice in the sense developed in Chapter 1.4.

In Section 4, I will propose instead that the algorithmic realist will be best off pursuing the strategy which Burgess and Rosen dub *contextual reduction*. Relative to this strategy, the algorithms in $\mathcal{A}$ are not held to be identified with members of $\mathcal{M}$ themselves. Rather members of this latter class are to be understood as precise mathematical representations of algorithms (or *implementations* as I will suggest they be called in Section 2) which allow us to refer to particular algorithms. I will suggest in Section 2, that such reference is

canonically effected by the use of the functional expression "the algorithm implemented by M" where $M \in \mathcal{M}$. According to this proposal, the class $\mathcal{A}$ may be understood as corresponding to the range of this function and the identity conditions of its members can be taken to be fixed in terms of an equivalence relation defined over $\mathcal{M}$ directly. I will argue that such a so-called *two-level criterion of identity* not only provides the algorithmic realist with a means of avoiding the indeterminacy and artefactuality problems discussed in the prior paragraph, but it is also most in keeping with the conceptual constraints adduced in Section 2. As such, I will conclude that what I will refer to as the *abstractionist* proposal for identifying algorithms with mathematical objects is the realist's best hope for vindicating his view.

2.2 Procedures, language and ontology

2.2.1 The linguistic view versus the metaphysical view

Before we can get very far in determining whether procedures are properly regarded as objects, we must first come to some a basic understanding of what it means to say of something that it is an object. This is a famously slippery and contentious issue, as witnessed by the fact that it is already difficult to even ask of a certain thing whether "it" is an object without prejudicing an affirmative answer by referring to it in such a manner. In recent times, the question "what is an object?" has also become tightly wrapped up with the related question "what objects are there?" This may be a methodological inevitability. For note that a given criterion of objecthood will obviously have as a consequence that a given class of items $\mathcal{D}$ are genuine objects and as such serve to answer the latter question. But our acceptance or rejection of this criterion will be at least in part based on whether the members of $\mathcal{D}$ satisfy our background intuitions about objecthood.

This situation has a particular significance with respect to the ontological status of algorithms. For as I have remarked above, it appears that conventional wisdom reflects no clear consensus on whether not just mathematical procedures (i.e. algorithms), but also everyday procedures like recipes or voting methods are properly regarded as objects as opposed to say, functions or properties. And for this reason, it does not appear that its

pronouncement either for or against the view that procedures are objects can be taken as part of the data relative to which a theory of objecthood should be assessed. In seeking to answer the question "Are procedures objects?" we must thus look to a theory motivated by independent conceptual criteria. However, my goal in this in this section will not be reach ultimate clarity on what form such a theory should take. To the contrary, all I wish to show is that relative to traditional views about objecthood and ontology there is a reasonably strong basis for regarding procedures as objects.

The traditional means of answering the question "what is an object?" is decidedly anti-metaphysical in its motivation. It suggests that in order to determine whether something is an object, we should start out by examining the language $\mathcal{L}$ which we use to talk about it. According to the most straightforward version of this view, we are committed to regarding as objects all entities which are the denotations of possible terms of $\mathcal{L}$. This proposal is customarily traced to Frege's argument in the [39] that since numerical expressions such as "the number of planets" or "17" function as terms when they appear as the grammatical subjects of what appear to be subject-predicate sentences such as "The number of planters is odd" or "17 is a prime" these terms must be taken to refer to objects in order to explain how these sentences can be true.

If we assume that we are given $\mathcal{L}$ as a formal first-order language, this argument can be explained in somewhat more detail as follows. In this setting, the class of closed $\mathcal{L}$-terms $CT_{\mathcal{L}}$ is standardly defined to be the smallest set containing i) all of the constants $c_1, c_2, \ldots$ (i.e. "names") of $\mathcal{L}$, ii) all expressions of the form $f(t_1, \ldots, t_n)$ whenever $t_1, \ldots, t_n \in CT_{\mathcal{L}}$ and f is an n-ary function symbol of $\mathcal{L}$ and iii) (assuming that definite descriptions are in the language) all expressions of the form $\iota x.\varphi(x)$ where ι is the Russellian definite description operator and $\varphi(x)$ a $\mathcal{L}$-formula containing only x free. Now consider a particular $CT_{\mathcal{L}}$ term t such that there exists some $\mathcal{L}$-atomic predicate $P(x)$ such that the sentence $P(t)$ is *true*.[2]

[2]The requirement that $P(t)$ be true is needed since it is not the intention of the Fregean argument to license the conclusion that all closed terms denote objects. For instance, even though we can imagine that $Winged(Pegasus)$ and $Rational(1/0)$ might be grammatically well-formed atomic sentences of $\mathcal{L}$, Frege's argument is not intended to license the conclusion that either $Pegasus$ or $1/0$ denotes an object. He achieved this in [38] by stipulating that sentences containing non-denoting terms lack truth value. But it could also obviously be achieved by insisting either that constants are denoting or by adopting a variant of free logic in which statements of the form $P(t)$ are regarded as false if $E(t)$ is false.

It is then claimed that in order for this sentence to be true, it must be possible to assign a reference to t so that it falls under the reference assigned to P. And according to the standard understanding of truth and predication as canonized in the Tarskian semantics of first-order logic, this means that the denotation of t must be an element of the domain over which $\mathfrak{L}$ is interpreted which is contained in the extension of the predicate P – i.e. an object of which the property of P holds.[3]

A related linguistic criterion of objecthood is customarily traced to Quine's [111] well known argument that ontological commitment is carried by existential quantification. In order to apply this view to a particular statement S in natural language which we regard as true (e.g. "there are cats" or "there are numbers") we must first regiment S' using semi-formal canonical notation which represents its logical form by, among other things, making quantification explicit. S' may then be translated into a first order sentence φ. And finally, the so-called *existential commitments* of S may be read off as the values which the quantifiers in φ must range over for it to be true.

The central feature of both the Fregean and Quinean proposals is that they propose to answer the question of what constitutes an object entirely on the basis of examining the language in which we standardly converse or theorize. Following Lowe [77], I will refer to proposals of this sort as instances of the *linguistic view* of objecthood. The first check on plausibility for proposals of this sort is to compare the class of items they entail must be objects against those nominated by intuition. And in this regard, it is well known that in

[3]The Fregean argument is often presented in a manner so that it applies in the first instance to natural languages as opposed to formal ones. In this context, it is often summarized by saying that it is the so-called *singular terms* of a language which potentially denote objects and thereby carry ontological commitment. However, this formulation presents an obvious problem for the linguistic view since it presupposes the availability of a purely grammatical means of characterizing the singular terms of a language. Although this presents no problem for formalized languages for which we possess a inductive definition of term-hood, the availability of much a criterion for natural language is a much more contentious issue. Dummett [29] seems to have been the first to realize that if we blind ourselves entirely to meaning, and consider only grammatical structure, defining singular terms as expressions which can appear as the subjects of subject-predicate sentences fails to exclude quantifiers (as in "Someone is Iraqi") or indefinite descriptions (as in "A dog is a man's best friend"). As a remedy, he proposed further inferential criteria which he hoped would rule out these expressions. These criteria have subsequently been modified and extended by Hale [55] and [155]. Although it is a still unsettled whether such analyses succeed in providing sufficient conditions for singular term-hood, I will suggest below that the class of expressions by which we can make reference to procedures is surprising limited. Thus despite the fact that we typically reason about procedures in natural language rather than in first-order logic, it would be straightforward to verify that all the terms in this restricted class pass even the most sophisticated tests which have been proposed to identify singular terms.

adhering to either version of the linguistic view we run the risk of embracing overly lavish domains of objects. This is perhaps most evident in the case of the Fregean variant as evidenced by the apparent acceptability of statements such as the following:

(2.1) a) *The whereabouts of Osama* are currently unknown.

b) *The identity of the UNABOMBER* was disclosed by the FBI.

c) *The status of Padilla* was disputed by Bush.

d) *The sake of liberty* was the stated motivation for the war in Iraq.

e) *The situation in Iraq* continues to decline.

f) *The fact that Clinton was impeached* was instrumental in Bush's victory.

The italicized expressions all serve as the grammatical subject of each of these sentences. According to both the Fregean criterion and the more refined inferential criteria mentioned in note 3, all of these expressions should be taken to denote objects. And it thus follows on this basis that a proponent of the Fregean criterion will be obliged to accept that these terms denote objects.

The obvious problem with this conclusion is that it flies in the face of our everyday intuitions. For note that to the extent that we do possess a native, unitary conception of what objects are, we would tend to cite macroscopic concrete items with definite boundaries like tables and chairs as its prototypical instances. We are often willing to generalize substantially from such paradigm cases in extending this status to very small items (e.g. electrons, quarks), very large ones (e.g. countries, planets), ones lacking definite boundaries (forests, oil spills), ones recognized by only social convention (e.g. the Supreme Court, the IBM corporation) and various classes of abstracta (e.g. letters, words, sentences, numbers, sets, etc.) which are typically held to lack spatial or temporal location. But it should be clear that in various ways and to varying degrees, entities of these sorts are all tightly integrated into our normal mode of describing and reasoning about the world. This is perhaps most basically reflected in the fact that in most cases where we agree that a terms denotes an object, we are able to state what sort of object it is – e.g. "17" denotes a number, "France" denotes a country, "IBM" denotes a corporation, etc. And it is also reflected in the fact that we can typically cite a large variety of characteristic features involving the

members of most of these categories – for instance the denotation of "17" is odd, is less than 18, etc. and the denotation of "France" is a country in Europe, is bordered by Germany, etc.

But despite the fact that the sentences in (2.1) demonstrate that terms like "the whereabout of Osama" and "the sake of liberty" have the same grammatical distribution as that of terms denoting table, chairs and numbers, we are generally not willing to countenance the proposal that such entities as whereabouts or sakes correspond to objects. This is reflected in the fact that not only do we appear to be at a loss as to what kind of object expressions like "the whereabouts of Osama" or "the sake of liberty" denote, it is also unclear what it means for such items to stand in various relation to one another. This is true despite the fact that we occasionally say things like "The sake of liberty played a greater role in Bush's decision to invade than the sake of human rights" or "The whereabouts of the UNABOMBER remained unknown for longer than the whereabouts of Saddam." Note that in such cases it appears doubtful that we have any definite conception of what relation must hold between pairs of sakes or whereabouts in order for these sentences to be true. In fact, the further we go in adopting such manners of speech, the greater our tendency becomes to think that we must explain what we mean in terms of paraphrases that do not involve reference to items like sakes or whereabouts.

Since a proponent of the linguistic view might thus take it as an embarrassment that his view has a consequence that these so-called "Fregean monsters" must be accepted as objects, we might additionally ask whether the Quinean strategy faired better in ruling them out. The first problem we encounter in attempting to implement this strategy is that although we can suppose that there is no problem in recognizing the language $\mathfrak{L}$ over which sentences like those in (2.1) are stated, it is more difficult to say with any certainty that they are part of a particular informally stated theory $\mathcal{T}$. As a consequence, it is difficult to know how to regiment such sentences so as reveal their logical form, let alone uniformly translate them into a first-order theory T' formulated over a language $\mathfrak{L}'$ which might employ a more impoverished class of predicates than $\mathfrak{L}$ (i.e. a leaner "ideology" in Quine's sense).

For note that on the one hand it appears plausible to think that in some instances reference to problematic entities like sakes and identities can be eliminated by regimenting

statements like those in (2.1) in a manner which quantified over less problematic classes of entities. For instance we might attempt to regiment (2.1a) in $\mathfrak{L}'$ as $\exists x[Location(x) \wedge Knows(Bush, InLocation(x, Osama))]$. But since it is difficult to construe such statements as being part of a systematic theory the truth value of whose theorems have to preserved under the proposed mode of paraphrase, it is not always clear when we should accept such a formalization as a legitimate representation of the content of the original sentence.

To make matters worse, there are other apparently acceptable $\mathfrak{L}$ sentences which quantify over the problematic entities explicitly – – e.g.

(2.2) a) There is some fact which Cheney knows which Bush does not.

b) There is some situation in Iraq which would prompt the United States to withdraw.

If anything, it seems even harder to know what (if any) principles constrain how sentences like these should be regimented and formalized compared to those appearing in (2.1). And for this reason, it is even less promising that we can seek to rule out the conclusion that facts and situations must be accepted as objects without appealing to extra-linguistic intuitions about how such statements may be allowably paraphrased or otherwise analyzed in order to eliminate the apparent need to quantify over such items.

I will refer to a strategy which accepts that such intuitions must be exploited in order to answer the question "what is an object?" as a *metaphysical view* of objecthood. One way in which the metaphysical view differs from the linguistic one is that it does not adopt a unitary approach to answering either this question or the related ontological questions about objects we are committed to accepting by speaking in a certain manner. For instance, according to the metaphysical view, if we want to know whether sakes are objects, we must look beyond the grammatical distribution of statements which purport to speak about such entities and examine our more general intuitions about what we take such statements to mean. I will argue in Section 2.3 that our practices for referring to procedures make it almost certain that proponents of the linguistic view embrace the conclusion that algorithms are objects. But if we want to know if this conclusion follows only because procedures are caught up in an explosion of ontological commitments which proponents of this are obliged

to embrace, we attend in more detail to particular features of how we refer to and reason about algorithms.This is how I will proceed in section 2.4.

But before starting out it will also be useful to consider a miniature example of the debate which might go between proponents of the linguistic and metaphysical views with respect to a fixed class of terms. To borrow an example from Lowe [77], consider the class of expressions Δ which can be used to denote facial expressions (e.g. grins, smiles, frowns etc.) in a natural language like English. It is easy enough to come up with a variety of sentences paralleling the form of those in (2.1) and (b) – e.g.

(2.3) a) The grin on Bush's face is broad.

b) Bush is wearing a broad grin.

c) The smirk on Bush's face belied his anger.

d) The reporter's question caused a frown to cross Bush's face.

These examples illustrate that it is possible to construct sentences which treat terms in Δ as grammatical subjects. And if we follow Quine in holding that when regimenting a statement for purposes of assessing ontological commitment, definite descriptions should be eliminated in favor of their Russellian analyses, these statements will also be regimented in a manner which employs quantification over facial expressions. This means that proponents of both the Fregean and Quinean versions of the linguistic view will be forced to concede that facial expressions such as grins are indeed objects.

But note that whatever grins are, they lack many of the features possessed by canonical examples of objecthood. For instance, while there is a sense in which a grin is located in space and time, it is difficult to say exactly what its boundaries in either dimension are. And while it thus might be said that grins are concrete entities, it is at least complicated to explain the relationship they bear to the material media (in facial muscles, skin, bones, etc.) in which they are embodied. As a consequence of the first observation, we appear to be without clear cut intuitions to guide us in answering questions such as whether a given person can wear the same grin on different occasions, whether a grin-like expression punctuated by a grimace should be regarded as one or two grins, whether two people can both wear the same grin, how grins are different than smiles, etc. And as a consequence of

the second observation, there also seems to be a sense in which we are unable to say what a grin might be other than that it is a kind of facial expression – i.e. is a grin a bounded volume of matter? a state or other form of temporal cross section of such a volume? a temporally extended material process? These considerations mirror many of the intuitive reasons we may wish to balk at treating sakes, whereabout, etc. as objects. On this basis, a partisan of the metaphysical view could reasonably balk at the conclusion that grins ought to be regarded as objects.

At this juncture, the proponent of the linguistic view must decide whether to bite the bullet and admit the existence of cases in which his characterization of objecthood differs from that embedded in everyday intuitions or attempts to sophisticate his proposal in some manner which can accommodate our misgivings about grins. One strategy he might attempt is to argue that sentences such as those in (2.3) can be paraphrased so as to avoid the use of grin-denoting expressions as grammatical subjects. For instance, without apparent change in meaning, both (2.3a) and b) can be rendered as "Bush is grinning broadly." If it could be shown that such paraphrases were available and could be constructed uniformly for all sentences containing terms in Δ, then one might plausibly suggest that we can interpret everyday discourse about facial expressions without trafficking in ontology which regarded individual grins, smiles, frowns, etc. as objects.

One problem with applying this strategy is that paraphrasability is generally a symmetric relation in the sense that if we can replace one sentence φ by another φ' with apparent preservation of meaning, than the converse will also generally be true. This means that in the absence of additional constraints, it may be impossible to tell whether φ or φ' more accurately reflects the ontological commitments of someone who holds one or the other to be true. This means that unless we defer to extra-linguistic (i.e. semantic or metaphysical) considerations specific to facial expressions, there appears to be no principled way to explain why we should take the sentence "Bush is grinning broadly" to be more ontologically transparent than the sentence "The grin on Bush's face was broad." But it is just these sort of consideration which a proponent of the linguistic view seeks to avoid in attempting to reduce questions of ontology to those of grammar.

Another way in which the linguistic view proponent might seek to rein in the ontological

explosion which takes in grins, along with sakes, whereabouts, etc. in its wake is to argue that the expressions such as those in Δ are not singular terms after all. One way in which this might be accomplished is to argue that all "genuine" singular terms must be introduced to the language of which they are a part along with conditions which determine when all sentences of the form $t_1 = t_2$ are true for all $t_1, t_2 \in \Delta$. In other words, this view proposes that it is part and parcel of regarding an expression as a singular term that we are able to state conditions which govern the truth value of identity statements in which it appears.

Such a move is generally made in conjunction with the recognition that it only makes sense to speak of an expression t as denoting an object if there is a so-called *sortal concept* F under which the denotation of t falls. Such a concept is generally defined as one which applies to entities which we can count. By way of example, terms like "man", "tree" and "book" all are paradigmatic terms denoting sortal concepts because it makes sense to ask "how many?" with respect to the men in the room, the trees in the yard or the books (qua physical volumes) on the shelf. But note that in order to count or enumerate items so as to answer such a question, we must be able to individuate between different men, trees or books in such a manner that it is clear what distinguishes one from another. Sortal terms are thus typically contrasted with terms like "red", "loud" or "salty" which express concepts G which can be applied to groups of objects but for which we appear to no means of responding to question like "how many Gs?" (e.g. how many salty things) absent additional contextual qualification about the sort of items we are speaking. Since the ability to count Fs appears to requires that there be a stable means of individuating items falling under this concept, it seems reasonable to suppose that our justification for regarding F as a sortal is at least partly constituted by our faith that identity conditions are available which allow us to definitively decide when two items are or are not the same F.[4]

[4]The use of the term "sortal" to refer to an expression of the former sort has a somewhat complicated history. The modern usage can be traced back to Aristotle and Locke who used it to refer (roughly) to classes of objects which had essences. Frege [39] is now commonly exposited (e.g. by Wright [154]) as having proposed that the concept *natural number* is a sortal in the sense that he proposed both that numbers were objects and their status as such was to be explained in terms of a criterion for statements of the form "the number of Fs." The introduction of the term "sortal" to describe a concept whose members can be counted is generally credited to Stawson [138].

We can now see that there is a potential problem with treating terms in Δ as denoting objects because although these terms putatively denote members of the concept *facial expression*, there appears to be no criterion of identity which accompanies either this concept or any of the subordinate concepts *grin*, *smile*, *frown*, etc. For instance, our understanding of what it is to be a grin does not systematically determine under what conditions (if any) we ought to accept sentences such as

(2.4) a) Bush wore the same grin throughout the press conference.

b) The grin that Bush was wearing at the inauguration was the same as the grin he was wearing during his State of the Union address.

c) The grin on Bush's face was the same as that worn by Rove.

Our inability to answer such questions seems to be wrapped up with our lack of firm intuitions about the properties of grins mentioned above.

These observations suggest that we are unable to cite proper identity conditions for the concept *grin* but also presumably for its superordinate concept *facial expression* more generally. If a proponent of the linguistic view adopts the proposed amendment to the definition of singular term, it thus seems as if we might be avoid to conclusion that facial expressions ought to be counted as objects in virtue of the facts that grins, smirks, smiles, etc. do not possess well-defined identity conditions. But note that the sort of considerations on which this conclusion is based again appear to rely on extra-linguistic considerations about the relevant concepts. For since sentences like those in (2.4) are certainly grammatical, we must go outside of language in order see that we do not know relevant identity conditions for grins that would enable us to answer them. But identity conditions are precisely metaphysical principles telling us what identity consists in for objects falling under a given concept.

Considerations of this sort can obviously be used to put the proponent of the linguistic view back on the defensive. But I will not attempt to draw any general conclusions here either about whether linguistic considerations alone suffice to plausibly delimit the class of entities which can justifiably be regarded as objects. For as we will see over the next subsection, there is indeed a strong linguistic case for regarding procedural expressions as

denoting objects. But at the same time, there is also a strong case both for regarding the notion of algorithm as falling under the superordinate concept *procedure* and also for regarding the latter concept as what is referred to as an *abstract sortal* – i.e. a sortal concept whose identity conditions are given in terms of a class of other abstract objects (in this case, that of implementations). We will ultimately be in a position to see that rather than standing in conflict, both the linguistic and metaphysical views strongly support the thesis that algorithms ought to be regarded as objects. And therefore, there is no present reason while we should try to decide between them.

2.2.2 Referring to procedures: names and quantification

In this section, I will examine our practices for making reference to algorithm in the technical discourse of computer science. The major finding will be that linguistic considerations of the sort reviewed in the prior section point strongly to the conclusion that proponents of the linguistic view will endorse the claim that algorithms are objects based on the characteristics of this discourse alone. This is already a significant observation. For as I have already noted, conventional wisdom seems to reflect no view as to whether either algorithm or procedures more generally ought to be regarded as more like tables, people, and numbers, which we are comfortable regarding as objects or more like sakes, whereabouts or grins, to which we are disinclined to extend this status.

Based on our discussion, however, tradition reflects that any fundamental examination of the ontological status of algorithms ought to begin with an examination of the class of expressions whereby we appear to make reference to such entities. And in this regard, there are a number of striking regularities which become apparent almost immediately when we begin to look systematically at how we speak about procedures in both our everyday and technical discourse. The first of these is that it appears to be possible to divide all terms referring to individual procedures into three types, two of which are present in everyday speech and one mainly reserved for technical discourse in computer science.

The most familiar way we have of referring to procedures is via expressions which bear all the linguistic hallmarks of proper names. Terms of this sort are used widely in everyday

discourse in mathematics and computer science. Familiar examples of such terms in mathematics include "Newton's method" in analysis, "The Runge-Kutta method" in differential equations, "Gauss-Jordan elimination" in linear algrebra, "The Henkin construction" in model theory and "The Gentzen cut elimination algorithm" in proof theory. In computer science the practice of naming individual procedures is taken to an even greater length. A cursory examination of any standard text on algorithms will reveal a raft of apparent proper names for procedures: "Bubble sort", "Insertion sort," "Selection sort," "Merge sort," "Heap sort,", "Radix sort", "Breadth first search", "Depth first search", "Euclid's Algorithm", "Prim's algorithm", "Dijkstra's algorithm", "Strassen's algorithm", etc.

Although little explicit attention is paid to the use of these expressions in the conduct of computer science, I have previously called attention to the fact that the use of these terms has become so conventionalized in fields like the analysis of algorithms that they are often written in a special font (e.g. EUCLID, MERGESORT, etc.). The use of such a stylistic device encourages the comparison of these expressions with special symbols such as Π or e which are used to denote numerical constants or K_5, D_3 or ω which denote fixed structures or sets. Terms of these sort are generally are introduced by explicit mathematical definitions. And consequently one might think that our use of procedural terms like EUCLID or MERGESORT could be explained in a similar manner. But a number of considerations point to the fact that our use of procedural names cannot be understood in such simplistic terms.

I examine the issue of how procedural names are introduced in detail in the next subsection.[5]First, however, it will be useful to note several features of our usage of such expressions which reinforce the fact that they are indeed treated as singular terms in the course of theoretical practices. I have already noted the most fundamental of these in Section 1 – i.e. that expressions like MERGESORT often appear in subject-predicate statements. This is witnessed by the fact that statements such as the following might all readily be found in a textbook on algorithmic analysis:

(2.5) a) EUCLID computes the greatest common divisor function.

b) HEAPSORT has running time $O(n \log(n))$.

c) INSERTIONSORT is a comparison sort.

d) MERGESORT is recursive.

e) Prim's algorithm uses a priority queue.

f) Kruskal's algorithm is a greedy algorithm.

The use of such procedural names is in fact pervasive in computer science, that it is common to come across passages where such terms are use uniformly to describe, compare and contrast the merits of various algorithms. Here are two example drawn from popular textbooks:

> In this chapter we'll study the sorting algorithm which is probably more widely used than any other, Quicksort ... Quicksort is popular because it is not difficult to implement, it's a good "general-purpose" sort (it works well in a variety of situations), and it consumes fewer resources than any other sorting method in many situations ... The desirable features of the Quicksort algorithm are that it is in-place (uses only a small auxiliary stack), requires only $n \log_2(n)$ operation on the average to sort n items, and has an extremely short inner loop. The drawbacks of the algorithm are that it is recursive ..., it takes about n^2 operations in the worst case, and it is fragile: a simple mistake in the implementation can go unnoticed and can cause it to perform badly ....[125], p. 115

> In this chapter, we introduce another sorting algorithm [HEAPSORT]. Like merge sort, but unlike insertion sort, HEAPSORT's running time is $O(n \log(n))$. Like insertion sort, but unlike merge sort, HEAPSORT sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, HEAPSORT combines the better attributes of [INSERTIONSORT and MERGESORT which] we have already discussed.[24], p. 140

These passages leave little doubt that in practice we are willing to use procedural names to form subject predicate sentences. Upon mild regimentation, these examples have the form $P(a)$ with a corresponding to terms like EUCLID, HEAPSORT, etc. and $P(x)$ taking on the values "x computes the greatest common divisor function", "x has running time $O(\log(\min(n, m)))$," etc. Similarly, the passages just cited also suggest that we ought to be willing to form relational sentences of the form $R(a, b)$ for comparing two algorithms,

[5] One relevant observation is that use of procedural names extends outside of the purely mathematical domain as witnessed by our tendency to refer to various sorts of everyday procedures by names as well. This is witnessed, for instance, by the fact that many industrial and chemical procedures have been given names such as the Le Blanc Process (for producing potash) and the Solvay Process (for producing sodium bicarbonate). Similar comments apply to the use of expressions such as "Borda count", "instant run-off voting", "approval voting", "Vickrey auction" and "Dutch auction" in social choice theory and economics.

stating, for instance, that a has faster running time than b.[6]

Another equally notable facet of our linguistic practices involving reference to procedures is our willingness to quantify over them.[7] Statements involving explicit quantification over procedures are often used to express technical results, some typical examples of which are as follows:

(2.6) a) There exists an algorithm for determining whether two non-deterministic push down automata accept the same language.

b) There exists a polynomial time algorithm for deciding whether $n \in \mathbb{N}$ is prime.

c) There exists a matrix multiplication algorithm with running time less than $O(n^{2.8})$.

d) There is no algorithm for determining where a polynomial with integer coefficients has a root.

e) If $P \neq NP$, then there is no polynomial time algorithm for determining whether a propositional formula is satisfiable.

f) There is no comparison sorting algorithm with running time complexity less that $O(n \log(n))$.

It is a notable fact that all of the statements in (2.6) are operationally non-trivial in the sense that they report results to problems which had either been open for a long time when they were solved or at least have reasonably involved proofs. (2.6a), for instance, reports the solution to a problem in automata theory which had been open for over forty years before it was answered positively by Senizergues [126] in 2003. (2.6b) reports a similar positive solution to a long standing problem in computational number theory solved by Argawal et al. [1]. And finally (2.6c) reports a watershed result of Strassen [137], which

[6] Although in talking about everyday procedures we are somewhat less systematic in identifying properties which apply to procedures directly, there are clear cut analogues to these statements which can occur in everyday discourse. For instance, we do not hesitate to say things like "the Solvay procedure produces sodium carbonate," "instant run-off voting is resistant to strategic voting" or "the Vickrey auction uses sealed bids" all of which appear to employ procedural names as grammatical subjects.

[7] This too is witnessed with respect to our discourse involving everyday procedures. Note the apparent acceptability of statements such as "There does not exist a chemical procedure for turning lead into gold", "There are six recipes for mayonnaise in *The Joy of Cooking*", and "All methods for producing electricity yield heat as a byproduct."

was entirely unexpected at the time of its publication since it was thought that it was impossible to multiply matrices in time faster than that achieved by the obvious $O(n^3)$ algorithm. Since practitioners of theoretical computer science would regard each of (2.6a-f) as expressing a significant technical result, it seems reasonable to regard such statements as being embedded within the context of a well-developed scientific theory.

This is significant from the standpoint of our ontological investigations because it means that when we go about assessing the commitments of (2.6), we ought not to regard these sentences as appearing in isolation. Rather, these statements should be seen as part of a larger theory to which our proposed mode of regimentation and formalization ought to be applicable uniformly. I will discuss what this theory might look like as a whole in more depth in Section 4. But for the time being, we may simply note that the surface structures of these sentences clearly express explicit quantification over algorithms. This is to say that (2.6a-c) would all naturally be regimented as existential statements of the form $\exists X \Phi(X)$ which express that an algorithm with a certain computational property Φ exists. And thus if this is indeed the appropriate means of rendering them into first-order logic, then proponents of the Quinean version of the linguistic view will conclude that not only should we regard algorithms as objects, but also that in accepting (2.6a-c) as true we are committed to the existence of at least the three algorithms which are required to witness their existential quantifiers.

With this said, however, cognoscenti of mathematical logic and theoretical computer science are also likely to protest that it is at best unclear whether (2.6a-c) commit us not to the existence of *algorithms* as opposed to some other sort of mathematical objects. In particular, although it may be most faithful to our informal understanding of these statements to treat them as quantifying over algorithms, they may also be paraphrased in a manner which replaces quantification over algorithms with quantification over various formal models of computation. This is most straightforwardly true for the case of (2.6a) which can be taken to have the basic form $\exists X[\forall x[Exec(X, x) = 1 \leftrightarrow x \in S]]$ where $Exec(X, x)$ gives the value of executing the algoritm correspond to X on input x and S corresponds to the set of encodings of ordered pairs $\langle M_1, M_2 \rangle$ of pushdown automata accepting the same language. We would standardly say that the algorithm claimed to exist

is thus a *decision procedure* for this set and the statement (2.6a) states that the set S is *decidable*.

Note, however, that decidability is usually understood extensionally in the sense that what we usually mean in saying that a set S is decidable is that there exists *some* effective procedure for deciding membership in S. By using Church's Thesis, it is thus possible to replace the quantifier in "There exists an algorithm for deciding membership in S" with "There exists a Turing machine for deciding membership in S" without apparent shift in mathematical meaning. (This is so because Church's Thesis tells us that in quantifying over all functions computed by Turing machines we are, *ipso facto*, quantifying over all functions computed by (effective) algorithms.) On this basis, it may be argued that a statement like (2.6a) may be paraphrased by a statement of the form $\exists T \forall x[T(x) = 1 \leftrightarrow x \in S]]$ where here T varies across Turing machines and not algorithms. But not only does this latter statement fail to quantify algorithms, if our interest in (2.6a) is exhausted by the proposition that S is decidable, then it follows that this statement also succeeds in expressing its mathematical significance. And thus if this or a similar mode of paraphrase can be applied to our discourse about algorithms uniformly, then there can be no direct argument from our belief that (2.6) is true to the either of the conclusions that algorithms either exist or that they are properly regarded as objects.

There are, however, substantial reasons to think that there is no uniform mode of paraphrase which is applicable to (2.6b,c). For note that these statements would be regimented as having the form $\exists X[K(X) \wedge \ldots]$ where K is a predicate (such as having polynomial running time or running time less than $O(n^{2.8})$) which in practice we apply directly to procedure-denoting terms. If we are to paraphrase these statements in a manner so that the derived statements express the same mathematical propositions as (2.6b,c), we must thus do so in a very particular manner. Specifically, we must replace the quantifiers appearing in (2.6b,c) so that the objects appearing in their intended domains $\mathcal{M}$ are of the appropriate sort such that predicates like K can be applied directly to terms denoting objects in $\mathcal{M}$. This may arguably be achieved for (2.6a) where $\mathcal{M}$ can be taken to be the class of polynomially bounded Turing machines. But as will again be appreciated by cognoscenti, it is considerably harder to know what class can be taken to serve this function in case of

statements like (2.6c) where our interest resides in the existence of an algorithm falling within one of the levels of Knuth's [72] asymptotic time hierarchy.

Before completing my survey of the linguistic evidence for treating procedures as objects, it will be useful to also reflect briefly on the significance of statements (2.6d,e,f). Knowledgeable readers will again realize that these statements each express solutions to non-trivial technical problems problems.[8] But whereas (2.6a,b,c) express the existence of a procedure with certain properties, these latter statements express the nonexistence of one. As such, they are most naturally rendered in the form $\neg\exists X \ldots$ which are, of course, equivalent to universal statements of the form $\forall X \ldots$.

Since these statements do not have existential form, our acceptance of them does not in and of itself commit us to the existence of any particular algorithms. But taken in conjunction with the provisional conclusion that our acceptance of statements like (2.6a,b,c) *does* have such commitments, statements (2.6d,e,f) take on a greater significance. For note that once we know that the domain of mathematical procedures form a non-empty class $\mathcal{A}$, the fact that there can be precise mathematical results about all members of this class (or an appropriately delimited subclass) suggests that $\mathcal{A}$ itself must be a well defined totality over which we can determinately quantify. For note that it would not be possible to prove a theorem all about *all* algorithms (providing that there are any) which failed to possess a certain property (e.g. that of deciding whether a polynomial with integer coefficient has a root) if the extension of $\mathcal{A}$ were not itself well defined. While we are not yet in a position to give an explicit definition of this class, the fact that limitative results of this kind are both quite common in theoretical computer science and also that we accept them as genuine theorems suggests that it ought to be possible to provide one.[9]

---

[8](2.6d) expresses the negative solution to Hilbert's 10th problem (i.e. does there exist an effective algorithm for determining whether a polynomial over $\mathbb{Z}$ has a root) jointly due to Matiyasevich, Davis, Robinson and Putnam. (2.6d) states the famous result of Cook [21] that the satisfiablity problem for proportional logic is NP-complete and is thus as difficult to solve as the halting problem for non-deterministic polynomial bounded Turing machines. (2.6e) expresses the well-known lower bound theorem for the complexity of comparison sorting which was one of the first uses of the important *decision tree* method in the analysis of algorithm developed by Johnson and Ford [35].

[9]Here again cognoscenti will realize that some of these statements can be paraphrased in a manner which appears to eliminate direct quantification over procedures while retaining their intuitive mathematical significance. This is most straightforwardly true of (2.6d) which would standardly be interpreted as being equivalent to a statement of the form "the problem of deciding membership in X is undecidable" which in turn can be taken to be equivalent to "there does not exist a Turing machine for deciding membership in

2.2.3 Algorithmic reference via description

From the perspective of a proponent of the linguistic view the, evidenced adduced in the prior preceeding section presents a fairly strong case for the dual theses that algorithms ought to be regarded as objects and that at certain objects of this sort exist. I will thus henceforth assume that if a proponent of the metaphysical view wishes to take exception to this conclusion, the onus lies on him to adduce additional considerations about the concept of algorithm which demonstrates why these conclusions should not be accepted. Ultimately, of course, I believe that such considerations can be advanced. But before we are in a position to do so, a number of additional observations about our practices involving algorithms have to be put in place.

To this end, I wish to consider further our conventions for using procedural names of the sort described in the previous section. In particular, given that our practices reflect that we can use terms like MERGESORT, EUCLID, etc. to refer to algorithms, it is open to a proponent of the metaphysical view to seek an explanation of how, and ultimately to what such expressions refer. I have already observed that our use of these expressions appears to have much in common with the use of mathematical terms like e, Π, C_6, ω. And as I have also noted, it is generally possible to trace back our use of these terms to an explicit definition whereby they are introduced to our mathematical language. In particular, it requires only a slight regimentation of standard textbooks to view terms of this sort as being introduced by definitions of the form $t = \iota x.\varphi(x)$ such as the following:

(2.7) i) $e =_{df}$ the y such that $\int_1^y \frac{1}{x}dx = 1$

ii) $\omega =_{df}$ the least limit ordinal

iii) $C_6 =_{df}$ the cyclic group of order 6

In such case, the expression provided on the left-hand side of the definition serve can, at least to a first approximation, be taken as a mere abbreviation for the expression on the

---

X" as above. Since in this case $\mathcal{A}$ could be taken to consist of all Turing machines, this example does not itself suggest that there is any great mystery involved with accounting for apparent quantification over all procedures. As with the corresponding existential statements (2.6b,c), the problem consists in determining the appropriate domain for the quantifiers in (2.6e,f) so that the properties of having polynomial running time and being a comparison sort may be directly predicated of its members.

right-hand side. In particular, these statements allow us eliminate the symbol appearing on their left-hand side for that appearing on their right hand in a manner which will preserve the truth value of the expressions in which they appear. And note that such a substitution must be often made in order to prove new statements involving the defined symbol (for instance to prove the statement "e is transcendental," we must prove "the y such that $\int_1^y 1/x$ is transcendental"). However the introduction of such symbols is standardly thought to have considerable heuristic significance since, among other things, the allow what would otherwise be long and unwieldy statements (e.g. "the group G that there exists $g \in G$ such that for all $h \in G$, $h = g^n$ for $n \leq 6$ is abelian") with succinct ones (e.g. "C_6 is abelian").

This analysis obviously fails to account for many interesting properties of mathematical definitions.[10] But in the case where a term a term t has been introduced by an explicit definition it does at least allow us to give an account of the *reference* of t relative to our account of the reference of the expression in terms of which it has been introduced. In particular, if t is defined as $\iota x.\varphi(x)$, then the mathematical object to which t refers will be precisely the item satisfying this definite description.

I have already noted that terms like MERGESORT and EUCLID have a similar grammatical distribution to terms like e and C_6. And if we examine a standard textbook on algorithmic analysis, it is possible to the usage of such expression back to particular contexts in which these terms are introduced to our computational language. But at the same time, it is not all clear whether we can understand these terms as having been introduced by explicit definitions in the same way the way as mathematical names. In particular, while a typical algorithmic name *a is* standardly introduced via the use of some other linguistic expression t (which may be a term in either a mathematical or natural language), t is normally neither a definite description or functional expression. And as such, the reference of A cannot be determined merely by determining the reference of t.

One of the major theses I wish to advance in this chapter is that although the methods by which we are able to make (apparent) reference to algorithms are complex, they are also

[10]For instance it fails to take into account that a single symbol c can often have more than one definition given by t_1 and t_2 (e.g. e is often defined as $\lim_{n\to\infty}(1+1/n)^n$) and that identities of the form $t_1 = t_2$ can often be informative. And it also fails to provide any direct insight into why we elect to introduce novel symbols to abbreviate some definite descriptions but not others.

sufficiently conventionalized that they may be profitably studied in order to gain better appreciation for the ontological status of algorithms. Reflection on our practices reveals that the linguistic mechanism by which terms such as a are introduced generally takes one of two forms which may be schematized as follows:

(2.8) a) $a =_{df}$ the procedure expressed by Π

b) $a =_{df}$ the procedure implemented by M

In scheme (2.8a), Π is meant to represent what an instance of what I will refer to as a *program* – i.e. a linguistic description of a procedure given in either natural language, pseudocode or a formal programming language. In scheme (2.8b), M is meant to what I will to as an *implementation* – i.e. a specification of a member of a formally defined model of computation (i.e. a Turing machine, RAM machine, logic circuit, etc.).

The thesis which I wish to advance is that if algorithms are indeed objects, our ability to refer to them must be understood as being mediated by these schemas. From this it would follow, for instance, that to say that in order to refer to the algorithm MERGESORT we must first make reference to either a particular program which expresses it or a particular implementation which (as I will say) implements it. But programs are merely sequences of linguistic expression types and implementations are (generally) finite combinatorial objects akin to graphs. Given that I assume that reference to mathematical objects in general is unproblematic for the purposes of this chapter, I will assume that there is little mystery about how we can refer to programs or implementations. For according to the claim I advance, our ability to refer to algorithms must be explained in terms of our ability to refer to members of these other classes of objects.

The task of defending the claim that reference to procedures must be mediated by applications of one of (2.8a,b) divides into two components: 1) that it is indeed possible to refer to algorithms by the use of these schemas; and 2) that such reference cannot be effected by any other means. If 1) and 2) can be established, then I will argue below that a number of consequences follow about the general notion of algorithm which may be of potential use to the proponent of algorithmic realism. But as thing stand, we face a number of challenges in demonstrating these claims. With regard to 1), for instance, we

face the initial task of demarcating appropriate notions of *program* and *implementation* so that (2.8a,b) may plausibly be taken to approximate the manner in which procedures are introduced in practice. On the basis of these definitions, something additional will then have to said about how, on the basis of fixing one of these objects, we are then able to refer to individual algorithms. And with regard to 2), we must at ask after the possibility of referring to algorithms by conventional means such as ostension.

In regard to the first problem, it is easiest to regard notions of "program" and "implementation" as terms of art in computer science. This is to say that although these notions both admit to loosely confederated classes of technical definitions, it is difficult to give a precise formal definition which applies to all formalisms which, in practice, are referred as programs or implementations. In practice, these notions are thus most commonly introduced by classes of simiarly structured programs or implementations known respectively as *programming languages* and *models of computation*. These definitions do arguably draw on a core set of intuitions about the theoretical purpose which individual programs and implementations play in our computational practices. And as I now wish to illustrate, not only are these roles quite different, but they also provide very different impressions on how we are to refer and reason about algorithms.

Algorithms and programs

The interplay between formal and pre-theoretical computational notions is most readily appreciated with respect to the concept of program. From a formal perspective, a program is simply a sequence of expressions who individual syntax and modes of composition are determined by a formal grammar known as a *program language*. Programs are thus intrinsically linguistic entities. There are, however, a variety of such languages, which are themselves clustered around different notions of what it means to interpret or carry out a program. For present purposes, two main classes will be significant: *imperative* programming languages such as `C`, `Pascal` and `FORTRAN` and *functional* programming languages such as `LISP`, `Scheme` and `Haskell`.[11] Since interpretation of imperative languages is more

[11]The other major class (or so-called *programming paradigm*) is that of *declarative* languages such as `PROLOG`. Declarative programs resemble statements in a fragment of first-order logic and as such do not have

straightforward, it will be useful to start out there.

A declarative programming language L can be taken to be given by a definition of a class of primitive statements $\alpha_1, \ldots, \alpha_n$ whose intended interpretation is to perform a given action. A program over L is then defined recursively as a the smallest of expressions containing each of the α_i and closed under certain formation rules for forming complex programs. A simple example of such a language is the programming language `While` whose only primitive statements are of the form $x := t$. Such statements are known as *assignments* and have the intended intepretation that upon execution, the value associated with the variable x will be the value of t. Here t is generally taken to be a term over a first-order language $Sig(L)$ which I will refer to as the *signature* of L and for present purposes can be taken to be $\{0, 1, +, -, \times\}$. The programs over `While` is then by the grammar $\Pi = (x := t) \mid \Pi_1; \Pi_2 \mid$ $\mathsf{if}\ \varphi\ \mathsf{then}\ \Pi \mid$ $\mathsf{while}\ \varphi\ \mathsf{do}\ \Pi\ \mathsf{od}$ where φ is quantifier free formula over $Sig(L)$. The symbols ;, and $\mathsf{if} \cdot \mathsf{then}$, $\mathsf{while} \cdot \mathsf{do} \cdot$ denote what are known as *control structures*. They function grammatically as connectives between programs. Semantically they respectively serve to sequentially conjoin programs Π_1 and Π_2, condition the execution of Π on the basis of truth value of φ and to cause Π to be successively executed until φ becomes true.

A simple example of a `While` program is as follows:

(2.9)
```
q := 0;
r := x;
while y <= r do
  r := r - y;
  q := q + 1;
od.
```

In contexts such as programming manuals or textbooks, it would be typical to see a name attached to this program such as `Divide(x,y)`.[12] It would also be common for an informal

as natural a procedural interpretation as do imperative or functional programs. Since they are rarely used in stating algorithms directly, I will thus not mention such programs further.

[12] In more complex languages like `C` or `Pascal`, which allow for the definition of subroutines, we might be formally required to give this program a name so that it could be called by other blocks of code. Although

interpretation of what this program does to be expressed using this name, along the lines of the following: "`Divide(x,y)` computes the quotient of the values stored as the variables `x` and `y`."

In addition to introducing names to denote programs, we use programs to denote procedures in a more abstract sense. This is evident in the apparent acceptability of statements such as

(2.10) The Euclidean division algorithm (DIVIDE) is the algorithm expressed by (2.9).

Although it is as yet unclear what it means for a program to "express" an algorithm, (2.10) at least exemplifies how schema (2.8a) is employed in practice. Sometimes such a statement would be issued when the name of the algorithm in question is already in our language. In such a case, a use of a statement like (2.10) would be one in which we *reidentified* an algorithm which we had already picked out in some other way (say as "the division algorithm described in Book VII of *Elements*).

More often than not, however, names for algorithms are introduced directly by formal programs like (2.9) or by informal procedural specifications of the sort I referred to in Chapter 1 as *psuedocode* programs. For instance, the Euclidean division algorithm might also be introduced as follows:

(2.11) In order to determine the integer quotient of n and m do as follows:

Step 1: Let $q = 0$ and $r = n$

Step 2: While $r \geq q$, repeatedly perform the following two steps:

Step 2a: Subtract m from r and let the result be the new value of r;

Step 2b: Increment q by 1.

Step 3: Return q and halt.

Pseudocode programs such as this one contain a mixture of prose, mathematical symbolism and programming language constructs such as line numbers, block structure, looping constructs, etc. In the practice of a field such as the analysis of algorithms or complexity theory, it is generally specifications of this kind which are used to introduce algorithms.

---

the term `Divide(x,y)` in such cases would receive a precise meaning relative to a semantics for the language in question, it is still customary to use this name informally to denote the procedure expressed by (2.9).

This is to say that in addition to the (2.9), the algorithm DIVIDE could also be introduced as "the algorithm expressed by (2.11)."

In the context of our general desire to better understand what algorithms are in an ontological sense, this is a very significant observation. For if we can refer to DIVIDE as the algorithm expressed by either (2.9) or (2.11), then it at least seems plausible to think that this algorithm corresponds to whatever object corresponds to the denotation of the functional expression "the algorithm expressed by Π" where Π is taken to be either of these specifications. To think about algorithms in this manner is, of course, to employ the well-known analogy introduced in Chapter 1 according to which the relation in which an algorithm stands to a program is the same as that in which a proposition stands to a sentence.

This analogy would be of little use in helping us to understand how algorithmic reference is possible if we do not also assume that some clear notion of a proposition as a mind- and language-independent abstract entity is available and it is these entities which sentences express. But many attempts to work out such a theory of propositions have been proposed.[13]Although theories of these sorts differ substantially in detail, what is relevant for present purposes is that they may all be seen as determining a mapping *prop* from the class $Sent_L$ of declarative sentences of a natural language K into a class of abstract structured entities $\mathcal{P}$. Such a mapping is generally supposed to have two features: i) $prop(S)$ is determined from S compositionally in terms of its grammatical constituent structure; ii) for each natural language sentence S, $prop(S)$ is taken to be the proposition expressed by S; iii) as such, the formal properties of $prop(S)$ such as its truth conditions or the inferential relationship which it stands to other propositions ought to reflect those which we intuitively assign to S. There is, of course, substantial disagreement among contemporary theorists about how each of these requirements should be made precise.[14]But theories of the sort just described at least provide a means of speaking about propositions as the sorts of things which are expressed by sentences in natural language.

---

[14]The tradition reflects that Frege [37] and Russell [115] are the originators of a view of propositions. Modern theories of structured propositions have been put forth by Soames [134], Salmon [117], Bealer [6] and Cresswell [25].

[14]E.g. about what it means for the value of *prop* to be determined compositionally from S, whether *prop*

If we take the algorithm/proposition analogy seriously, one way in which we might hope to understand how reference to algorithms is possible is by providing a formal semantic theory which interprets procedural specifications such as (2.9) and (2.11). According to this view, what is required to make sense of a statement such as (2.10) is that we develop something akin to a semantic theory which can be applied to (2.9) or (2.11). Such a theory could be described as a mapping *prog* for the class of well-formed programs $Prog_L$ over a programming language L into a domain $\mathcal{A}$ of abstract objects such that for each program Π, the value of $prog(\Pi)$ was the algorithm which it expressed. Presumably such a theory would also be subject to same sort of constraints as a semantic theory for natural language – i.e. i) the value of $prog(\Pi)$ should be determined compositionally in terms of the structure of Π; and ii) the object $prog(\Pi)$ must reflect at least certain aspects of our informal interpretation of Π.

One substantial obstacle the we face in attempting to work out the details of such a proposal is that a pseudocode program like (2.11) is neither completely in a natural language nor completely in a formal programming language. This is problematic, for, as noted above, such specifications correspond to the primary means of introducing algorithms in the computer science literature. However, it is also generally acknowledged that such specifications are not an entirely precise means of introducing an algorithm, precisely because our understanding of many of the constructs they employ is informal. As such, there is also a consensus among practitioners that any sufficiently "high level" programming language L provide sufficiently expressive resources that any pseudocode program can be replaced with a formal program over L which express the same algorithm.[15] And if this is indeed the case, then we proceed with the current strategy for understanding algorithmic reference by constructing a formal semantic theory of L.

---

must take contextual parameters as arguments in addition to S itself and about which informal characterized aspects of the meaning of S must be preserved by $prop(S)$.

[15] This view reflects the widely subscribed ideology that programming languages are media for expressing algorithms and that modern high level languages like `C`, `C++` and `Java` are expressively complete – i.e. that for each informal algorithm A they contain at least one program Π which expresses A. It must be kept in mind, however, that since the underlying notion of algorithm in question is informal, this view must be understood as an informal thesis rather than a precisely articulated technical hypotheses. Note, however, that if it turns out to be false, this presumably only makes life hard for the algorithmic realist who wishes to understand algorithmic reference by (2.8a).

Having now settled on the project of proving a formal semantic theory for formal programs as the most promising means of accounting for reference to algorithms via (2.8a), a number of problems remain. For note that even if we also agree that such a theory should take the form of a mapping $prog : Prog_L \rightarrow \mathcal{A}$, we must also determine which program language or languages should serve as our linguistic basis for stating algorithms. And more fundamentally yet, we must also provide some background characterization appropriate to these language of what it means to say that a particular form of abstract object $prog(\Pi)$ serves as the semantic interpretation of Π. For note that whereas in the case of a natural language sentence S, we know at least some of the properties which $prop(S)$ ought to be possess in order to serve as a plausible candidate of the proposition expressed by S – e.g. $prop(S)$ must be truth evaluable, must have the same truth conditions as S, must stand in the same entail relations as S, etc. But in the case of a program it is not immediately clear what the relevant notion of meaning is which must be preserved between Π and $prog(\Pi)$.

Both of these questions represent prima facie problems for a theorist who wishes to explain algorithmic reference via (2.8a). For on the one hand, we have already noted that there are a great variety of distinct programming languages, some of which are inspired by quite different computational paradigms. And as such, not only is there no clear background notion of program meaning, it may also be that what intuitions we have about this notion vary from one programming language to another. But since I will ultimately argue that the algorithmic realist will be better off looking at (2.8b) as the fundamental device by which we are able to refer to individual algorithm, it will not be a constitute a great disservice to his cause if we answer these question partially by fiat.

For purposes of elucidating the consequences of adopting (2.8a) as a means of referring to algorithms, it will, for instance, suffice to confine our attention to the language `While`. And for this reason, it will also suffice to assume that the domain of *prog* is the class $Prog_{\texttt{While}}$ of well-formed `While` programs. One useful consequence of this stipulation is that since this language has been studied extensively within the subfield of theoretical computer science known as *programming language semantics*, it is possible to experiment with a number of different alternatives for how *prog* ought to be formulated. For note that within this subject there are at two means of approaching the general conceptual problem

of assigning a formal interpretation as the "meaning" of a program. These correspond to the paradigms known as *denotational* and *operational semantics*.[16]

The difference between the goals of denotational and operational semantics for `While` can most easily be understood in terms of the sort of object which is assigned as the value of $prog(\Pi)$ for $\Pi \in Prog_{\texttt{While}}$. In the case of denotational semantics, $prog(\Pi)$ is taken to be the (extensional) function $f_\Pi : \mathbb{N}^n \to \mathbb{N}$ which Π would compute were it to be carried out in the manner which I will discuss further below. And in the case of denotation semantics, $prog(\Pi)$ is taken to be an abstract description of a computational mechanism by which this function is carried out. As I will explore further below, this is commonly achieved by associating Π with a so-called *transition system* M_Π which, when executed, induces the function f_Π.

In the practice of computer science, both forms of the semantics can be said to "give the meaning" of Π in different contexts. This leads us naturally to ask whether either f_Π or M_Π can be taken to correspond to the reference of the expression "the algorithm expressed by Π" as it appears in a statement such as (2.10). The first alternative can be easily ruled out, however, on the basis of the fact that `While`, like all other general purpose programming languages contains programs Π_1 and Π_2 such that $f_{\Pi_1} = f_{\Pi_2}$, despite the Π_1 and Π_2 express intuitively distinct procedures. For instance, let $\Pi_1 =$ `Divide(x,y)` and let Π_2 be as follows:

(2.12)
```
a := 0;
q := 0;
while a < x do
```

[16] These correspond to two of the four major branches of programming language semantics, the other two being axiomatic and *algebraic semantics*. Like denotational and operational semantics, these approaches can seen as attempting to formalize different notions of "meaning" which might can be applied to programs. Roughly speaking, for instance, an axiomatic semantics for a programming language can be seen as a means of specifying the effect of carrying out a single instruction Π in the execution of a program in terms of logical descriptions (known as *pre-* and *post- descriptions*) of the states which obtained before and after Π is carried out. Algebraic semantics, on the other hand, is an attempt to understand programs and data structure by using algebraic axioms to characterize their structural properties. But rather than make conflicting explanatory claims in the way that different form of semantics for natural language often appear to, different forms of programming language semantics are often used concert with one another. For instance, a programs are generally proved correct with respect to either a denotation or axiomatic semantics, where as they are analyzed structurally by employing an operational or algebraic semantic. It is not surprising that a number of general results can be proven about the relationship between the different objects which are the outcome of the different forms of analyses. For a uniform presentation, c.f., e.g., [100], [88] or [48].

```
    q := q + 1;
    a := a + y;
  od;
```

It can be shown that $f_{\Pi_1}(x, y) = f_{\Pi_2}(x, y) = \lfloor x/y \rfloor$ with respect to a denotation semantics for the `While` language. But if we attempt to interpret (2.12) informally, it should be obviously that this program expresses a different procedure than (2.9). For instance, we would say that (2.9) describes a procedure that computes the quotient of the values of `x` and `y` by "counting down" from the value of `x` while (2.12) describes a procedure which "counts up" up to this value.

This sort of example is not isolated in the sense that for an arbitrary general purpose programming language L, it will generally be possible to construct infinitely many intuitively distinct programs $\Pi_1, \Pi_2 \in Prog_L$ such that $f_{\Pi_1} = f_{\Pi_2}$. As such, a semantics for L which takes $prog(\Pi) = f_\Pi$ can be likened to a semantic theory *prop* for English which takes $prop(S)$ to be either a truth value or a set of possible worlds. In particular, theories of both types have the consequence of associating syntactic objects which we intuitively think of having different "meanings" (e.g. Π_1 and Π_2 in the case of `While` and "2 + 2 = 4" and "There are infinitely many primes.") with the same abstract semantic representation. Arguments have been made (e.g. by Stalnaker [136]) that such a coarse-grained assignment of semantic values is indeed the appropriate way of associating sentences with propositions so as to explain, e.g., propositional attitude attributions.

There are also theoretical contexts in computer science where a denotational form of semantic interpretation is useful. For instance, if our primary desire is to prove that Π is correct with respect to a mathematical function g which has been defined independently of Π in the sense illustrated in Section 1.4, we will interested precisely in the function f_Π. However, it ought to be equally clear that such an assignment of interpretations to programs will not serve the theoretical purpose which the realist wishes an explanation of algorithmic reference via (2.8a) to fulfill. For it is precisely the fact that (2.9) and (2.12) express *different* algorithms (as would, for instance, the examples EUCLID and NAIVEGCD from Chapter 1, were they to be appropriately translated into `While` programs) for which the realist wishes to account. And as such, no theory which assimilates the objects which

are taken as "the algorithm expressed by Π_1" and "the algorithm expressed by Π_2" can appropriately serve this function.

This leads us naturally to inquire whether adopting an operational semantics for `While` will lead to better results. However, in this case we face the substantial challenge, that even after we have fixed a language L in which we are interested, there will in general be a wide variety of different choices of operational semantics. For note that in practice of computer science, operational semantics are often provided so as to prove the correctness not of a program itself, but rather of its interpretation relative to a particular model of computation relative to which we wish to implement the programs over L. Such models will correspond to different choices of the class $\mathcal{M}_L$ which will serve as the range of what I will refer to as an *interpretation function* $op_L : Prog_L \to \mathcal{M}_L$ of L.

In many cases of interest, however, $\mathcal{M}_L$ will not directly support the mathematical operations corresponding to the interpretations of the terms in $Sig(L)$. And in such cases, op_L will often be defined as the composite function $op_L(\Pi) = op'_L(\tau_{L,L'}(\Pi))$ where L' is an intermediate level language into which L' is translated (or *compiled*) before being interpreted operationally. One sense in which the term "operational semantics" is used in practice is to refer to the mapping $\tau_{L,L'}$ which renders members of $Prog_L$ into a form where they can be associated with machines in a more directly or transparent manner. In practice, semantics of this latter are by far the most common way in which programs in high-level languages such as `While` are interpreted. But by itself, the compilation function $\tau_{L,L'}$ can no more be taken to give the meanings of the programs in L than can a method of translating one natural language into another. For note that not only would a mode of interpreting the programs of L' still require a formal semantics for this language, but operational semantics for common "low level" languages (e.g. `MIPS` or `MIX` assembly) are renowned for their intuitive opacity.

What is of more theoretical interest are theories of operational semantics which interpret the program of L directly so as to eventuate in a class of models $\mathcal{M}_L$ which support mathematical operations which are in approximate correspondence with those denoted by the symbols in $Sig(L)$. In such cases, the values of $op_L(\Pi)$ are generally abstract machines whose structure is determined inductively according to the structure of Π. This sort of

semantic theory for a programming language L_1 may thus be roughly compared with a semantic theory for a natural language L_2 which seeks to correlate a declarative sentence S with a structured proposition $prop(S)$ such that the compositional structure of the latter is determined according to the grammatical or logical structure of the former. For note that in both cases, the abstract object which is assigned as the semantic value $Prop(S)$ and $op_L(\Pi)$ is capable of distinguishing between co-denoting linguistic objects (i.e. sentences with the same truth value, or programs determining the same function) by associating them with an abstract object which is derived according to their compositional structure.

There are, however, a number of theoretical obstacles which prevent us from directly employing an operational semantics for L to directly interpret expression of the form "the algorithm expressed by Π." For note that we cannot judge whether $op_L(\Pi)$ is plausible candidate for the denotation of "the algorithm expressed by Π" unless we have some background intuitions about the meanings of programs over L. The gravity of this concern is partially obscured by the fact that both the lexicon and syntax of simple programming languages l are designed to mimic that of natural languages. This is true of many imperative languages like `While` whose programs resemble strings of imperative statements conjoined by control structure. But it is less true of more complex imperative languages such as `C` or `C++` which allow for the use of subroutines, pointers and recursion calls, none of which have clear analogues in natural language. And it is even less true still of functional languages like `Scheme` or `Haskell` whose programs are more aptly liken to complex functional terms than to natural language sentences in either the declarative or imperative moods.

Yet at the same time it is fairly common for practitioners to employ expressions like "the algorithm expressed by Π" and think they have successfully referred to an algorithm regardless of the language L from which Π is drawn. If we assume that such expressions actually do succeed in making determinate reference to algorithms, we must additionally ask whether it is possible to determine the reference of this expression in some non-stipulative manner based on an operational semantics for L. Although I think this question must ultimately be answered in the negative, it is already gaining some insight into the difficulties involved by studying the paradigmatic case where L is a language like `While`. For since this language is at least outwardly similar to the idiom we use in specifying pseudocode

programs like (2.11), a case can at least be made that certain forms of operational semantics do a better job at formalizing our intuitions about the meanings of `While` programs than others.

In order to see this concretely it will be useful to examine a particular form of operational semantics which can be given for `While` in more detail. In particular, consider the so-called *natural semantics* for `While` as given in [100]. The basic idea motivating this form of semantics is that the interpretation of an individual program Π is to be understood relative to the effect its execution would have on the so-called *computational state* of an abstract computational device which was carrying it out. Such a state can be represented as a finite vector of abstract locations or *registers* of the form $\sigma = \langle l_1, \ldots, l_n \rangle$, each of which may be used to store a integer value when n is the highest index of any variable appearing in Π (assuming that they are numbers $x_1, x_2, \ldots$). I will use Σ_Π to denote the class of all such states of a program Π.

The interpretation that the natural semantics assigns to arbitrary `While` programs Π can be understood in terms of the transformation it induces on such states. As mentioned above, the most primitive statement in this language is that of an assignment expressed by $x_i := t$ where x_i is a variable corresponding to the ith location and t is a term over $Sig(L)$. The intended effect of carrying out such a statement is that the value stored in location l_i will be updated to contain value the value denoted by t. If we adopt the notation $\sigma[l_i \mapsto a]$ to denote the state just like σ except that the location of l_i has been updated with the value a, then this fact is standardly codified by a so-called *state transformation axiom* of the form

$$\langle x := t, \sigma \rangle \Longrightarrow \sigma[l_i \mapsto [\![t]\!]] \tag{2.13}$$

Here $[\![t]\!]$ denotes the denotation of t relative to a particular interpretation of $Sig(L)$. $\Longrightarrow$ is the so-called *transition relation* and is a subset of $(\Sigma_\Pi \times Prog_{\texttt{While}}) \times \Sigma_\Pi$. The intended interpretation of $\langle \Pi, \sigma \rangle \Longrightarrow \sigma'$ is that in state σ, Π can be executed so as to yield state σ'. By way of example, we have $\langle \texttt{z := x}, \langle 1, 2, 3 \rangle \rangle \Longrightarrow \langle 1, 2, 1 \rangle$.

It is largely because of the intuitive interpretation which can be given to the state transition rule that the natural semantics for `While` may be taken to provide a connection

between our informal understanding of programs over this language and the pseudocode specifications which they might be claimed to formalize. For note that while specifications like (2.11) are structurally similar to formal programs, their individual statements correspond to English statement like "Increment q by 1" or "Let $r = n$" which are in the imperative mood. The semantic interpretation of these statements has been studied extensively in formal semantics and pragmatics, where it is generally agreed that a semantic analysis of an imperative utterance should be given not in terms of its truth value, but rather in terms of the effects of carrying it out.

One well-known theory advanced by Searle [123] holds that an imperative utterance S functions as a command or directive on the part of one party (the speaker s) to get another party (the addressee a) to perform some action α. But although this is a plausible analysis of the so-called illocutionary force of an imperative utterance, Searle [124] also holds that this notion itself must be further analyzed in terms of a speaker's intentions in producing the utterance. For instance if s utters "Shut the door," it is only in virtue of an appropriate set of beliefs and incentives on the part of s and a that this command will be carried out. It is, however, also possible to abstract away from this aspect of the role imperatives play in ordinary language and to attempt to analyze the semantic contribution of an imperative of the form "Do α" solely in terms of the effect its proper execution would have on the current state of affairs. On this analysis, for instance, the meaning of "Shut the door" can understood as a function mapping an arbitrary physical world state σ into σ' identical to σ save for the fact that the door in question is shut.

Since this analysis abstracts away from the intentions of individual speakers and addressees, it is possible to apply it to cases where an imperative statement is not actually uttered, but merely appears in the context of a set of written instructions such as a recipe or a sequence of driving directions. While such sequences of instructions clearly are part of natural language, they are also much pseudocode programs in both their structure and intended significance. In particular, both everyday procedures and pseudocode programs are formed of sequences of imperative statements composed according to simple controls which are intended to mediate the order in they would be carried out, were the procedure in question to be executed. As such, it is also reasonable to view an occurrence of the a

statement S of the form "Do α" in such a context as a sort of standing command to an arbitrary agent to perform the action denoted α were the set of instructions in which it was embedded to be carried out. For note that when we reason about the effects of carrying out such a sequence, we explicitly fail to assume anything about the individual intentions of an agent a who might carrying and rather simply assume that a carried out each step correctly in accordance with the control structure in which it is embedded.

In such a context, it is therefore reasonable to attempt to formalize the meaning of S purely in terms of the effect the execution of the action denoted α would have were it to be carried out. Note that this can be achieved by generalizing the form of our formal state transition axiom to take into account the effect of performing arbitrary operations α on a physical state σ. In particular, if we agree that natural language imperatives ought to be analyzed in terms of the function their successful execution induces on states, then a general analysis can be framed precisely by adopting an axioms of the form $\langle \sigma, \alpha \rangle \Longrightarrow \sigma'$ under the interpretation that σ' is the state derived from σ by applying α and holding everything else constant.

It is on the basis of this sort of affinity between the effect of assignment in the language `While` and the analysis of natural language imperatives that we are able to build a partial bridge between our informal understanding of a pseudocode specification like (2.11) and its putative formalization as (2.9). For note that former is specified precisely as a sequence of imperative statements conjoined by the three forms of control structures mentioned above – i.e. *composition statements* of the form $\Pi_1; \Pi_2$; *conditional statements* of the form `if` φ `then` Π_1; and *iterative statements* of the form `while` φ `do` Π. The intended interpretation of these statements may be expressed in natural language respectively as follows: 1) "do Π_1 and then do Π_2"; 2) "if φ holds in the current state then do Π"; and 3) "while φ holds in the current state, repeatedly do Π."

Of these, the most significant with respect to how the natural semantics for `While` codify intuitions deriving from our understanding of pseudocode specifications is the operation of composition. For consider a complex statement in either `While` or natural language of the form $\Pi \equiv \Pi_1; \Pi_2$ (e.g. {`z:=x;x:=y`} or "shut the door; turn the key"). We can think of the effect brought about by executing Π in terms of the individual effects brought about by

executing Π_1 and Π_2 individually as follows: if the result of executing Π_1 in state σ_1 is σ_2 and the result of executing Π_2 in state σ_2 is σ_3, then the result of executing the $\Pi_1;\Pi_2$ in state σ_1 is σ_3. This sort of reasoning can codified by the following rule of inference which is part of the natural semantics for `While`:

$$\frac{\langle \Pi_1, \sigma_1\rangle \Longrightarrow \sigma_2 \quad \langle \Pi_2, \sigma_2\rangle \Longrightarrow \sigma_3}{\langle \Pi_1;\Pi_2, \sigma_1\rangle \Longrightarrow \sigma_3} \tag{2.14}$$

For example this rule tells us that since $\langle$`z:=x`$, \langle 1,2,3\rangle\rangle \Longrightarrow \langle 1,2,1\rangle$ and $\langle$`x:=y`$, \langle 1,2,1\rangle\rangle \Longrightarrow \langle 2,2,1\rangle$, we can infer that $\langle$`z:= x;x:=y`$, \langle 1,2,3\rangle\rangle \Longrightarrow \langle 2,2,1\rangle$. And similarly, it tells us that if the result of carrying out "shut the door" in a physical state σ_1 is another physical state σ_2, and if the result of carrying out "turn the key" in state σ_2 is state σ_3, then the result of carrying out "shut the door; turn the key" in σ_1 is σ_3.

Similar rules can be stated for conditional and iterative statements.[17] Together the state transition axiom and the compositional rules can be taken to form a proof system T_{ns} in which derivations correspond to executions of `While` programs. The derivation of corresponding to the execution of Π in initial state σ_0 and resulting in a terminating state σ_n will have $\langle \Pi, \sigma_0\rangle \Longrightarrow \sigma_n$, internal nodes corresponding to of the form $\langle \Pi', \sigma_i\rangle \Longrightarrow \sigma_{i+1}$ for Π' a subprogram of Π. For instance the tree corresponding to the execution of the program {`z:=x;x:=y`}`;y:=z` in initial state $\sigma_0 = \langle 1,2,3\rangle$ will be as follows:

$$\dfrac{\dfrac{\langle \texttt{z:=x}, \sigma_0\rangle \Longrightarrow \sigma_1 \qquad \langle \texttt{x:=y}, \sigma_1\rangle \Longrightarrow \sigma_2}{\langle \{\texttt{z:=x;x:=y}\}, \sigma_0\rangle \Longrightarrow \sigma_2} \qquad \langle \texttt{y:=z}, \sigma_2\rangle}{\langle \{\texttt{z:=x;x:=y}\};\texttt{y:=z}, \sigma_0\rangle \Longrightarrow \sigma_3} \Longrightarrow \sigma_3 \tag{2.15}$$

where $\sigma_1 = \langle 1,2,1\rangle$, $\sigma_2 = \langle 2,2,1\rangle$ and $\sigma_3 = \langle 2,1,1\rangle$. I will write $T_{n,s} \vdash \langle \Pi, \sigma\rangle \Longrightarrow \sigma'$ if there is a tree that can be constructed with the specified root and denote the tree itself by $\mathcal{T}^{ns}_{\Pi,\sigma}$.[18]

It should be clear that $\mathcal{T}^{ns}_{\Pi,\sigma}$ can be taken to represents the execution of Π on input σ in the sense that if such a tree exists, then its internal nodes specify the intermediate states in what we would informally describe as the execution of Π on σ. If we abstract one level

[17] For a complete presentation of the operational semantics of `While` see [100].

[18] Note that it can be shown under the current semantics that `While` is deterministic in the sense that if $T_{ns} \vdash \langle \Pi, \sigma\rangle \Rightarrow \sigma'$ and $T_{ns} \vdash \langle \Pi, \sigma\rangle \Rightarrow \sigma''$, then $\sigma' = \sigma''$. As such the, there is at most one tree with root $\langle \Pi, \sigma\rangle \Rightarrow \sigma'$ and the definition of $\mathcal{T}^{ns}_{\Pi,\sigma}$ just given is well defined.

from this we can look at $\mathcal{T}^{ns}_{\Pi,\sigma}$ as being given uniformly in σ. And on this basis we can view the result of applying the natural semantics for `While` to Π to be given by the function

$$(2.16)\quad op^{ns}_{\texttt{While}}(\Pi) = \lambda\sigma.\begin{cases}\mathcal{T}^{ns}_{\Pi,\sigma} & \text{If there exists } \sigma' \text{ such that } T_{ns} \vdash \langle \Pi, \sigma\rangle \Longrightarrow \sigma' \\ \text{undefined} & \text{otherwise}^{19}\end{cases}$$

Since the result of interpreting Π relative to $op^{ns}_{\texttt{While}}$ is a derivation tree, it thus follow that we should take the class of such a tree to comprise the class $\mathcal{M}_{\texttt{While}}$ of mathematical interpretations of `While` programs. This definition thus provides some insight into the sense in which an operational semantics provides a means of associating programs with mathematical objects.

We may now return to the question of whether the function $op^{ns}_{\texttt{While}}(\Pi)$ can plausibly be used to help us understand algorithm reference via (2.8a). This question can be framed concretely by taking Π to be equal to the program given in (2.9). As noted above, the problem that we face in answering this question is that any intuitions we have about the meaning of (2.9) are presumably grounded in the fact that we view it as a means of formalizing (2.11). So in order to determine whether $op^{ns}_{\texttt{While}}(\Pi)$ is a viable candidate for serving as the algorithm expressed by Π, we need to need inquire about the relationship between this mathematical object and the procedure informally described by (2.11). But as noted above, it does seem as if the natural semantics for `While` *can* plausibly be thought of as deriving from an analysis of the semantic contribution of imperative statements in natural language as they appear in sets of instructions.

Unfortunately, however, this observation alone is not sufficient to demonstrate that a systematic justification can be provided for taking the reference of "the algorithm expressed by Π" to be given by $op^{ns}_{\texttt{While}}(\Pi)$. In order to see this, most convenient to some of the initial worries I voiced about mathematical reference above. For as I noted there, even when a mathematical name t is introduced by an explicit definition of the form $t =_{df} \iota x.\varphi(x)$, we

[19]The "otherwise" clause of this definition comes into effect when there is no well-defined derivation tree for Π on input σ. This occurs in cases where we would informally describe the operation of Π as "looping forever" and thus failing to terminate. It is the presence of the iterative construction `while` φ `do` Π `od` which can lead to the possibility of non-terminating executions of `While` programs. In such cases it is unclear whether it is more natural to assign some representation of the infinite computation of Π on σ as the meaning of Π or to simply stipulate that $\mathcal{S}_{\texttt{While}}(\Pi, \sigma)$ is undefined. But since non-terminating executions rarely arise in the sort of the cases we will be considering, there is no reason to dwell on this point here.

still face the problem of determining the reference of the description $\iota x.\varphi(x)$. In standard cases such as those given in (2.7), this may be facilitated by the fact that we already possess either a model or an axiomatic theory of the language over which $\iota x.\varphi(x)$ is stated. But in the general case, the problem of referential indeterminacy remains in a sense which is aptly demonstrated with respect to (2.7b). For note that if we explicitly agree that the symbol ω has been introduced to abbreviate the description "the least infinite ordinal" (which itself can be taken as an abbreviation for a definite description in the language of set theory), it will still generally be conceded that it is arbitrary whether we take this description to be satisfied by the least infinite von Neumann ordinal ω_N or the least infinite Zermelo ordinal ω_Z.

I have proposed above that we seek to understand algorithmic reference by treating "the algorithm expressed by Π" as denoting a functional expression which can be taken to denote a function $prog(\Pi)$ mapping from programs into algorithms. If this expression can in fact be so interpreted, this allows us to also view (2.10) as an explicit definition on a linguistic par with those in (2.7). But it is also important to realize that there are still more axes of indeterminacy which will plague an account of algorithmic reference developed on this basis than arise in standard accounts of how terms like e, ω or C_6 refer to mathematical objects. As we have seen, one such axis arises because the Π can be drawn from any of a wide class of programming languages or can even be a pseudocode specification. Relative to a choice of language L, I have suggested that we might attempt to specify $prog$ by using an operational semantics for L. Such a semantics can be thought of as giving rise to a mapping $op_L : Prog_L \to \mathcal{M}_L$. But as pointed out above, the class $\mathcal{M}_L$ taken as the semantic values of L programs will vary not only with L, but also with the form of operational semantics we choose to supply for it. Unless additional considerations can be adduced in favor of a particular form of operational semantics, this consideration leads to another substantial axis of indeterminacy.

The forgoing argument attempted to make a case for taking the so-called natural semantics for the language `While` given by $op^{ns}_{\texttt{While}}$ as allowing us to interpret its programs in a manner which coincided with some of our background intuitions about imperative

statements in natural language. But even a cursory examination of the literature of programming semantics reveals that this is just one of several different forms of operational semantics which can be given for imperative languages. In addition, for instance, there are so-called *structural semantics* for `While`. In this setting programs are interpreted as inducing transitions of the form $\langle \Pi, \sigma \rangle \Longrightarrow \gamma$ where $\gamma \equiv \sigma$ if the execution of Π in σ has terminated and $\gamma \equiv \langle \Pi', \sigma' \rangle$ if the execution of Π in σ has not terminated and $\langle \Pi', \sigma' \rangle$ represents the next intermediate step in its remaining computation. It is possible to workout a similar definitions of a deductive system T_{ss} and deduction tree $\mathcal{T}^{ss}_{\Pi,\sigma}$ for the structural semantics. And on this basis, we may also define $op^{ss}_{\texttt{While}}(\Pi)$ which is parallel in structure to (2.16). But obviously since the trees $\mathcal{T}^{ns}_{\Pi,\sigma}$ and $\mathcal{T}^{ss}_{\Pi,\sigma}$ will in general be distinct, this means that the object $op^{ns}_{\texttt{While}}(\Pi)$ which the natural semantics gives as the meaning of Π will be distinct from that given by $op^{ss}_{\texttt{While}}(\Pi)$.

The structural semantics for `While` offer certain technical advantages over the natural semantics. For instance, deduction trees of the form $\mathcal{T}^{ss}_{\Pi,\sigma}$ will correspond to linearly ordered sequences of configurations of the form $\gamma_0, \gamma_1, \ldots$ such that $\gamma_0 = \langle \Pi, \sigma \rangle$ and such that $\gamma_i \Longrightarrow \gamma_{i+i}$ for all $i > 0$. As such sequences will be finite just in case the computation of Π on σ halts, this property facilitates the construction of correctness proofs. However both the natural and structural semantics include the same state transformation axiom (2.13). And since this was the point of contact with natural language imperatives on the basis of which I motivated the natural semantics above, it appears unlikely that any intuitions which we have about the meanings of `While` programs can be cited in favor of adopting one of $op^{ns}_{\texttt{While}}(\Pi)$ or $op^{ss}_{\texttt{While}}(\Pi)$ over the other as formalizing what we mean by "the algorithm expressed by Π."

I will take this consideration to demonstrate that if algorithms are indeed abstract mathematical objects, our ability to refer to them cannot be completely explained in terms of the schema (2.8a). But since we still need to give an account of algorithmic reference so as to explain the status of statement like those given by (2.5), it remains incumbent upon an algorithmic realist to either show how the present account can be repaired or to formulate a more successful one which can replace it. The thesis for which I wish to argue in the rest of this section is that additional headway can be made on both front by considering how

(2.8b) can be used both on its own and in conjunction with those of (2.8a) in course of referring to algorithms.

Algorithms and implementations

It is somewhat more difficult to construct a completely explicit example of an instance where apparent reference to an algorithm A is made via the expression "the procedure implemented by M." This owes both to the fact that this means of referring to procedures is generally used in specialized contexts within computer science. And it is also due to the fact it is somewhat more difficult to explicitly specify an implementation than it is a program. These caveats aside, however, the idea motivating the proposal that we can refer to an algorithm by refering to an implementations is reasonably straightforward. For note that common sorts of inplementations such as Turing mcahinces may also be thought of as mean of explicitly representing particular computational methods which may be carried out uniformly over a class of input values. In specifying an instance M of such a model, it is thus reasonable to claim that we have provided a means of referring to a general method which M instaniates – i.e. the algorithm implemented by M.

In the course of our computational practices, the sorts of contexts in which (2.8b) is standardly applied are ones in which the implementation M has already been explicitly specified. We might, for instance, speak of the algorithm QUICKSORT as being the procedure expressed by a particular instance of a `Java` virtual machine or a `MIPS` machine. Such models are, howver, quite complex. And thus due to the sheer number of combinatorial details which must be attended to in order to precisely define one of their members, it is relatively rare in everyday procedural discourse for individual algorithms to specified by (2.8b). However we will instances of this schema at work in Chapters 3 and 4. And I will also argue below that it ought to be viewed as our paradigm device for referring to algorithms.

The first issue to which we must attend is that of circumscribing what is meant by an implementation. As I have mentioned above, the notion of an implementation is essentially that of an instance of a model of a computation and as such implementations bear the same relation to such a model as do programs to programming languages. Roughly speaking,

a model of computational may in turn be characterized as a class of uniformlt structured mathematical structure taken together with a definitio of application which allows them to be viewed as representations of temporally extended computational processes. Since it is possible to take different about what constitutes a computational process, it is possible to give different general definitions of what constitutes a model of computation. This makes it difficult to provide a truly comprehensive definition of this notion. For present purposes, it will useful to follow the route taken by standard textbooks on computational models (such as those of Savage [118] and Hopcroft and Ullman [63]) and attempt to define a model of computation by example.

Some of the best known examples of formalism which we now think of s models of computation date from the foundational period in the history of comutability theory. These includes the Turing machine and the untyped λ-calculus. These models were designed so that the primitive mathematical operations relative to which they are defined could each be given a constructive justification so as to facilitate the defense of the claim that the class of functions which their operation induce could are intuitively effective. The majority of the models which now find application in computer science, however, were developed after the period when the analysis of effectivity was a primary theoretical concern. The features of recent models of computation are thus generally justified on the basis of either their in principle physical implementability or their resemblance to extant physically embodied computing devices.

Since the 1930s, literally hundreds of such models have been defined. And although it is again probably safest to refrain from attempting to provide a uniform characterization of what such models have in a common, several general areas of commonality can be noted:

1) A model of computation can be formally taken to consist in a quadruple $\mathcal{M} = \langle \boldsymbol{M}, X, Y, App \rangle$ which I will respectively refer to as its *implementations* (or occasionally as the *machines*), the *input set*, the *output set* and the *definition of application*. I will generally abuse notation and write $M \in \mathcal{M}$ (as opposed to $M \in \boldsymbol{M}$) to denote that implementation M is drawn from model $\mathcal{M}$.

2) Individual implementations $M \in \mathcal{M}$ are intended to be interpretable as inducing functions from X to Y. This is accomplished by an operation known as *application* whereby M the computational operations described by M are applied to a value $x \in X$, giving rise to a sequence of intermediate configuration $\sigma(x)_0, \sigma(x)_1, \ldots$ which may be of either finite or infinite length. I will refer to such sequences as the *execution* of M on input x and denote it by $exec_M(x)$. The formal definition of how this sequence is derived in terms of the structure of M is given by the definition of *App*. If $exec_M(x)$ is finite, I will write $App(M, x)$ to denote the value $y \in Y$ corresponding to its final member.

3) Generally, but not always, the elements of $exec_M(x)$ will correspond to what I will refer as the *computational states* Σ of M. These will correspond to various configurations of its combinatorial components which are modified during the course of an execution.

4) A *transition-based* model is one whose computational states can be characterized as structures upon which it is possible to define a notion of *locality* around which updates (corresponding to computational steps) are made. The definition of *App* for such a model can thus be given relative to a *state transition function* $\Delta : \Sigma \rightarrow \Sigma$ whose values are constrained by the relevant notion of locality. An execution of a transition based model can thus be thought of as a sequence $\sigma_0, \sigma_1, \ldots$ where σ_0 corresponds to an initial state of M (possibly parameterized in its input) and $\sigma_{i+1} = \Delta(\sigma_i)$ for $i > 0$. A paradigm instance of such a model is the single-tape, single-head Turing machine originally introduced by Turing [144]. This class also includes well-known classes of automata (deterministic and non-deterministic finite state, push-down, linear bounded, etc.) as well as the historically important Schönage storage modification machine [120].

5) A *register-based* model is one whose computational states can be characterized as finite or infinite vectors of storage locations $\langle l_1, l_2, \ldots \rangle$ which can be accessed and operated on by numerical index. The definition of *App* for such a model can be given relative to an *update function* $Update : \Sigma \rightarrow \Sigma$ which takes an input an index i, an operation f (which are encoded as part of its computational state σ) and returns as value a computational state σ' where the value stored l_i has been updated by applying f to one or more other values. Paradigm examples of models in this class include the *random access machine*

of Hartmanis [59] and its many variants.

6) A *recursion-based* model is one whose definition of *App* is based on recursion on a class of objects D relative to some well-founded relation $<_D$ defined on this class. Paradigm examples of models in this class include the *primitive recursive definitions* of Skolem [132] and Gödel [45], the *general recursive definitions* of Gödel [44] and Kleene [65], and pure `LISP` of McCarthy [85]. With suitable modifications, well known formalisms based on term rewriting such as the typed and untyped λ-calculus can be assimilated to this paradigm.

Although lacking precision along several dimensions, the terminological framework introduced in 1) - 6) will provide us with a uniform basis for talking about implementations in the sequel. Returning now the topic of procedural reference as mediated by implementations, it will useful to fix an example of (2.8b). To this end, consider the statement

(2.17) INSERTIONSORT is the procedure implemented by M.

Here I will assume that M is an example of a fixed implementation – say a RAM machine – which is a member of some formally defined model of computation $\mathcal{M}$. It is somewhat more common to see simple procedures like INSERTIONSORT described linguistically – i.e. by a program or pseudocode specifications – and as noted above this is indeed how algorithms are generally introduced in textbooks and journal articles. But in the actual practice of computer science, it is fairly common to explicitly specify implementations by standard mathematical definitions. As we will see in Chapter 4, for instance, this can be accomplished in the case of RAM machines by specifying a certain form of transition function defined over a vector of storage locations.

The conceptual question which arises when we reflect on our use of statements like (2.17) is, of course, how is it that we are able to make reference to an algorithm by referring to a structured mathematical object like a RAM machine. One point which may immediately be noted in this regard is that the expression "the algorithm implemented by $\cdot$" serves grammatically as a functional expression in much the same way as "the algorithm expressed by $\cdot$." I suggested in the previous subsection that in the latter case some purchase may gained on understanding how such an expression refers to an algorithm by attempting

to semantically interpret programs in something like the way we attempt to semantically interpret sentences. But it is clear that no parallel analysis can be applied in cases like (2.17) because implementations are mathematical as opposed to linguistic objects. Thus if "the algorithm expressed by M" as used in (2.17) succeeds in referring to an algorithm, it cannot be because we have semantically interpreted M to refer to such an object because implementations no more admit to interpretation than do natural numbers.

Given that implementations like M simply are mathematical objects, it seems natural to suggest that we take expressions like "the algorithm implemented by M" to denote the identity function. This would have the effect of simply equating algorithms with implementations. Whatever its other consequences, this proposal would have the consequence of satisfying one of the central tenets of algorithmic realism. For if in order to refer to an algorithm A, we needed to refer to an implementation M, which was then discovered to be identical to A, then A would *ipso facto* be a conventional mathematical object like M. But one does not need to look very far to see that this would be a Pyrrhic victory for algorithmic realism. For a number of theoretical practices in computer science appear to rely on the fact that a distinction between algorithms and implementations be preserved. As such, any attempt to collapse such a distinction from the outside would correspond to a form of realism about algorithms which failed to conform to the details of our computational and mathematical practices.

One easy observation which can be made in this regard is that not only does it appear possible to discover an algorithm without thereby discovering an implementation, but that this is almost always the way things happen in practical mathematical practice. Euclid's algorithm, for instance, was first described over 2500 years before the first formal definitions of implementations were given in 1930s and bore the name "Euclid's algorithm" for at least 100 years prior to this point. The same is true for a variety of numerical procedures which were developed prior to the twentieth century: tableau and Russian peasant multiplication, Newton's method, the Sieve of Erosathenes, the Lucas-Lehmer test, Gaussian elimination, MERGESORT, etc.. Each of these cases represents an instance in which a particular mathematical procedure was presented in mathematical practice (even up to the point of being given a proper name) without the explicit representation in terms of what we now refer to

as implementation. And as such, this at least suggest that our ability to refer to algorithms cannot be tied to any particular class of these objects.

The other readily framed observation about our computational practices which suggests that the denotation of "the algorithm implemented by M" cannot be taken to be M itself is that it is typically possible to refer to a single algorithm using more than one implementation. The situation here is very much the same as it is with programming languages. For recall that I alluded above to the widespread belief that all sufficiently "high-level" languages are capable of expressing any intuitively effective algorithm. But since the notions of "high-level" and "express" are only vaguely defined with respect to the conventional interpretation of this statement, it is safest to regard such a generalization as at best folkloric. An analogous statement about implementations would have the following form:

(2.18) For all models of computation $\mathcal{M}$ meeting conditions Φ, and for all intuitively effective algorithms A, there is a $M \in \mathcal{M}$ such that M implements A.

With sufficient attention to the definition of the statement of Φ and analysis of intuitions about implementation, it is possible to make (2.18) into a reasonably precise statement about the relationship between algorithms and implementation. And as I now wish to briefly discuss, with sufficient attention to these details and to the choice of $\mathcal{M}$, various versions of (2.18) have been put forward by practitioners as informal but rationally defensible theses about the relationship between implementations and algorithms.

The first point to note about (2.18) is its relation to Church's Thesis. Traditionally this proposal is reported in the summarized form "All algorithmically computable functions are general recursive" which equates the algorithmic computability of a given function f on the natural numbers with the fact that f (extensionally) coincides with the function determined by some general recursive function definition. It is well-known, however, that many of the most familiar models of computation are extensionally equivalent in the sense of containing members which (subject to an appropriate effect encoding) determine the class of functions on the natural numbers.[20] For present purposes we may therefore present Church's Thesis in a more general form as follows:

(2.19) For all models of computation $\mathcal{M}$ meeting Ψ, and for all intuitively effective

algorithm A determining a possibly partial function $f_A : X \to Y$, there exists an implementation $M \in \mathcal{M}$ such that $\forall x in X[f_A(x) = App(M, x)]$ and conversely for all $M \in \mathcal{M}$, there exists a intuitively effective algorithm such that $\forall x \in X[App(M, x) = f_A(x)]$.

Here the predicate Ψ can be taken to incorporate two kind of conditions: 1) that the structures and operations in terms of which $\boldsymbol{M}$ and App are defined are constructive in the appropriate sense; 2) that $\mathcal{M}$ contains members which can be shown to emulate the operations of any member of a reference model of computation such as the single-tape, single-head Turing machine.[22] Since these properties are known to be satisfied by many of the models of computation mentioned above, (2.19) can be taken as stating that any one of these models is sufficiently expansive to contain a member computing the function induced by any intuitively effective algorithm.

This may seem like a very broad and powerful statement relating implementations to algorithms. But as pointed out in Chapter 1, it must be kept in mind that (2.19) itself does not directly relate algorithms with implementations. Rather it relates the function computed by algorithms with that computed by implementations. Even stated in this generalized form, Church's Thesis is thus most accurately perceived as an *extensional* statement about the class of functions computed by individual algorithms and not a statement about how individual algorithms can be implemented by Turing machines or some other model of computation. But by directly invoking the notion of implementation, on the other hand, (2.18) apparently does make a claim about not just *what* function can be computed by the members of $\mathcal{M}$ but also *how* they may be computed by its members. In particular, it makes the claim that given an algorithm A, there is a member of $\mathcal{M}$, in virtue of implementing A,

---

[22]Suppose that $c_i : \mathbb{N} \to X_i$ and $d : Y_i \to \mathbb{N}$ for $i \in \{1, 2\}$ are respectively effective bijective encoding functions for the inputs and output sets of $\mathcal{M}_1$ and $\mathcal{M}_2$. Then using the notation introduced adopted above, the result just alluded to can be formulated as follows: $\forall M_1 \in \mathcal{M}_1 \exists M_2 \in \mathcal{M}_2 \forall n[d_1(App_1(M_1, c_1(x))) = d_2(App_2(M_2, c_2(x)))]$.

[22]The first of these results states a necessary condition required to ensure that $\mathcal{M}$ is a so-called *effective* model of computation. Since is an intuitive notion, effectively must here be taken as a term of art which is typically described by example. Paradigmatically, for instance, the various features of the Turing machine model – e.g. the finiteness of a Turing machine's alphabet, the fact that it may only observe one square at a time, etc. – are described as individually effective in the sense of being both finitary and applicable to a resource bounded computed agent. This sort of analysis can, however, be substantially generalized so as to encompass all of the models of computation mentioned above – cf., e.g, Gandy [42].

which somehow represents its mode of operation.

As things currently stand, however, no precise sense has been assigned to the notion "mode of operation." And thus to a potentially greater extent than Church's Thesis itself, (2.18) must be taken as an informal thesis rather than any sort of formal adequacy theorem. But for present purposes, what is more significant is that a variety of theorists have endorsed versions of (2.18) as characterizing the relationship between implementations and algorithms.[23] Although their endorsements have often been categorical, we will see in subsequent chapters that there are substantial matters of detail to attend with respect to how the requirements on models Φ are formulated. In particular, it must be ensured that $\mathcal{M}$ is sufficiently power that individual steps in the executions of implementations $M \in \mathcal{M}$ can mirror those of the algorithms studied in algorithmic analysis and complexity theory. And as we will see, this is true of some, but not all of the models which satisfy the requirements Ψ.

What is significant for present purposes, however, is that (2.18) *can* plausibly be taken to be true with respect to certain explicitly definable models of computation. One notable example, for instance, is the class of so-called `MIX` machines introduced by Donald Knuth. This class was introduced as a reference model in Knuth's seminal (and encyclopedic) work on algorithmic analysis [72] in an attempt to isolate a computational formalism which was both mathematically streamlined and also took into account some of the features of real world electronic computers. What is remarkable about [72] is that for virtually every algorithm A which he considers in detail, Knuth presents a explicit implementation M_A of A as a `MIX` machine. Although Knuth himself does not use this to argue for an explicit thesis like (2.18), the work he has performed in constructing `MIX` implementations which accurately reflect the properties of the informally specified algorithms he discusses is strong inductive evidence that such a proposal ought to be accepted.[24]

---

[23] The textual situation with respect to (2.18) is complicated somewhat by a tendency of theorists to refer to this statement (as opposed to one resembling (2.19)) as "Church's Thesis." Since the notion of individual algorithm was not clearly delineated by any of the framers of Church's in 1930s (i.e. Gödel, Church, Turing, Kleene and Post), this is almost certainly a historical confusion. Bearing this in mind, versions of (2.18) (sometimes advertised as "Church's Thesis") can be found in the writings of Gödel [44] (postscript), Kreisel [69], Rogers [113], Matchey [80] and Lewis and Papadimitriou [74].

[24] Knuth's own views on this are this subject are not entirely clear. For note that in the introduction to the first volume of [72], he does state a thesis akin to (2.18) but with respect to a simpler class of what he

Conventional wisdom thus reflects that there is at least one of the model of computation in contemporary use which satisfies Φ. But note there is also no reason to suppose that `MIX` corresponds to the only model with this property. For instance, in the more recent volumes of [72], Knuth employs a updated form of the `MIX` model known as the `MMIX` machine by which to provide implementations of the algorithms he considers. And although many details have to be checked in each case, it is currently possible to provide plausible arguments that a number of other models can be used in this capacity – e.g. the `Java` machine [76], the `MIPS` machine [107], the Schönage storage modification machine [120], the interpreted `While` programs of Jones [64].

This situation has substantial ramifications with respect to for how we ought to understand statements like (2.17). For suppose that there are at least two models $\mathcal{M}_1$ and $\mathcal{M}_2$ which satisfy Φ and without loss of generality, assume that there classes of implementations $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are disjoint. Then it follows from (2.18) that for every informal algorithm A, both $\mathcal{M}_1$ and $\mathcal{M}_2$ contain at least one machine implementing A. If we call these implementations M_1 and M_2, we would then have

(2.20) a) A is the algorithm implemented by M_1;

b) A is the algorithm implemented by M_2.

From this it follows that the denotation of the expression "the algorithm implemented by $\cdot$" as it appears in these statements cannot be the identity function. For since M_1 and M_2 are, by supposition, distinct mathematical objects we cannot consistently have $A = M_1$ and $A = M_2$.

If we wish to understand how reference to algorithms is possible via (2.17), something additional must thus be said about the meaning of the expression "the algorithm expressed by $\cdot$." One possibility is that we treat this expression as denoting a mapping $imp : \mathcal{M} \to \mathcal{A}$ which maps classes of implementations $\mathcal{M}$ (which may include machine drawn from more than one model) of into algorithms in a possibly many-one manner. Such a mapping is obviously similar in form and motivation to the mapping $prog : Prog_L \to \mathcal{A}$ discussed in the

---

refers to as *computational methods* (cf. p. 7-9). However, this model is not referenced again throughout the rest of the work. I will discuss the evolution of Knuth's at greater length in Chapter 5.

prior section which is intended to map programs over a language L (which might be taken to consist in the union of several traditional programming languages) into algorithms. But note that in this latter case, there was an obvious manner in which we could set out to define *prog* on the basis of the fact that programs are linguistic entities and thus admit to several forms of compositional semantic interpretation. And it was on this basis that I proposed taking *prog* to be given by the interpretation function op_L induced by an appropriate form of operational semantics for L.

But matters stand somewhat differently with respect to giving an explicit definition of *imp*. For in particular, implementations of the sort we have been considering are not linguistic entities and as such there is no clear sense in which they admit to linguistic interpretation.[25]This means that there is little hope that *imp* might be defined on $\mathcal{M}$ so that the value of $imp(M)$ is determined compositionally in terms of the structure of M. For while implementations generally are structured mathematical entities, there is no reason to think that the structure of a Turing or RAM machine can recursively determine an algorithm in anything like the way we think of the structural decomposition of a sentence as determining a proposition.

As things stand, there is thus much left to be explained about how reference to algorithms can be achieved by (2.8b). The considerations just adduced suggest that there is little hope of attempting to produce an explicit definition of *imp* which resembles the definitions of op_L applicable to programs. In Section 3, I will argue that rather than attempting to give an explicit definition of the function *imp*, the realist will be better off attempting to define this function implicitly. As of yet, it is difficult to give concrete meaning to such a suggestion. For, in particular, we have yet to see whether there any other means of effective reference to algorithms with which such a definition would ultimately have to comport. I will argue in section 2.4 that, other things equal, there are not. And in sections 3 and 4, I will attempt to show on this basis how such an implicit definition might be given.

2.2.4 Algorithmic reference by other means?

Over the course of the previous section, I attempted to illustrate how it may be possible to refer to algorithms through the use of programs and implementations. Provisional on

our ability to resolve the determinacy problems just reviewed, these considerations suggest that an ability to refer to such entities is *sufficient* to enable us to refer to algorithms. I now want to consider the claim that it is also necessary – i.e. that it is impossible to refer to an algorithm A without having first referred to a program which expresses it or a machine which implements it. Although the significance of this result may not be immediately apparent, it will turn out to be important with respect to locating the concept of algorithm with respect to traditional classifications of sortal concepts, a task I will begin to undertake in Section 3.

In attempting to argue that procedural reference must indeed be mediated by reference to programs or implementations, we face the initial methodological problem that this claim has universal scope. To demonstrate it, one might think that we must thereby examine all other potential mechanisms of reference to algorithm and conclude individually that each was deficient. But of course it is far from clear what ought to be counted as a "referential mechanism" even in paradigm cases of concrete macroscopic objects. And thus since the prospects of uniformly enumerating different ways in which we could potentially refer to algorithms seems dismal at best, one might think it implausible that such a claim could ever be determinately demonstrated.

And as such, it seems that we can do little better than to consult our intuitions about other possible means by which we might succeed in referring. The most obvious such device is that of ostention. However, given our current understanding of an algorithm as a procedure which operates on mathematical objects, it is far from clear what it would mean to ostend an entity such as Euclid's algorithm. For note that although we may not yet have a solid grasp on the ontological status of such a procedure, we do know that this procedure operates by transforming pairs of natural numbers $\langle n, m\rangle$ according to a rule which tells us how to sequentially update the values of n and m. Since this rule is itself given by a mathematical function g (i.e. that satisfying the implicit definition $g(n, 0) = n$ and $gcd(n, m) = gcd(m, m \bmod n)$) which would typically be part of the presentation of this algorithm, it seems doubtful that we could refer to Euclid's algorithm unless we are also able to refer to g. But g is a abstract mathematical object, and thus presumably not a possible target of ostension. As such, it seems doubtful at best that our ability to refer

to Euclid's algorithm can be explained in this manner.

A potentially more promising route by which we might seek to explain algorithmic reference involves the use of the notion of execution introduced above. For note that as I discussed in Chapter 1, accompanying every informal algorithm A we generally possess some grasp of what it means to carry out or execute A. It is, of course, this notion which is formalized as the definition of $App(M, x)$ relative to a particular model of computation. Generally speaking, however, our informal notion of executing A corresponds to the sort of calculation which we might carry out by hand in the course of performing an arithmetic problem – i.e. a sequence of intermediate numerical or symbolic configurations $\sigma_0, \ldots, \sigma_{n-1}$ from whose final step the output of A is derived.[26]

The fact we possess such an independent conception of carrying out an algorithm, suggests that it may be possible to refer to an algorithm by referring to an execution. In particular he might imagine that in parallel of (2.8a,b), we can also refer to an algorithm through the use of the schema

(2.21) $A =_{df}$ the procedure with execution $\sigma_0, \ldots, \sigma_{n-1}$.

One point in favor of (2.21) is that if we were indeed able to refer in this manner, then this lowers the requirements on algorithmic reference in general. For at least in the paradigmatic case, an execution consists of a finite sequence of computational states. And while these states will presumably still be abstract mathematical structures, one might think that there were fewer background hurdles to be overcome in explaining reference to such sequences than to the infinitary sets of states and transition functions out of which the formal specification of an implementation will often be comprised. In particular, it seems possible to refer to a (finite) execution by ostending the physical tokens of the symbols of which it consists (e.g. the entries on an adding machine tape). And given the well-known problems about reference to infinitary mathematical objects through their putative physical embodiment as concrete devices, one might think we were better off attempting to refer

[26] I will here ignore the case where executions are infinite. For famous reasons, this possibility cannot be ruled out for sufficiently general models of computation. However, this simplification will not affect the point of my argument.

to algorithms via (2.21) then by either of (2.8a,b).[27]

This advantage turns out to illusory, however, because it is quite clear that (2.21) cannot be accepted as a viable means of referring to an individual algorithm. There are two reasons for this. First, it will generally be the case that infinitely many distinct algorithms can generate any fixed execution $\sigma_0, \ldots, \sigma_{n-1}$ on the appropriate input. This means, for instance, that although we may identify a sequence of pairs $\langle 289, 131\rangle, \langle 131, 27\rangle, \langle 27, 23\rangle, \langle 23, 4\rangle, \langle 4, 3\rangle, \langle 3, 1\rangle, \langle 1, 0\rangle$ as corresponding an execution of Euclid's algorithm, we may not justifiably work backward from this sequence to the conclusion that it was generated by carrying out this algorithm. For in the general case, there will be infinitely many intuitively distinct mathematical procedure which could give rise to such an execution.

One solution to this might be to attempt to specify an entire class of executions of the form $\sigma_0(x), \sigma_1(x), \ldots$ parameterized by what we take to be A's input x. But even if we were somehow able to determinately refer to such an infinite set of sequences, this would not solve the more fundamental problem. For note that given any extensional description of the potential behavior of some mathematical procedure – say the function $f : \mathbb{N} \to \mathbb{N}$ which the procedure was claimed to compute or even the entire class of sequences state $\vec{\sigma}(n)$ for each $n \in \mathbb{N}$ through which is claimed to induce – it still generally impossible to determine what algorithm induced the behavior.[28]As such, it appears that neither (2.21) nor any obvious variant involving executions allows us to make determinate reference to individual algorithms.[29]

The foregoing observations provide at least some inductive evidence that since other referential mechanism do not seem to be forthcoming, (2.8a,b) may indeed exhaust the

---

[27]The *locus classicus* for such worries is Kripke [70]. In assuming that we can determinately refer not only to abstract implementations such as individual Turing machines but also that a determinate notion of execution is specifiable for such machines, I largely ignored the problems about mathematical meaning and reference raised therein. These are, however, worries which an algorithmic realist would ultimately have to confront if he wished to argue that not only are algorithms abstract objects, but also that our ability allows us to ground the meaning of other non-procedural mathematical notions.

[29]This is essentially the content of a formal result in computability theory known as Rice's Theorem. Call $S \subseteq \mathcal{P}(\mathbb{N})$ a *extensional property* of a computable function just in case if for all $i, j \in \mathbb{N}$, if $\varphi_i = \varphi_j$ (i.e. the partial recursive function with the indices i and j determine the same function), then $\varphi_i \in S$ if and only if $\varphi_j \in S$. Call S *non-trivial* if $S \neq \emptyset$ and $S \neq \mathcal{P}(\mathbb{N})$. Rice's Theorem then states that if S is non-trivial, then problem of deciding whether $\varphi_i \in S$ given i is undecidable. Examples of non-trivial properties are the set of computable functions which determine a non-empty language, the class of computable functions which halt on all inputs and, most germanely, the class of recursive functions which compute a given function.

means at our disposal for referring to algorithms. But what I now want to suggest is that potential modes of algorithmic reference are yet more circumsribed than this. The basis of this claim is the observation, that when properly understood, putative instances of reference to algorithms mediated via (2.8a) can be subsumed under reference mediated via (2.8b). In order to see this, first recall that I argued in section 2.3.1 that it was at best unclear whether reference to programs actually allows us to refer to algorithms. The crux of my argument turned on the fact that even if we view operational semantics as an attempt to analyze the meaning of programs in a compositional manner which conforms to pre-theoretical intuitions grounded in a comparison of programming languages to natural language, the abstract objects assigned as the result of interpreting Π relative to such a semantics cannot be taken as the algorithm which Π expresses. For we appear to have such intuitions about programs only over a small class of languages L like WHILE. And even in such case, there will generally be different forms of operational semantics which give rise to distinct interpretation functions $op^i_L : Prog_L \rightarrow \mathcal{M}_i$ which map L programs into different classes of formal objects. But whatever semantic intuitions we have about L will generally be insufficient to distinguish which, if any, of the object $op^i_L(\Pi)$ ought to be taken as the denotation of "the algorithm expressed by Π." And it therefore seems that seems that compositional semantics for programming languages cannot be of direct use in helping to us understand how we can refer to an algorithm by referring to a program.

Another way of putting the observation is that we have not, at least as of yet, not seen any justification for treating any of the classes $\mathcal{M}^1_L, \mathcal{M}^2_L, \ldots$ as themselves constituting the domain of algorithm $\mathcal{A}$ which the realist wants to characterize. But note that we now do possess another means of characterizing these classes – i.e. as constituting models of computation. In particular, it should now be evident that objects $op^i_L(\Pi)$ derived by interpreting programs Π according to a particular form of operational semantics will generally be characterizable as an implementation. Since we currently lack a sharp definition of implementation it is difficult to make this claim as precise as desirable. But this will be obviously true for forms of operational semantics which operate through the "interpretation via compilation" paradigm briefly discussed above. For in such cases, programs are first translated into a form of so-called *machine code* – i.e. sequence of instructions which can

be directly executed by a given form of real or notional miscroprosessor such as the `MIX` or `MIPS` machine. Such programs generally stand in a one-to-one correspondence with specific instances of these models. And thus the practice of interpreting a "high level" program such as `Divide` in this manner leads directly to an recognizable form of implementation. After we study implementations in more detail in Chapter 4, it will be evident, however, that even other forms of operational interpretation functions such as $op^{ns}_{\texttt{While}}$ determine a class of implementations as their range.[30]

One consequence of this observation is that although operational semantics cannot be taken as means of specifying algorithms as the interpretation of programs, it does map programs into another well-known class of computational object. Such interpretations may thus be seen as a potential stepping stone in an explanation of how we can use programs to refer to algorithms. But taking into account the considerations adduced in this section, it cannot itself yield a complete account of how such reference is possible. For as we have seen, implementations identified with algorithms by virtue of the fact that different machines M_1, M_2 can be taken to implement the same algorithm A. And thus on the basis of foregoing observations, this gives us another reason to dissent from the conclusion that the semantic interpretation of a programs eventuates in an algorithm.

Taken collectively, the observations recorded in this section 2.3.1, 2.3.2 and this section thus suggest that the realist will be best off adopting the following overall picture of algorithmic reference:

(2.22) a) Reference to algorithms must be understood indirectly in the sense that there seems to be no plausible way we can refer to an individual procedure like Euclid or Mergesort by a process like ostension.

b) In practice, we typically employ one of the two schema (2.8a,b) to refer to individual algorithms. However, instances of these schemas cannot directly be viewed as complex mathematical descriptions. For among other things, they contain the undefined functional expressions "the algorithm expressed by

[30]This can already be seen by observing that the class of derivation trees $\mathcal{T}^{ns}$ are essentially an abstract representation of individual executions. In particular, the nodes of these tree can be taken as computational states and the definition of the deductive calculus T_{ns} can be taken to induce a definition of application defined over such trees.

program Π" and "the algorithm implemented by machine M."

c) If we seek to understand the meaning of these expressions, we are led to the conclusion that inasmuch as it makes sense to employ conventional notions of semantic meaning program, the interpretation of a program ought to be taken to be an implementation and not an algorithm.

d) For this reason, all instances of procedural reference must ultimately be understood as being mediated by (2.8b). This means in particular that if we wish to understand instances of (2.8b) as successfully referring to algorithms, we must first interpret Π as denoting an implementation $M = op_L(\Pi)$ by specifying an appropriate form of operational semantics. And only then may we ask after the algorithm which M implements by seeking an interpretation of (2.8b) of the sort described at the end of Section 2.3.2.

In the next section I will begin to convert these considerations into a positive account of the ontology of algorithms. Along the way I will that the strategy outlined appears to be the best and only option for vindicating algorithmic realism in the manner discussed in Section 1.

2.2.5 *Algorithm* as an abstract sortal

The significance of this observations collected in (2.22) should not be understated for these technical conclusions can also be taken to correspond desiderata allow us to locate the notion of algorithm with respect to a variety of traditional metaphysical distinctions. The first such classification which I want discuss is that of whether individual algorithms ought to be regarded as abstract or concrete entities. As will be familiar to readers familiar with the recent debate about nominalism in the philosophy of mathematics (as chronicled, for instance, by Burgess and Rosen [16]), there is some disagreement about how this distinction should best be drawn. And thus although it appears as if algorithms appear to satisfy many of the conditions which are traditionally put forth as necessary conditions on abstractness, it is not entirely clear whether they should be placed with respect to the abstract/concrete distinction.

To see this we may begin by noting recalling that abstract objects have traditionally been characterized on their basis of inaccessibility to the sense, causal inertness, and lack of spatial and temporal location. Mathematical objects are standardly thought of as satisfying all three sets of desiderata and are thus standardly regarded as a paradigm case of the abstract. Of course, at this stage to simply assume the same was true of algorithms in virtue of the fact that are mathematical objects beg the very question against algorithmic realism which we are trying to answer. But at the same time, I have already observed that it seems reasonable to suppose that whatever algorithms are, they both operate on any are specifiable in terms of mathematical objects, it is reasonable to assume that they too satisfy the traditional "negative" criteria of abstractness.

There, is however, at least one consideration which appears to stand in the way of settling on this classification. For note that while it seems appropriate to say that a procedure such as EUCLID has no spatial location, we often do speak of executing algorithms using language which suggests that they are located in time. For instance, not only does it make sense to speak of executing EUCLID at temporally located times occasions, but we also ought to use various temporal locutions to describe of executions themselves – e.g. we speak of one step in an execution as occurring *after* another and even of one execution as having a longer *duration* than another. In this sense, a variety of affinities can be cited with respect to the concept *algorithm* and a class of other concepts such as those of *language*, *game* and *musical* or *dance compositions* whose members are taken to be temporally but not spatially located. For instance, while the game of chess or the English language certainly do not appear to be located in space, a case can be made for the fact that they are not atemporal in the sense that they came into being at a certain points in human history and have subsequently evolved over time. And similarly, while a musical composition is itself presumably not located in space, we do speak of different performance of it as occurring in time and have duration.

In Chapter 4 I will attempt to illustrate that the use of similar temporal terminology to describe the operation of algorithms is both widespread and systematic. But barring the potential for a systematic means of de-temporalizing our discourse about algorithms, it would also be useful to determine how algorithms whether algorithms ought to be classified

as abstract or concrete on the basis of extent means of classification. In regard to both algorithms and the other sort of problematic or borderlines cases of abstractness mentioned in the previous paragraph, it is useful to also consider a family of tell known criteria first introduced by Dummett [29].

Dummett's criterion is based on a distinction between objects which are the possible objects of ostension and those which are not. On the basis of observation that we can ostend an object only if it is located and space and time (in which case it can also presumably enter into causal interactions as well), it seems plausible to regard the possible ostendability as at least a sufficient condition for concreteness. But what is of more interest is Dummett's negative characterization of abstract objects as those which cannot be referred to via ostension. But given that be cannot refer to the members of a class of objects $\mathcal{C}$ via ostension, one might reasonably wonder how it was possible to refer to the $\mathcal{C}$s at all.

Dummett's answer is that there are many such classes such that the only way of referring to the $\mathcal{C}$s is by the use of a functional expression applied to a term denoting a member of another class $\mathcal{D}$. In schematic form, Dummett's positive characterization of the abstractness is as follows:

(2.23) If a class of objects $\mathcal{C}$ are such that it is only possible the $v \in \mathcal{C}$ by an expression of the form $f(t)$ (i.e. such that $f(t) = u$ is true) where t denotes a member v of some other class $\mathcal{D}$, then the $\mathcal{C}$s are abstract.

Paradigm example of the application of this criteria correspond to the Frege's [37] well-known examples of the class $\mathcal{C}_1$ or directions and $\mathcal{C}_2$ of shapes.[31]For note that while it appears impossible refer to a particular direction (e.g. north) via ostension, we can readily refer to them by specifying lines of which we customarily say the directions are "of." Similar remarks apply to reference to shapes via geometric figures. In particular, we may schematically represent the expression by which we are able to refer to the directions and shapes as follows

(2.24) a) $x_1 =$ the direction of t_1;

b) $x_2 =$ the shape of t_2

where t_1 and t_2 are terms objects in the classes $\mathcal{D}_1$ and $\mathcal{D}_2$ correspond to particular lines and shapes. On the basis of availability of these schemas and our apparent inability to refer to the members of $\mathcal{C}_1$ and $\mathcal{C}_2$ by some other way, we may conclude that these classes are abstract by Dummett's criterion.[32]

If Dummett's functional criterion does indeed provide a sufficient condition for regarding a class of objects as abstract, then that the considerations adduced above would constitute a strong case for regarding them as abstract. For note that the referential situation in which we appear to stand with respect to algorithms is very much like that in which, per Frege and Dummett, we stand in with respect to directions and shapes. For according to their view, if we want to speak of a particular direction d or shape s, we must produce a particular line ℓ or figure g and then speak of d and s as being determined relative to these entities. This view not only seems plausible in own right, but it appears an obvious affinity to the situation in which, per sections 2.4, we find ourselves with respect to algorithms. In particular, it seems that if we want to refer to MERGESORT we have little choice but produce a either a program Π which we then use to refer it to as "the algorithm expressed by Π" or an implementation M which we then to use to refer to as "the procedure implemented by M." And on this basis, we claim to have discovered a reason to regard the class $\mathcal{A}$ of algorithms as abstract which is independent of the traditional criteria.

Matters are not entirely this simple, however, because it now generally thought that the fact that a class of objects $\mathcal{C}$ satisfies Dummett's criterion is not sufficient to classify the $\mathcal{C}$'s as abstract. The standard counterexamples are due to Noonan [102] who observed that out that there are classes of $\mathcal{C}$ consisting of intuitively concrete objects are also such that they can only be referred by the use of an expression denoting a function which defined over a distinct class $\mathcal{D}$. This is, for instance, arguably true of concepts volumes and masses of physical substance such as water or gold. For instance, it appears that in order reference

[32] An obvious challenge to the functional criterion from the fact that it also appears possible to refer to specific directions by the use of apparent proper names such as "east" or "north by northwest." But in seems reasonable to suppose that terms of this sort may be likened to mathematical constants in the sense of having been introduced by explicit definition such as "north = the direction of the line running from Greenwich to the north pole." On the basis of this consideration, Dummett modifies the above criterion so as to require, for an object to abstract, that there some functional such that "an understanding of any name of that object involves a recognition that the objects is in the range [of the function denoted by] that functional expression," [29], p. 485.

to a certain spatio-temporally located volume of water (e.g. one liter) or a mass of gold (e.g. one gram), we must respectively employ expressions like "the volume of v" or the "the mass of m" where v and m respectively vary over water filled vessels and gold objects.

Several theorists have attempted to refine Dummett's criterion in order attempt to exclude such objects from the range of the abstract. For our purposes, however, the most important development is that in cases Dummett's criterion appears applicable to a class $\mathcal{C}$, we have reason to think that this class is determined by a sortal concept C – i.e. one for which we can ask "how many?" and, per section 2.1, for which we thus presumably possess a criterion of identity. And in fact something somewhat stronger appears to true of such concepts in that in cases where it seems necessary to pick out the $\mathcal{C}$s via expressions of the $f(t)$ for t denoting a member of $\mathcal{D}$, our means of individuating the $\mathcal{C}$s is usually given in terms of the $\mathcal{D}$s by which they are given.

This situation is again paradigmatically illustrated using the concepts *direction* and *shape*. For note that not only does it seem that if we wish to refer to a direction d we have no choice but specify it as the direction of the line ℓ, but also if we wish to determine whether two directions d_1 and d_2 are the same, are only means of two doing so is by reference of the lines ℓ_1 and ℓ_2 in terms of which they have been given. Similar remarks apply to distinguishing shapes s_1 and s_2 in terms of the figures g_1 and g_2 in terms of they have been given. On their own, of course, these observations do not tell us what the criteria of identity of identity for lines and directions are. But they do, however, tell us that they ought to taken to have the following schematic forms

(2.25) $f(x_1) = f(x_2) \Longleftrightarrow x_1 R x_2$

where f is the functional expression mapping the $\mathcal{D}$ into the $\mathcal{C}$ and R is an equivalence relation defined over the $\mathcal{D}$s.

Such a statements with the form are typically known as an *abstraction principle* for $\mathcal{C}$ and R is referred as its *grounding relation*. Following Hale [55] the idea is if $\mathcal{C}$ is the extension of the sortal concept C, then R is a grounding relation for C just in case for any pair of C-denoting terms $f(x_1)$ and $f(x_2)$, the truth of $x_1 R x_2$ is necessary and sufficient for the truth of $f(x_1) = f(x_2)$. On the basis of this analysis, it is canonically contended that

the relation of parallelism grounds the concept direction in terms of lines and the relation of geometric congruence ground the concept shape in terms of geometric figures.

Much of interest can be said about the plausibility of the claim that a statement of the form (2.25) be formulated for every concept determining a class $\mathfrak{C}$ satisfying Dummett's criterion. For instance, we can ask what sort of conceptual relationship R must bear to C in for a statement having the form of (2.25) to state what we would intuitively take be a genuine criteria of identity for the $\mathfrak{C}$s. This question is closely related to the status of the instances of (2.25) by which Frege [39] wanted to give the identity conditions for numbers and classes. And due in part to variety of logical and conceptual issues which arises in these cases, the general situation with respect the use of such principles to define domain of so-called "logical objects" is quite complex. I will return to consider these issues with respect to the relevant cases at great length in section 3.2.

First, however, I wish to record the standard us of grounding relations to characterize the notion of an abstract sortal. For recall the putative counterexamples to Dummett's criteria were based on the concepts like *V volume of water* and *M mass of gold*. Note that it does indeed seem plausible to say that we can no more refer to a particular volume of water v than as, say, "the water in this bucket" or particular mass of gold m than as, say, "the gold of this ingot" than we can refer to refer a shape than as, say "the shape of this figure." On this basis, we can look at these functional expressions "the water in x" and "the gold of x" as mapping from domains $\mathcal{D}_1$ and $\mathcal{D}_2$ (respectively consisting approximately of vessels and gold objects) into objects falling under V and M. But note that when the grounding relations R_V and R_M for V and M are such that, at any given time, they can only hold of a particular members of $\mathcal{D}_1$ and $\mathcal{D}_2$ and themselves. In others words, unlike the relations $\parallel$ and $\cong$, R_V and R_M are such that they cannot hold of spatially but not temporally separated objects.

This observation led Hale [55] to propose the following basic definition of a so-called *abstract sortal* (which itself derives from an early proposal of [154]):

(2.26) The objects falling under a sortal concept Cs are abstract if and only iff any relation R grounding C is such either

(i) R can hold between things which are spatially, but not temporally separated.

(ii) R cannot hold between spatially located items at all.

On the basis of the foregoing example, the motivation behind (2.26i) should now be apparent. For consider a concept C like direction or shape which is grounded by a relation R which can hold between indisputably concrete objects. It seems reasonable to characterize the fact that the Cs are not located in space in terms of the fact that R can hold between items x_1 and x_2 which are different places at the same time. For in this case there could be a $y \in C$ such that $y = f(x_1)$ and $y = f(x_2)$ meaning that if y were spatially located, it would have to be simulateneously in the location of x_1 and x_2 which is apparently inconsistent with our normal sense of spatial location.

It is important to realize, however, that if we were to formulate (2.26) with only condition (i) that this condition would yield the counter-intuitive result that concepts such as *natural number* and *set* whose members we would paradigmatically classify as abstract would be judged concrete. For at least according to their Fregean or (modified Fregean) analyses, there concepts are grounded by relations (i.e. those of equinumerousity and co-extensivity) which cannot hold between spatially located items at all. The problem in this case is, of course, that the items between which these relations hold (i.e. sets or Fregean classes) are themselves standardly understood as abstract, and hence incapable of spatial location. And thus without clause (ii), these items would ruled as concrete.

This is amendment is relevant because the thesis for which I wish to argue is that *algorithm* ought to be taken to correspond to abstract sortal in much the same way as *direction*, *shape*, *natural number* of *class*. I have already laid much of the ground work involved with showing this on the basis of the foregoing argument that this concept satisfies Dummett's functional criterion (2.23). And if we follow the realist in regarding the concept *algorithm* as having a well defined class $\mathcal{A}$ as its extension, we can also now put several other pieces in place. In particular, on the basis of section 2.4, we can now see that the class $\mathcal{D}$ in terms of which $\mathcal{A}$s are specified must be taken to be some sufficiently comprehensive class of implementations $\mathcal{M}$, possibly formed by taking the disjoint union of a variety of different models of computation. And the function f which maps from the $\mathcal{M}$s to the $\mathcal{A}$s should be taken to be that denoted by function *imp* which I have introduced to schematize the expression "the algorithm implemented by M."

As we will see in the next section, these are already significant finding which the algorithmic realist may potentially put to use in constructing the sort of interpretation of mathematical discourse considered in section 1. However, if we wish to see that concept *algorithm* as naturally falling under Hale's characterization of abstract sortal, what thus remains is to identify an appropriate relation R_M which grounds the class this concept with respect to the class $\mathcal{M}$. And with respect to this question it is possible to note a substantial disanalogy between the concept *algorithm* and the concepts *direction*, *shape*, *natural number* and *class* which are classified as abstract sortal according to (2.26). For note that regardless of the conceptual status of we wish to assign to the corresponding of instance of (2.25) in these cases, the appropriate grounding relation for these concepts is close at hand. For instance, although we may refrain from regarding a statement like Hume's Principle[33]as analytic of the concepts *natural number*, we at least do not have to look very far to find a natural candidate for a relation R which serves the task of individuating number in terms of the cardinalities of the classes which they measure.[34]

Matters stand quite differently with respect to the concept *algorithm*. For although it seems reasonable to take the considerations adduced in section 2 to show that our practices for referring to and reasonable algorithms in terms of implementations are programs are highly conventionalized, it is not entirely obvious by what relation we go about individuating algorithms in terms of the machine which we take to implement them. But as I will begin to discuss in the next section, our general conception of algorithm does indeed rise to some intuitions about how R_M ought to be defined. But as we are about see, there also are substantial matters of logical, conceptual and technical detail which must be taken into account with respect to identifying such a relation.

---

[34]I.e. the statement "the number of Fs = the number of Gs if and only if the Fs and Gs are equinumerous" – cf. [13] and section 4.

2.3 Toward a theory of procedural identity

2.3.1 Varieties of reduction

In section 4 of this chapter I will embark on the project which will also occupy most of Chapter 4 and 5 – i.e. that of attempting to define the equivalence relation R_M which serves to appropriately ground the concept *algorithm* with respect to an appropriately comprehensive class of implementations $\mathcal{M}$. Before getting under way this task, however, it will useful to briefly reconsider how and why this is likely to be of central to the fate of algorithmic realism. The most convenient way of doing this will be to describe several different ways in which the sort of reduction of algorithmic discourse to mathematical discourse described in section 1 might be carried out.

In understanding what it would mean satisfy the realist's desire to reduce discourse about algorithms to discourse about mathematical objects, it will again be useful to draw several parallels between goals of algorithmic realism and those of various nominalist programmes in mathematics. I suggested in section 1 that of the desiderata motivating algorithmic realism is the desire to provide a purely mathematical interpretation of statements such as (2.5) and (2.6) which contain terms which appear to name or quantify over mathematical procedures. The fact that a realist will wish to find such an interpretation of these statements is likely to be due to the fact that such statements in a broad theoretic practice which seeks not only to prove results about the intrinsically computational properties of algorithms but also seeks to relate and justify their application to the derivation of various statements of pure mathematics.

We studied a paradigmatic example of this kind in Chapter 1.4 wherein I considered the use of the algorithm EUCLID to derive mathematical statements of the form $gcd(n, m) = q$. I argued that by using such a procedure we were practically equipped to derive a range of purely mathematical statements such as

(2.27) $gcd(43928, 27149) = 17$

which would be difficult or impossible to derive directly from the definition of the functional expression $gcd(x, y)$. But since (2.27) is a purely mathematical statement (i.e. one that

does contain any procedural terms or quantifiers), one might reasonably wonder why we were justified in using EUCLID to derive it. In particular, we might wonder if in accepting that EUCLID can be use to derive such a statement, whether we have properly expanded the class of proof methods we take to be justifiable means of demonstrating mathematical statements.

One of the concerns of the realist is to justify the use of computational methods in mathematics by showing that this is not the case. And one way he might seek to show this is by showing that use of the use of computational methods in classical mathematics is *conservative* with respect to our original non-computational practices – i.e. that the use of such methods does not allow us to derive purely mathematical statements which we could not (in principle) derive without them. I argued in Chapter 1 that in practice there is no great mystery in how the conservativity of individual algorithms can be demonstrated. For by conventional standards, all that is required in order to justify the application of EUCLID to derive a purely mathematical statement like (2.27), is that we first prove this algorithm to be proven correct via a conventional mathematical methods.

Depending on how EUCLID has been presented, there will be several different ways in which this might be accomplished. For instance, if EUCLID has been presented as a program a formal program – i.e. as "the algorithm expressed by Π" over a programming language L – it can verified by using an axiomatic or denotational semantics for L. And if it has been presented informally as a pseudocode specification, it can be verified in the informal mathematical manner discussed in Chapter 1.4. And finally, if it has been presented as an implementation – i.e. as "the algorithm implemented by M" where M is a mathematically specified machine – then it can be proven correct by arguing about M directly.

Note, however, that in each of these cases it is necessary to construct some sort of mathematical representation of EUCLID which represented its mode of operation in order to prove that it is correct. I argued in Chapter 1 that this appears to be a necessary condition on a correctness proof for an algorithm A with respect to a function f to have the epistemic significance of justifying our use of A to compute the values of f. For note that if a correctness proof for EUCLID did not proceeded by constructing a mathematical object which could plausibly be taken to represent the computational properties which we

conventionally assign to this algorithm, then it would remain mysterious why were entitled to use EUCLID (as opposed to some other algorithm, say NAIVEGCD) to derive a statement like (2.27).

The significance this sort of this sort of observation to the realist's goal of constructing a mathematical interpretation of procedural discourse should not underestimated. For it means that in order for such an interpretation to judged successful, it must not only show the conservativity of computational methods over mathematical ones, but it must also do so in a manner which allows us to correlate individual algorithms with individual mathematical objects which represent the computational properties in terms of which we describe and classify with respect to their practical utility. Note that it is for this latter reason which why the realist presumably will also wish to construct an interpretation which is applicable to "purely procedural" statements such as those occurring in (2.5) and (2.6). For it by statements of these forms by which we standardly report on the complexity theoretic properties of algorithms which we take to bear on their potential applicability for deriving conventional mathematical statements.

We may additionally note that such statements of these sorts are often expressed by explicit quantification over procedures. This is true both in the case of positive statements such as (2.5a,b,c)) which the quantify over algorithms in order to express the existence of an algorithm with certain properties which it feasible for us to compute the values of a given function. And it is also true in case of negative or limitative results like (e.g. (2.6e,f) which quantify over algorithms in order to express that there is no algorithm which we can feasibly use to compute the values of a given function.

Taken together, these observations suggest that a theorist who wishes to explain the utility of computational methods within classical mathematics must ultimately seek to construct what would traditionally be referred to as a *reconstrual* of computational discourse in a purely mathematical idiom.[35]In particular, they must seek to interpret procedural statement in a manner whereby singular terms which, per section 2, appear to convey reference to an algorithm are replaced by terms denoting mathematical objects. And similarly, realist must presumably seek to interpret statements containing quantifiers over algorithms with statements those containing quantifiers over some other suitably delimited mathematical

domain..

This is a significant methodological observation because it allows us to draw a number of systematic parallels to between the aims of algorithmic realism and a various traditional forms of mathematical nominalism. For note that the overarching aim of traditional forms of nominalism has been to show that we "get by" in the empirical sciences without embracing an ontology of mathematical objects which outstrips that required in order to explain the truth of concrete statements. And as we can now see, it is possible to conceive of the algorithmic realist as seeking to demonstrate we can "get by" in classical mathematics without embracing an ontology of procedures which outstrips that required to account for the truth of mathematical statements. But in addition to this, we have may also note that the manner in which most programmes which have attempted to nominalize mathematical discourse have proceeded precisely by providing a systematic reconstrual of statements involving mathematical terms and quantifiers. And as such, it is possible also possible for the algorithmic realist to attempt to proceed in the manner of one the traditional strategies by which mathematical nominalists have attempted to reduce discourse about mathematical objects to discourse about concreta.

By far the most common case strategy of this sort is what Burgess and Rosen refer to as *Tarskian reduction* (after Tarski, Mostowski and Robinson [141]). This approach is nothing other than the familiar technique of interpreting the language of one mathematical T_1 into that of another T_2 so that terms and quantifiers which purport to refer to the items which T_1 talks about (say the real numbers) are translated into terms and quantifiers which purport to refer to the items which T_2 talks about (say sets of natural numbers). In the case of traditional nominalistic reductions, T_1 is taken to correspond to a theory T_e with a mixed empirical-mathematical vocabulary (say that of classical mechanics) and that T_2 is taken to correspond to the subtheory T_c of T_e with terms and quantifiers referring only to concrete objects. Suppose that these theories are respectively stated over the languages $\mathcal{L}_e$ and $\mathcal{L}_e$. In this setting, the mathematical nominalist generally perceives his task of being

[35]My use of the term "reconstrual" to describe a means of interpreting one language $\mathcal{L}_1$ in another $\mathcal{L}_2$ in a manner such that the terms and quantifier of $\mathcal{L}_2$ are translated into terms of the same grammatical category as well as much of the other terminology employed in this section from derives Burgess and Rosen [16].

that of showing that T_e is conservative over T_c for $\mathfrak{L}_c$ sentences – i.e. for all $\mathfrak{L}_c$ sentences φ, if $T_e \vdash \varphi$, then $T_c \vdash \varphi$.

Based on the foregoing discussion, it seems reasonable to liken the use role which the algorithmic realist takes computational methods to play in mathematics to that which mathematical nominalist takes mathematical method to have in the empirical sciences. To make this analogy precise in a manner that would allow us to formally demonstrate the sort of conservation results discussed above, the algorithmic realist might thus contemplate proceeding as follows: 1) formulate a procedural-cum-mathematical theory T_p in which both specific results about mathematical statements (2.27) and also general results about algorithms like (2.5) and (2.6) could be demonstrated; 2) prove that T_p was conservative over its mathematical subtheory T_m for purely mathematical statements.

It is, of course, a massive idealization to think that we currently possess a theory T_p which simultaneously provides an axiomatic formulation of the methods of various disparate subfields of theoretical computer science by which statements like (2.5) and (2.6) are demonstrate. But it also seems that we can imagine what such a theory might look like with sufficient precision to at least aid us in formulating the theoretical goals of algorithmic realism. For as noted above, we would ideally like to able view statements a statement such as (2.5) as having the subject-predicate form $K(a)$ where a is a name for an algorithm (e.g. MERGESORT) and K is a possibly complex procedural predicate (e.g. "has running time $O(n \log(n))$"). And similarly, we would like to able to view the statements of (2.6) as having the respective forms $\exists X[K(X) \wedge \ldots]$ and $\forall X[K(X) \rightarrow \ldots]$ where X is a variable over algorithms.

On my argument in section 2 that mechanisms of procedural reference are highly conventionalized, it thus seems possible to have a rough description of the language $\mathfrak{L}_p$ in which T_p ought to be framed. In particular, we can assume that $\mathfrak{L}_P$ is a two-sorted language containing a purely mathematical part $\mathfrak{L}_m$ and a mixed procedural-mathematical part $\mathfrak{L}_p$. In particular, I will assume that $\mathfrak{L}_p$ will contain variables $X_1, X_2, \ldots$ over algorithms as well as constants $a_1, a_2, \ldots$ corresponding to common names of the sort consider in 2.1 as well standard mathematical variables $x_1, x_2, \ldots$ and terms $t_1, t_2, \ldots$. Moreover,

this language will also have to contain predicate symbols corresponding to common properties $K_1, K_2, \ldots$ which are standardly predicated directly of procedures. Paradigmatic examples of such predicates will include $Time_{O(f(|x|))}(a)$ and $Space_{O(f(|x|))}(a)$ least common complexities bounds (e.g. $f(n) = c, \log(n), n^k, 2^k$ etc.). These are meant to formalize the properties expressed by "algorithm A has running time complexity $O(f(|x|))$" and "algorithm A has space complexity $O(f(|x|))$" for at least common complexities bounds (e.g. $f(n) = c, ln(n), n^k, 2^k$ etc.) which are the primary topics of concerns in algorithmic analysis and complexity theory. This will mean that a statement like "MERGESORT has running time $O(n \log(n))$" will be formalized in $\mathcal{L}_p$ as a statement of the form $K(a)$ and a statement like "There is a polynomial time primality algorithm" will be formalized in $\mathcal{L}_p$ as $\exists X[K_1(X) \wedge K_2(X)]$.[36]

Of equal significance, $\mathcal{L}_p$ will also have to contain a set of relations which serve to relate procedural entities with mathematical ones. Such terms will be required in order to frame the results of the sort of informal reasoning about procedures which T_p is intended to formalize. In this regard, there are at least two significant classes of statements which we would like to formalize which span the gap between procedural entities and mathematical ones: 1) those expressing that a particular mathematical object (as per section 2, canonically an implementation) is used to introduce a name for a procedure; 2) those expressing either that the result of applying a procedure to a given mathematical argument yields another mathematical value or, more generally, that a procedure is correct with respect to a given mathematical function;

Pursuant to the discussion in section 2, I will assume that all instances of the former sort are informally expressed by statements of the form "$A =_{df}$ the algorithm implemented by M." These constructions can be mimicked in T_p by including in the $\mathcal{L}_p$ functional expressions $imp(m)$ meant to formalize the schema (2.8b) where I will now assume m is a (possible complex) mathematical term over $\mathcal{L}_m$ which denotes an implementation. The intended interpretation of imp will thus be a function mapping mathematical entities of

[36]The intention in the latter case is that K_1 correspond to the predicate "polynomial time algorithm" and K_2 to the predicate "primality algorithm." As mentioned above, there are well known mathematical analyses of these notions. While it thus the possible to formalize these properties as complex $\mathcal{L}_p$ predicates, but for simplicity I will not bother to do so here.

the appropriate types into objects in the domain of the algorithmic quantifiers.[37]In order accommodate mixed statements of the second form, I will also assume that $\mathcal{L}_p$ contains a formal term meant to represent an application functional. This term will have the form of a functional expression $Exec(A, x)$ whose first argument is in the algorithmic domain and whose second is in the mathematical domain and which returns values in the mathematical domain. The intended interpretation of sentence of the form $Exec(A, x) = y$ will be that the result of executing the procedure denoted by A to the arguments denoted by x yields the value y.

The necessity for including mixed mathematical-procedural terms in $\mathcal{L}_p$ also puts substantial demands on the mathematical language $\mathcal{L}_m$ which must be employed to talk about such entities in T_p. For not only will language need to be sufficiently rich to define the structure of individual implementations, but we will also need to be able to define an application functional $App_{\mathcal{M}}(m, x)$ for various models of computation $\mathcal{M}$. In short, this means that $\mathcal{L}_m$ must be sufficiently rich to describe not only the mathematical objects on which the algorithms we want to formalize operate, but also the combinatorial structure of various forms of implementations (i.e. Turing machines, RAM machines, etc.). This $\mathcal{L}_m$ must thus either contain primitive terms denoting a great variety of mathematical structures, or, more plausibly, be taken to contain the a language like that of first-order set in which such structures can be defined. And correspondingly, T_m must be rich enough to prove that the definitions we give over $\mathcal{L}_m$ have the appropriate properties – e.g. that the various structured objects exist, that inductively defined define terms have unique extensions, etc.. This in turn suggests that T_m should be taken to contain (or interpret) at least a fraction of the first-order ZF axioms.

The sketch of T_p just given is admittedly schematic and it would require substantially

[37]If we T_p were going to develop in more detail, it would also be desirable that we develop some means of accounting for procedural reference via (2.8b). One way this could be accomplished would be to include a functional expression *prog* as a $\mathcal{L}_p$ term in parallel to *imp*. Note, however, that pursuant to the conclusion elicited in section 2.4, such as an expression should not be treated as a primitive expression but rather defined in terms of an appropriate family of interpretation functions op_L. The necessity of including such terms into $\mathcal{L}_p$ would require that we develop the syntax and operational semantics of various programming languages in T_p. There is no reason to think that this requirement could not be accommodated if we assumed that $\mathcal{L}_m$ was sufficiently expressive and T_m sufficiently powerful. But since this added layer of complexity need to accomodate the possibility of indirect reference to algorithms via program within T_p will not effect the basic point I will make below, I will suppress these details in the sequel.

more effort to formally define such a theory in detail. However, the framework presented thus far is just strong enough to describe the relevant set of analogies between, on the one hand, the procedural and mathematical theories T_p and T_m, and on the other, the empirical and mathematical theories T_e and T_c. In particular, the algorithmic realists claim that the use of computational methods in mathematics can be eliminated in favor of purely mathematical ones, can be taken to be equivalent to the claim that T_p is conservative of T_m for purely mathematical statements – i.e. for all $\mathcal{L}_m$-sentences φ, if $T_p \vdash \varphi$, then $T_m \vdash \psi$. This may be compared to the mathematical nominalists claim that the use of mathematical methods in empirical sciences can be eliminated in favor of purely concrete ones.

The question to which we may now address ourselves is how the algorithmic realist might go about proving such a result based on what we know both about T_p and also about the other explanatory pressures which he is under. One option is to construct an interpretation of the language $\mathcal{L}_p$ in the language $\mathcal{L}_m$. This would correspond to a mapping of terms and sentences $(\cdot)^* : \mathcal{L}_p \to \mathcal{L}_m$ with the following properties:

(2.28)
i) For all primitive $\mathcal{L}_m$-terms t and $\mathcal{L}_m$-predicates P, $t^* = t$, and $P^* = P$.

ii) For all primitive $\mathcal{L}_p$-terms a and $\mathcal{L}_p$-predicates K, a^* is some (possibly complex) $\mathcal{L}_m$-term and K^* is some (possibly complex) $\mathcal{L}_m$.

iii) $s(t_1, \ldots, t_n)^* = s^*(t_1^*, \ldots, t_n^*)$, $P(t_1, \ldots, t_n)^* = P^*(t_1^*, \ldots, t_n^*)$.

iv) If $\varphi \equiv \psi \circ \chi$, then $\varphi^* \equiv \varphi^* \circ \psi^*$ where $\circ = \wedge, \vee, \to$ and $(\neg\varphi)^* \equiv \neg\varphi^*$.

v) If $\varphi \equiv \forall x\psi$, then $\varphi^* \equiv \forall x\psi^*$.

vi) If $\varphi \equiv \forall X\psi$, then $\varphi^* \equiv \forall x[A(x) \to \psi^*(x)]$ for some $\mathcal{L}_m$-predicate A.

One of the significant clauses in this definition is ii), which tells us that procedural terms and predicates are to be interpreted as mathematical terms and predicates. The other is vi), which tells us that quantification of algorithms in $\mathcal{L}_p$ is to interpreted as restricted quantification over mathematical objects in $\mathcal{L}_m$ with the mathematical predicate $A(x)$ as a "guard." In particular, this clause tells us that only those objects satisfying the predicate $A(x)$ are to be treated as the mathematical correlates of algorithms.

Upon constructing an interpretation of $\mathfrak{L}_m$, the realist could then attempt to show that for all axioms φ of T_p, $T_m \vdash \varphi^*$. And from this it follows by an immediate induction on derivations that for all $\mathfrak{L}_p$ sentences φ, if $T_p \vdash \varphi$ then $T_m \vdash \varphi^*$. This would immediately demonstrate the conservativity result in the strong sense if $T_p \vdash \varphi$ for some purely mathematical statement φ then not only do we have $T_p \vdash \varphi$ (for in this case $\varphi^* \equiv \varphi$), but also that any computational premises $\psi_1, \ldots, \psi_n$ occurring in the original T_p proof can be simply eliminated in favor of the derivable mathematical statements $\psi^*, \ldots, \psi^*$.

This means of showing a conservativity result also has the beneficial consequence of providing a reconstrual of $\mathfrak{L}_p$ in terms of $\mathfrak{L}_m$. For note that under the mapping $(\cdot)^*$, $\mathfrak{L}_p$-terms will be systematically correlated with $\mathfrak{L}_m$-terms in a manner so that terms denoting algorithms will be systematically correlated with terms denoting mathematical objects, and procedural quantifiers will be correlated with mathematical quantifiers. So suppose for instance, that a is a $\mathfrak{L}_p$ denoting EUCLID and that T_p proves (2.27) via a derivation containing the statement the statement $\forall n \forall m[Exec(a, \langle n, m \rangle) = gcd(n, m)]$ and $Exec(a, \langle 43928, 27149 \rangle) = 17$. If were we able to prove the conservativity of T_p over T_m in manner described in the previous paragraph, not only would be able to conclude that T_m proved (2.27), but we would also know that T_m was able to "internalize" the correctness proof by proving a statement of the form $\forall n \forall m[Exec^*(a^*, \langle n, m \rangle) = gcd(n, m)]$ where $Exec^*$ and a^* where is a mathematical translation of a and $Exec^*$ a mathematical translation of $Exec^*$. By using these expressions we would thus be able to mimic a calculations carried out by EUCLID within T_m.

The central question with which the algorithmic realist must thus seek to answer is whether it is indeed possible to construct such an interpretation so as to prove the required conservativity theorem. Ultimately I believe this question must be answered in the negative. This is not, however, because T_m turns out too weak to prove that any specific algorithm is correct in a manner which would allow us to eliminate reference to it in the proofs of a statements like (2.27).[38] And it is not the related to any of the traditional reasons why specific nominalistic programmes in mathematics are often taken to be either question

beginning or fail outright for plausible choices of T_e and T_c.[39]

Rather, the more general worry which I think the realist must face is that theory T_p may turn out to be*inconsistent*.[40] In order to see the basis for both sorts of worries, it will be helpful to temporarily abandon the proof theoretic perspective on T_p we have been employing to thus favor of a model theoretic one. For to ask for ask whether T_p is consistent is precisely to ask whether there exist exists an $\mathfrak{L}_p$ structure in which it is satisfied. Note that a model of T_p will be of the form $\mathfrak{P} = \langle \mathcal{A}, \mathcal{N}, I \rangle$. Here $\mathcal{A}$ and $\mathcal{N}$ will correspond to non-empty sets which are respectively intended to serve as domains of procedural and mathematical objects and I is an interpretation function assigning arbitrary $\mathfrak{L}_p$ terms and predicates extensions in these two sets of the appropriate type. For instance, if K is a unary procedural predicate (e.g. "has running time $n\log(n)$"), then K^I will be a subset of $\mathcal{A}$, if t is the term 0, then t^I will be a member of $\mathcal{N}$, etc.

In order to get a better impression of there can be such a structure satisfying T_p, we must next consider the sorts of statements axioms it is likely to contain in more detail. As I have noted, the situation which we stand in with respect to specifying T_p is unlike that in

---

[39]The primary hurdle which must be cleared by such theories is often that of showing that showing that T_c is sufficiently strong to guarantee the existences of sufficiently concrete objects to serve as representatives of the mathematical items in the intended domain of the mathematical subtheory T_n of T_e. This may occur, for instance, when T_n is or contains a fraction of the theory of second-order arithmetic or real analysis and as such implies the existence of uncountably many things. The need to accomodate this sort of eventuality has traditionally lead nominalists such as Field [34] and Hellman [60] to enrich the intended domain of T_c to include various forms of geometric or intensional entities which are likely to have been spurned by traditionally minded nominalists such as Quine and Goodman [112].

[39]Extreme care must be taken in order to make this generalization precise. For note that almost certainly is the case that we can refer to and reason about algorithms A which it point of fact do determine total functions f (say of type $\mathbb{N} \to \mathbb{N}$) but which we cannot *prove* to be correct with respect to f. This observation follows from a classical result in computability theory that the class of Σ_1-definable functions f which are provably total over a recursively axiomatized arithmetic theory Z (e.g. PA) – i.e. for which $Z \vdash \forall x \exists! y \exists z \theta(x,y,z)$ where $f(x,y)$ has been defined as $f(x) = y \leftrightarrow \exists z \theta(x,y)$ – is a proper subset of the functions which are actually total – i.e. for which $N \models \forall x \exists! y \varphi(x,y)$. This result can be generalized some what to show that the function computed by f can be any partial recursive function on $\mathbb{N}$. This means, for instance, that there will be algorithms A which compute the function $gcd(x,y)$ but which cannot be proven correct with respect to its definition. If we were to employ such an algorithm to derive (2.27) in T_p, then this result might be taken to show that T_p does indeed outstrip T_m with respect to derive mathematical statements. The question which realist must face is whether in this sort of case – i.e. one which an algorithm A has been used to derive a statement of the form $f(\overline{a}) = b$ of an actual calculation of A with input $\overline{a}$ and output b and unproven claim that A is correct with respect to f, the statement in question should be taken to be part of T_p itself. Although I will not take the time to provide a complete analysis of this situation, it seems that the realist is entitle to respond in the negative for the simple reason that without a proof of the correctness claim, we have no way of relating the outputs of A with the values of f.

[40]Note that in the case that T_p is inconsistent, it is trivially not conservative over T_m as long as T_m is itself consistent. For in this case, $T_p \vdash \bot$ but $T_m \nvdash \bot$.

which we stand with respect to specifying the mathematical T_n component of an empirical theory T_e. For while in standard cases, it will be reasonably clear what mathematical principles must be embodied by T_n and also for axiomatize them over a standard first- or second-order mathematical signature. But in the case of T_p we do not start out with anything like a full axiomatization or even a completely clear delineation of the signature corresponding to $\mathcal{L}_m$. What we know about the form which this language and theory should take is gleaned from piecemeal observations about the role of algorithms which algorithms play in different subfields in computer science whose interrelationships themselves may not entirely worked out.

One way in which this problem may be canonically illustrated is by asking after which $\mathcal{L}_m$ statements ought to be regarded as axioms of T_p and which as theorem. Certainly, for instance, our practices suggest that we take the statement in (2.6) as expressing true statements about algorithms. But as I have noted above, many of these statements are non-obvious in the sense that they admit to non-trivial proofs. And certainly this is good evidence that these statements should not themselves be counted as axioms. But at the same time, it also seems doubtful that we are currently in a position where we can provide a plausible positive criteria for judging a statement about an algorithm to be sufficiently fundamental to treated as an axiom.[41]

However, there do appear to be two categories of statements involving algorithmic terms which appear to admit no non-trivial analysis into more basic mathematical or procedural components. These correspond to statements of the form $a = imp(m)$ and $a \neq imp(m)$ which respectively report that the algorithm named by a is or is not implemented by the machine denoted by m. We will have to wait until the next chapter to see in detail how statements of this figure in our computational practices. It is, however, useful to point

[41]This is evident, for instance, by examining the statements in (2.5). Again we certainly do view these statements as expressing propositions about algorithms. And above I have even argued that these statements are logically of subject-predicate form. But note, however, that it is also possible to provide a further mathematical-cum-procedural analysis of each the predicates in question. For instance the property of being a comparison sort (which is predicated of the algorithm INSERTIONSORT in (2.5c)) can be further analyzed in terms of the operations which a sorting algorithm uses in the course of its execution. Thus not only should this property presumably be defined via a complex $\mathcal{L}_p$ predicate $\Phi(X)$, but since this is presumably a property which can be *proven* to apply to an algorithm based on its structure, there seems to be no basis for taking this statement as axiomatic.

out in advance that statements of these sorts will be quite common in the practice of computer science wherein we routinely construct implementations of individual algorithms using common models of computation of the sort mentioned above. Recall, however, that I argued in section 2.3.2 that we appear to be incapable of giving a explicit definition for the functional expression "the algorithm implemented by m" which the expression imp (which I will now regard as a $\mathcal{L}_p$ functional expression) is intended to formalize. As such, it seems reasonable to conclude such statements must be taken as primitive judgments about the relationship between algorithms and implementations. And this in turn suggests that should be among the axioms of T_p.

Now consider the class of $\mathcal{L}_p$ statements Γ containing all equalities of the form $a = imp(m)$ and inequalities of the form $a \neq imp(m)$ that are so reflected by our practices. Although we have yet to see concrete examples of intuitively plausible cases our intuitions which either a given algorithm either is or is not implemented by a given machine, I will argue in Chapter 3 that cases of this sort abound in computational practice. And in fact it seems reasonable to assume that every algorithmic name of the sort considered section 2.2 is introduced by at least one statement of the form $a = imp(m)$ which I will assume is contained in Γ. Since I just argued that such statements ought to regarded as axioms of T_p, Γ will be contained in this theory. And thus in order to show that T_p is consist, an algorithmic realist will have to show that Γ is itself consistent. But in order to do this, it seems he will have to construct $\mathcal{L}_m$-structure in which all the sentences in Γ are satisfied. And as I now which to suggest, there is indeed some room to doubt whether this task can be can accomplished.

The first point to note in this regard is that since imp is intended to denote a function mapping implementations into algorithms, its interpretation in any $\mathcal{L}_m$-structure $\mathfrak{P} = \langle \mathcal{A}, \mathcal{N}, I \rangle$ must be of type $imp^I : \mathcal{N} \to \mathcal{A}$ – i.e. a function from mathematical objects to procedural ones. But of course not our intention for imp to defined on all mathematical objects in $\mathcal{N}$, but only those which are properly regarded as implementations. It is, however, reasonable to think that these structures will form subclass $\mathcal{M}^I \subseteq \mathcal{N}$ which is explicitly definable by an $\mathcal{L}_p$ open sentences $Impl(x)$. And thus without loss of generality, we may assume that for any $\mathcal{L}_p$ interpretation $\mathfrak{P}$, imp can be taken to have the type $imp^I : \mathcal{M}^I \to$

$\mathcal{A}$.[42]

The next point to note is that the specific statements in Γ do not provide an explicit definition of *imp*, but merely give its values on those implementations $m_1, m_2, \ldots$ such that the sentence $a_j = imp(m_i)$ appears in Γ. If we are seeking to construct the structure of $\mathfrak{P}$ based solely of what we know about T_p, we can thus not assume that this theory is sufficiently strong to constraint the definition of $\mathcal{A}$ or of I uniquely. It thus appears that in seeking to show that Γ is consistent, the realist will have little choice but to adopt the strategy proposed in section 2.3.2 and regard the statements in Γ as points of data which constrain a potential implicit definition of imp^I and $\mathcal{A}$.

In asking whether we can construct a structure $\mathfrak{P}$ satisfying T_p it is reasonable to concede to the algorithmic realist that there T_p will uniquely determine an interpretation $\mathfrak{N} = \langle \mathcal{N}, I_m \rangle$ of the mathematical theory $T_m \subseteq T_p$. This in turn means that, in $\mathcal{N}$, he will be able to identify a determinate class of mathematical objects $Impl(x)^{I_m}$ as corresponding to the domain $\mathcal{M}$ of *imp* in $\mathfrak{P}$. The task of showing that T_p is consistent thus reduces to that of showing that relative to $\mathfrak{N}$ and $\mathcal{M}$ we will be able to define $\mathcal{A}$ and I extending I_m such that $\mathfrak{P} = \langle \mathcal{A}, \mathcal{N}, I \rangle$ is a model of T_p.

Given these concessions, it may still not be entirely clear how the realist ought to do about defining the class $\mathcal{A}$ and I. But it is precisely at this point where the conclusions of section 3.1 may be brought to bear. To recapitulate, these were as follow: 1) the that extension of the concept *algorithm* must be given as the range of values of the functional expression "the algorithm implemented by M"; and 2) the identity conditions for objects in this class are given by equivalence relation R_M over extension of the concept *implementation*. The first observations can be readily parlayed our current framework by noting first that in an $\mathfrak{L}_p$-structure $\mathfrak{P}$, the extension of *algorithm* out to be taken to correspond to the procedural domain $\mathcal{A}$. The second observation can also be accommodated by showing how the relation R_M can be defined using over $\mathfrak{L}_m$ as a predicate $\beta(x, y)$ which may then shown

[42]This can most readily be accomplished by taking T_p to include the axiom $\forall x \exists X[imp(x) = X \wedge \neg Impl(x) \rightarrow X = A_\uparrow]$ where $A_\uparrow$ is interpreted as an arbitrary or undefined algorithm. It is, of course, also somewhat unclear how the predicate $Impl(x)$ ought to be defined so as to uniformly characterize the class of mathematical structures which can serve as implementations. I will examine this question in more detail in Chapter 4. For the time being, however, it will is safe to assume this predicate is defined as a long disjunction of structure descriptions of common models of computation.

to be an equivalence relation over the class $\mathcal{M}$ by reasoning in T_p.

But in taking this steps, the realist faces the initial challenge of demonstrating that a given definition of $\beta(x, y)$ bears the appropriate relation to our background concept *algorithm* so as that an appropriately instance of (2.25) can taken to serve as a definition of identity for members of the class $\mathcal{A}$ which he hopes to define. If such a case can in fact be made then the statement

(2.29) $imp(m_1) = imp(m_2) \Longleftrightarrow \beta(m_1, m_2)$

can be adjoined as an axiom to T_p obtain a theory T_p^+. Since $\beta(x, y)$ will be defined uniformly over $\mathcal{M}$, the realist may then attempt to prove that T_p^+ (and thus T_p) is consistent by using (2.29) to induce explicit definitions of imp^I and $\mathcal{A}$ as follows: i) he can take $imp^I(m)$ to be the equivalence class $[m]$ of implementations which bear β to m^I in $\mathfrak{N}$ (i.e. set $[m^I] = \{m' \, : \, \mathcal{N} \models \beta(m, m')\}$); and ii) he can take $\mathcal{A}$ to be the set of all such equivalence classes for $m^I \in \mathcal{M}$.

Several point can be noted about the proposal. The first of these is the obvious observation that if it is indeed successful (i.e that the mapping imp^I can be shown to exist and $\beta(x, y)$ shown to have the appropriate properties) then it will follow that the construction just outlined will suffice to demonstrate the realist's central claim that algorithms are mathematical objects. In particular, since all elements of $\mathcal{A}$ will be given as the values of the form $imp^I(m)$ are here taken to simply correspond to equivalence classes $[m]$, it seems safe to conclude that the members of this set are indeed bona fide mathematical objects.[43]There is, however, a substantial question of detail her as with respect to whether these classes can additionally be taken to members of the mathematical domain $\mathcal{N}$. For note that the statement (2.29) bears at least a superficial resemblance to other so-called *abstraction principles* whose adjunction to seeming mundane background theories can be shown to lead to inconsistency (the most famous, of course, being Frege's [38] Basic Law V). One route by which it might be shown that the realists desire to assimilate algorithms to mathematical objects is that the sort of abstraction principle which appears to be required to do so may lead to a form of logical or set-theoretic contradiction.

[43]What is on offer to a realist via (2.29) is a form of what Burgess and Rosen refer to as *contextual*

In the next section, I will attempt to show that if he sufficiently careful in giving the definition of $Impl(x)$, the realist will not have to worry inconsistencies introduced in this manner. But what I believe does turn out the be a much significantly more worry is the possibility that inconsistencies will arise due to a clash between the class of identities which will be induced by (2.29) given a particular definition of $\beta(x, y)$ and those which are already codified in Γ. For note that as matter currently stand, it is open for a realist to nominate *any* definable equivalence relation over $\mathcal{M}$ to this role. This gives rise to worry that for certain definitions of $\beta(x, y)$ the following situation will arise: 1) for some terms $\mathcal{L}_m$ terms m_1, m_2 denoting implementations, $T_p \vdash \beta(m_1, m_2)$ and thus via (2.29) $T_p^+ \vdash imp(m_1) = imp(m_2)$; but 2) the statements $a = imp(m_1)$ and $a \neq imp(m_2)$ are both in Γ. If this were to arise for rise for given definition of $\beta(x, y)$, then T_p^+ will obviously turn out to be inconsistent. And the realist will thereby be deprived of the option of defining $\mathcal{A}$ and imp^I in the manner suggested above as the resulting structure will not satisfy Γ.

Such a situation will correspond to one in which the identity conditions for algorithms which are induced by a given conceptually motivated definition of algorithmic identity diverge from the constraints imposed on such a definition by our case-by-case intuitions about which algorithms are implemented by which machines. On the basis of what I have said thus far, we have no reason to suspect that this is likely to be a cause for concern. For in particular, it may turn out the realist is able to define $\beta(x, y)$ so that it uniformly satisfies all of the statements in Γ. It will be the burden of Chapters 3 and 5 to argue that a variety of details conspire to make such a clash inescapable. But before turning to the negative considerations which show why this is so, it will useful to end this chapter by considering the conceptual desiderata which constrain the proper definition of $\beta(x, y)$.

---

reduction – i.e. a means of reinterpreting algorithmic discourse in a manner so that the truth conditions of identity statements of the form $imp(m_1) = imp(m_2)$ which does *ipso facto* fix the denotation of the terms it equates. It is this by this mode of reduction which neo-Fregean philosophers of mathematics (e.g. Wright [154]) have sought to interpret number theoretic discourse by using Hume's principle to give truth conditions for statements of the form "the number of Fs = the number of Gs." The additional proposal outline in the preceeding paragraph that relative to a background mathematical structure $\mathfrak{N}$ which fixed the domain of $\mathcal{M}$, terms of the form $imp(m)$ can be explicitly defined as equivalence classes of implementation under the relation β^{I_m} can be likened to Frege's own proposal that the denotation of "the numbers of Fs" can be explicitly defined as the classes of (Fregean) concepts equinumerous with F. This correspond to what Burgess and Rosen call *objectual reduction*. I will discuss distinction between these two options and their relationship to algorithmic realism at greater length in the next section.

2.3.2 Identity and bisimulation

The purpose of this section is to outline in broad terms the task of providing a satisfactory definition of algorithmic identity. In the previous section we have seen why the framing of such a definition will be central to the defense of algorithmic realism. And in the following chapters I will explore the exigencies involved with providing a satisfactory definition of this sort in considerable depth. But before getting under way in earnest, it will be useful to attempt to tie together the largely conceptual considerations which have driven the account provided in this chapter with the more technical ones which will be at play in Chapters 3, 4 and 5.

Recall that I argued in Section 1 that if an algorithmic realist wishes to maintain that individual algorithms are mathematical objects, he will presumably be charged with providing a theory which determines the truth values of all statements of the form $t_1 = t_2$ and $t_1 \neq t_2$ where t_1 and t_2 are drawn from class of all terms τ_p which we take to denote algorithms. But on the basis of Section 2, we can now say quite a bit both about the form which the terms in τ_p must take and also about how they should be introduced to our computational-cum-mathematical language $\mathcal{L}_p$. For on the one hand, we now know that these terms must correspond either to algorithmic "names" (i.e. terms like EUCLID or MERGESORT which correspond to $\mathcal{L}_p$ constants $a_1, a_2, \ldots$) or functional expressions of the form "the algorithm implemented by M" (which we are now assuming to be formalizable in $\mathcal{L}_p$ as terms of the form $imp(m)$). And on the other, we also know that terms of the former sort must be introduced by explicit definitions of the form "A is the algorithm implemented by M" (which we would now formalize in $\mathcal{L}_p$ as $a = imp(m)$).

On the basis of Section 3, we also know quite a bit about how we ought to go about devising a theory which would fix the truth values of the statements in question given these assumptions about how reference to algorithms is effected. The first step in providing such a theory is to identify the class of sentences Γ which will state various equality and inequality statements between terms of the form a and $imp(m)$. There will generally be at least one recognized implementation M by which we make reference to each algorithm A for which we possess a name, it is also reasonable to suppose that Γ will contain at least

one sentence of the form $a = imp(m)$ for each procedural constant of $\mathfrak{L}_p$.

There are, however, also substantial limitations on how far we can hope to use Γ to constrain the details of a definition of algorithmic identity. For, on the one hand, there will often be more than one implementation by which we can denote A, meaning that Γ will contain groups of the sentences of the form $a = imp(m_1), a = imp(m_2), \ldots$ for distinct terms $m_1, m_2, \ldots$ abbreviating descriptions of implementations. (Note in particular this means that Γ will entail statements of the form $imp(m_1) = imp(m_2)$ which, as we saw above, may come into conflict with those induced by a general definition of algorithmic identity.) But at the same time, we since we cannot always expect there to be some algorithm A which we take to be that implemented by each implementation M, we cannot always expect Γ to contain a sentence of the form $a = imp(m)$ for every term m denoting an implementation. And thus we cannot expect Γ to decide the truth value of every equality between τ_p terms. There is thus substantial work left for an independently motivated theory of algorithmic identity to do.

In Section 3.1 I describe how such an account could be developed in the context of the axiomatic theory T_p which is intended to uniformly formalize the theoretical practices of the various subfields of computer science in which we reason about algorithms directly. But even the most committed realist must acknowledge that we are currently able to gesture at what the details of such a theory might look like. And I have correspondingly conceded to the realist that we may assume that the mathematical component T_m of T_p has a determinate interpretation $\mathfrak{N}$ in which $\mathfrak{L}_m$-terms are all interpreted. Putting these two observations together means that, without loss of generality, I can return to my original practice of referring to implementations directly – i.e. via expressions $M_1, M_2, \ldots$ which I take to denote objects in the domain of $\mathfrak{N}$, as opposed to referring to them via expressions $m_1, m_2, \ldots$ which should be officially regarded as complex definite descriptions over $\mathfrak{L}_m$.

In seeking to determine a definition of algorithmic identity which determines a truth value for every identity statement between τ_p terms we may also return to the setting of Section 2.5 and speak directly of equivalence relations defined over an appropriately defined class of implementations $\mathcal{M}$. In concrete terms this means that the realist will be responsible for providing both a principled mathematical definition of this class together

with a definition of an equivalence relation over $\mathcal{M}$ which I will subsequently denote by $\underline{\leftrightarrow}$. He can then attempt to adopt the following statement as part of an attempt to simultaneously define the class $\mathcal{A}$ and the definition of the function $imp : \mathcal{M} \rightarrow \mathcal{A}$:

(2.30) $\forall M_1 \forall M_2 [imp(M_1) = imp(M_2) \Longleftrightarrow M_1 \underline{\leftrightarrow} M_2]$

For reasons which will emerge below, I will refer to the relation $\underline{\leftrightarrow}$ as a *bisimulation* relation and (2.30) as the *bisimulation principle* [(BP)].

To meet the minimal requirement of showing that algorithmic terms can be assigned reference in a mathematical domain in a manner which preserves the truth value of $\mathcal{L}_p$ statements, the realist would thus have to show the following:

(2.31) i) the resulting class of identities of the form $imp(M_1) = imp(M_2)$ is consistent with Γ;

ii) $\mathcal{M}$ is sufficiently broad to contain all forms of implementations by which we refer to algorithms in practice;

iii) the relation $\underline{\leftrightarrow}$ is an adequate grounding relation for the concept *algorithm* with respect to $\mathcal{M}$.

We have already seen why it is important that the first of these requirements is satisfied. The second requirement is significant because it ensures that manner in which the realist proposes to define $\mathcal{M}$ is sufficiently broad so as to cover all instances in which we appear to make reference to algorithms via (2.8b). I will defer a thorough discussion of the technical issues which arise with respect to how requirements (2.31i,ii) constrain the definition of $\underline{\leftrightarrow}$ until Chapter 3, at which point we will have a somewhat better handle on the notion of implementation in general.

For the time being, however, I will concentrate on requirement (2.31iii). In this regard, it will first be necessary to expand slightly on my prior discussion of grounding relations for sortal concepts. For instance, consider again the abstraction principles for directions which was discussed in Section 2.5 above. If we let $dir(\cdot)$ denote the function expressed by "the direction of $\cdot$" and $\parallel$ denote the relation of parallelism between lines, the corresponding instance of (2.29) for the sortal *direction* now takes the form

(2.32) $\forall \ell_1 \forall \ell_2 [dir(\ell_1) = dir(\ell_2) \Longleftrightarrow \ell_1 \parallel \ell_2]$

According to the terminology adopted above, (2.32) is an abstraction principle for the sortal *direction* as grounded by the relation of parallelism with respect to the sortal *line*. As I mentioned above, such a statement can be thought of as giving the identity conditions of directions in terms of the geometric relations which hold among the lines by which they are give (via $dir(\cdot)$). But it can also be seen as a method for delimiting a domain of abstract objects relative to a domain of concrete ones – namely those things which fall in the range of the function denoted by $dir(\ell)$ when it is applied to a concrete line ℓ.

The question to which we must now address ourselves is that of qualifying the relationship between the relation of parallelism has to bear to that of the concept *direction* in order for (2.32) to play these respective roles. Following Hale, I required above that in the case all items falling under the sortal C must be referred to as $f(x)$ for x denoting an object falling under the sortal D, then R is a grounding relation for C just in case the truth of $x_1 R x_2$ is necessary and sufficient for the truth of $f(x_1) = f(x_2)$. If all we require of a principle like (2.32) is that it provide a means of individuating the directions whose extensions coincide with our pre-theoretical judgments of sameness and difference of directions, than this is presumably a sufficient condition on a grounding relation.

Note, however, that relative to this understanding of the role of (2.32), a large variety of other relations will also serve to ground *direction*. For instance, consider the family of relations Q_θ which holds between lines ℓ_1 and ℓ_2 just in case there exists another line ℓ_3 which intersects both ℓ_1 and ℓ_2 and is such that the smaller angle formed between both ℓ_1 and ℓ_3 and ℓ_2 and ℓ_3 is θ. Note that since we have $\ell_1 Q_\theta \ell_2$ iff $\ell_1 \parallel \ell_2$, then clearly for all θ, Q_θ grounds *direction* just in case $\parallel$ does. And this might be thought to cause a problem if we were taking (2.32) as a means of implicitly determining the extension of *direction* in terms of the class of lines. For, at the very least, it would invite the question whether the class of abstract objects falling in the range of the function $dir(\cdot)$ implicitly defined via (2.32) is the same as those falling in the range of the function $dir'(\cdot)$ implicitly defined by the analogously formed principle which employed the grounding relation $Q_{\pi/17}$ instead of $\parallel$.

Due to these sorts of considerations, theorists who have wished to appeal to an abstraction principle for an abstract sortal C as a means of delimiting a domain of abstract objects have generally wished to impose some additional adequacy conditions on a grounding relation R for C. Such requirements have generally had a broadly epistemic character. Noonan [101], for instance, requires that a relation R must be "epistemically prior" to a sortal C which it is claimed to ground. In this case, R is characterized as bearing this relation to C if the "necessary order of language acquisition" reflects that we must grasp R prior to grasping C.

Although it is unclear whether Noonan's criterion actually sanctions the use of parallelism to ground the concept *direction*, it presumably does rule out the use of the analogous principle based on Q_θ. More importantly, however, it suggests that in attempting to determine whether a grounding relation R is adequate for a sortal C, we ought to look for a relation in content between C and R. This sort of view is taken to an extreme by the neo-Fregeans who wish to revive Frege's [39] tentative proposal that the natural numbers can be taken to be introduced by the statement now known as Hume's Principle. Recall that this principle has the form

(2.33) $\forall F \forall G[\#F = \#G \iff F \approx G]$

where $\#(\cdot)$ denotes the function expressed by "the number of Fs" and $\approx$ denotes the relation of equinumerosity. Frege [39] tentatively proposed that this statement could be taken as a means of defining the natural numbers. His official view was that (2.33) is not epistemically fundamental since it was derivable from a similar principle about sets which he took to be a principle of pure logic. But it is now well-known that the adjunction of this other principle (i.e. Basic Law V) to the second-order system in which Frege sought to develop arithmetic leads to a contradiction. Although Frege did not take this route himself, neo-Fregean philosophers such as Wright [154] have thus wished to reclaim the elementary status of (2.33) so as to restore the possibility of reviving Frege's view that arithmetic can be reduced to logic.

In the course of doing so, such theorists have wished to defend the view that (2.33) is analytic. When understood in the traditional sense, this means simply that it is true in

virtue of the meaning of "the number of $(\cdot)$" and "is equinumerous with." But note that if we now look on (2.33) as an abstraction principle for the sortal *natural number*, this former expression ought to be taken as serving a role analogous to $dir(\cdot)$ in (2.32) – i.e. that of delimiting a domain of abstract objects as the items its range. The neo-Fregean answer as to why (2.33) – potentially as opposed to some similarly-formed statement employing an equivalence relation materially equivalent to equinumerosity – performs this function with respect to *natural number* turns on the semantic relationship which they claim to hold between the concepts *natural number* and *equinumerous with*.

The claims that there is indeed such a close conceptual connection in content between these statements and that such a connection renders (2.33) analytic have, of course, proven to be highly contentious.[44]However, it is not the actual status of these proposals which bears on our current interests, but rather their potential use in helping us in locating a definition of $\underline{\leftrightarrow}$ to serve as a grounding relation in (2.30). For note that if what makes (2.33) a successful means of introducing the natural numbers as the items in the range of $\#(\cdot)$ turns out to be the semantic relation which *natural number* bears to *equinumerous*, then it follows that in seeking to construct a similar principle for algorithms, we must look for a means of defining $\underline{\leftrightarrow}$ so that this definition bears a similar semantic relation with respect to our background concept of algorithm (say as roughly characterized in Chapter 1.1).

As we will see in Chapters 3, 4 and 5, the task of formulating a truly general definition of this relation turns out to be a surpassingly complex task. However, one substantial source of technical difficulties is the need to define $\underline{\leftrightarrow}$ so that it holds uniformly across different forms of implementations. I have already pointed out in (2.31ii) that it will be important to the general goals of algorithmic realism to ensure that this relation is defined over as broad a class $\mathcal{M}$ as possible. For not only will any artificial delimitation of this class (of the sort which would, for instance, be imposed by equating $\mathcal{M}$ with a single model of computation) restrict the ability of the resulting formulation of algorithmic realism to explain reference to algorithms via implementations which are not in $\mathcal{M}$, but we will also see in Chapter 3 that

[44]For the positive claim, see Wright [154]. For a rebutal see Boolos [12]. For a reply see Wright and Hale [155].

excessive parochialism in this respect may also complicate the formulation of a definition which satisfied (2.31i).

In attempting to provide some insight into how $\underline{\leftrightarrow}$ might be defined so as to satisfy (2.31i,iii), it will, however, be useful to consider a class of implementations which, while not quite broad enough to cover all of the models mentioned in Section 2.3.2, is still sufficiently general to subsume several major computational paradigms. The class I will employ for this purpose is a generalization of the simple transition model introduced in Chapter 1.4. In particular, such a system can be taken to be a septuple $M = \langle X, Y, \Sigma, \delta, H, in, out\rangle$ where X and Y are respectively the domain and the range of the function which we want to view M as computing. The other components are defined as follows:

(2.34) i) Σ the set of *states* of M;

ii) $in : X \to \Sigma$ is the *input function* of M;

iii) $\delta : \Sigma \to \Sigma$ is the *transition function* of M;

iv) $H \subseteq \Sigma$ is the set of *halting states* of M;

v) $out : \Sigma \to Y$ is the *output function* of M

Transition systems operate on input $x \in X$ by iterating δ on the state $in(x)$ so as to obtain a sequence $\sigma_0(x), \sigma_1(x), \ldots \in \Sigma^{\leq\omega}$ defined via

(2.35) i) $\sigma_0(x) = in(x)$

ii) $$\sigma_{n+1}(x) = \begin{cases} \sigma_n(x) & \text{if } \sigma_n(x) \in H \\ \delta(\sigma_n(x)) & \text{else} \end{cases}$$

As defined, the sequence $\sigma_0(x), \sigma_1(x), \ldots$ is always infinite, even in cases where there exists i such that $\sigma_i(x) \in H$. To formalize the distinction between halting and non-halting computations we may additionally need to define the length of the application of M to x to be

(2.36) $$len_M(x) = \begin{cases} 1 + \text{the least } n \in \mathbb{N} \text{ such that } \sigma_n(x) \in H & \text{if such an } n \text{ exists} \\ \text{undefined} & \text{else} \end{cases}$$

In cases where $len_M(x)$ is defined, I will refer to the sequence $exec(x) = \sigma_0(x), \ldots, \sigma_{len_M(x)-1}(x)$ as the *execution* of M on x. And on this basis we may now define the result of applying M to $x \in X$ as $out(\sigma_{len_M(x)}(x))$. This also allows us to define the function induced by the application functional for M as $App(M, x) = out(\sigma_{len_M(x)}(in(x))$. Note that $App(M, x)$ thereby induces a possibly partial function of type $X \rightarrow Y$.

Although the structure of the states and transition functions will vary substantially from one model to another, I will argue in Chapter 4 that all well-known forms of transition- and register-based models of computation can be naturally represented as transition systems of this form.[45] This is partly due to the fact that if we fail to impose any additional constraints on the definition of Σ and δ, then the current definition corresponds to what I will refer to as an *open* model of computation – i.e. one whose members may individually operate on arbitrary mathematical structures (i.e. the states in Σ) via the application of arbitrary mathematical operations (i.e. the transition function δ). While this is a desirable feature in many contexts, it obviously implies that there are transition systems which compute arbitrary non-recursive functions by making use of infinitary operations or structures. It is, however, straightforward to restrict the definition of Σ and δ so that this is no longer possible (cf., e.g., Gandy [42], Boker and Dershowitz [11]). And although I will not go into the details concerning how this may be achieved, I will subsequently assume that such a restriction has been effected so as to ensure that we may pick out a determinate class of transition systems $\mathcal{M}$.

It is easiest to gain insight into what is meant by implementing an algorithm as a transition system by considering concrete examples. This is how I will proceed in Chapter 3. But so as not to become bogged down in details prematurely, it will be useful to now attempt to characterize this relation informally. I have alluded above to the idea that part of what is involved by grasping an algorithm A is that we have some idea of what it would mean to carry it out on a given input. For suppose A is given to us as a formal program or pseudocode specification taking inputs in set X. In this case, one way of

[45] For this reason, several definitions very similar to this one have been proposed by a variety of different theorists – e.g. Gandy [42], Moschovakis [95], Gurevich [51]. I will return to discuss the status which these theorists assign the transition model in Chapter 5.

understanding what it means to carry out A is as a process which, given an arbitrary value $x \in X$, eventuates in an informal calculation whose intermediate stages $\alpha_0(x), \ldots, \alpha_{n-1}(x)$ are derived by executing the individual steps in terms of which A is specified.

But now consider the case where we are given a transition system M and then asked to consider what algorithm it implements. In this case, M itself is specified in terms of formally defined computational states and transitions. In particular, presuming that M halts on all inputs, there will be a finite sequence of states $exec(x) = \sigma_0(x), \ldots, \sigma_{len_M(x)-1}(x)$ corresponding to the execution of M on each $x \in X$. The claim which I will advance in the next chapter is that what would justify us in regarding M as an implementation of A is that if for all $x \in X$, there is a means of systematically correlating the formally defined states $\sigma_i(x)$ appearing in M's execution on x with the informal stages of the form $\alpha_j(x)$ derived by carrying out A on x. Although there is still much latitude in how we define a "systematic correlation" of stages and states, the general idea is that M may be said to implement A just in case the execution of M on x may be seen as a precise "working out" of the execution of A on x.

The foregoing is an attempt to provide a rough intuitive characterization of what is presumably meant in practice when we speak of a machine implementing a particular algorithm. As such, these observations can also be taken as a possible first step in an attempt to characterize the function $imp(\cdot)$ which, per (2.30), should now be thought of as delimiting the class of algorithms as its range. Note, however, that we concluded in Section 3.1 that an algorithmic realist should not be held responsible for defining this function explicitly as long as he can show that it is definable implicitly by giving an appropriate definition of $\underline{\leftrightarrow}$.

Returning now to the question of how such a definition might be given, consider an instance in which we are given two transition systems $M_1, M_2 \in \mathcal{M}$ and are asked if they implement the same algorithm. In this case, there is no external procedure to which we can compare the operations of M_1 and M_2 singularly. We may, however, attempt to compare the operation of M_1 with that of M_2. In particular, we can attempt to determine if M_1 and M_2 operate in a manner which would justify us in saying that they "work the same way" in the sense of having similarly structured executions $\sigma_0(x), \ldots, \sigma_{len_M(x)-1}(x)$ and

$\tau_0(x), \ldots, \tau_{len_M(x)-1}(x)$ on all inputs $x \in X$. For note that if their executions were found to be systematically related to one another in an appropriate manner, then it would be reasonable to conclude that they implemented the same algorithm despite the fact that they were not explicitly introduced as doing so.

Here again there is much room left in which to wiggle with respect to the formalization of such notions as "works the same way" and "similarly structured." And thus parlaying the foregoing observations into a formal definition of $\underline{\leftrightarrow}$ over $\mathcal{M}$ will require additional work. The task of doing this will, again, be greatly facilitated by considering concrete examples, as I will do Chapter 3. But since it is not until a precise definition is produced that we can attempt to determine whether $\underline{\leftrightarrow}$ satisfies requirements (2.31i) and iii), it will be useful to proceed one step further on the basis of general considerations alone.

To this end, it is useful to note that while the observations adduced in the previous paragraphs certainly do not determine a unique definition for $\underline{\leftrightarrow}$ over $\mathcal{M}$, they do share a motivation with an important class of computational relations studied in theoretical computer science under the name of *machine simulations*. Roughly speaking, a machine simulation between implementations M_1 and M_2 (which for present purposes can be assumed to be transition systems) is a way of relating their respective sets of states Σ_1 and Σ_2 so that pairs of states $\sigma, \sigma' \in \Sigma_1$ such that $\delta_1(\sigma) = \sigma'$ are S-related to pairs of statements $\tau, \tau' \in \Sigma_2$ such that $\delta_2(\tau) = \tau$.

In other words we require that S satisfy the following requirement:

(2.37) $\forall\sigma\forall\sigma'\forall\tau[(\delta_1(\sigma) = \sigma' \wedge \sigma S\tau) \rightarrow \exists\tau'(\sigma' S\tau' \wedge \delta_2(\tau) = \tau')]$

This condition ensures that every state transition which might occur in some execution of M_1 will be mirrored by a transition in M_2 which is ensured to be analogous in the sense that if the initial states σ and τ are S-related, then the subsequent states σ' and τ' will be as well. Such a condition is often referred as the "forth" condition in a stronger form of simulation relation referred as a *back and forth* equivalence. In particular, a back and forth equivalence between M_1 and M_2 is a relation $S \subseteq \Sigma_1 \times \Sigma_2$ satisfying (2.37) and also the converse "back" condition

(2.38) $\forall\tau\forall\tau'\forall\sigma[(\delta_2(\tau) = \tau' \wedge \sigma S\tau) \rightarrow \exists\sigma'(\sigma' S\tau' \wedge \delta_1(\sigma) = \sigma')]$

which ensures that every transition in M_2 is mirrored under the converse of the relation S in M_1.

In the literature of modal logic and process algebra in which such notions were first introduced, a relation S between implementations M_1 and M_2 satisfying both (2.37) and (2.38) is also known as a *bisimulation*. Correspondingly, M_1 and M_2 are said to be *bisimilar* just in case such a relation exists. Such a definition can either be made more specific or generalized in a number of ways giving rise to a family of relations with similar definitions all of which are conventionally termed "bisimulation." One can, for instance, replace the requirement that single state transitions in M_1 are mirrored by single state transitions in M_2 by the less restrictive requirement that they are mirrored by finite sequences of transitions (and vice versa). Another common variation is to introduce a system for labeling transitions in M_1 and M_2 according to the type of computational action they represent and to then require that the labels of each transition be preserved under the simulation relation as well as the ordering of states. I will discuss how such modifications to the conditions (2.37) and (2.38) can be formalized as well as why they may be desirous in the next chapter.

It should be clear, however, that under most foreseeable sophistication of the definition of bisimulation, the corresponding definition of bisimulation will turn out to be an equivalence relation on the class over $\mathcal{M}$ of implementations. Such a definition thus at least has the appropriate formal properties to serve as a definition of $\underline{\leftrightarrow}$ in (2.30). Of course it remains to be seen whether any relation in this family satisfies any of (2.31i-iii). For although I have motivated the definition of bisimulation given above on the basis of general reflections about the notions of algorithm and implementation, these considerations are not sufficiently refined to determine a precise mathematical definition of this relation. Thus we are still a fair distance from being able to judge whether any form of bisimulation relation can be taken as a conceptually adequate grounding relation for the notion of algorithm.

In Chapter 3, I will attempt to push considerably harder on intuitions about the relationship between algorithms and implementations. Out of this will come a number of concrete examples against which we can test not only our intuitions about the conceptual adequacy of various definitions of $\underline{\leftrightarrow}$, but also about its extensional adequacy in the sense

of (2.31i). However, as we are about to find out, doing so embroils us in considerable questions of detail about the features of the individual models of computation. And matters will become yet more complex in Chapter 4 when I consider the potential for generalizing such definitions over a larger class of implementations so as to address (2.31ii).

Chapter 3

A cautionary example

3.1 Introduction

The purpose of the previous chapter was to delimit the possible strategies open to a proponent of algorithmic realism. In the course of so doing, I presented a positive proposal on behalf of the realist which was intended to illustrate what I argued was the most plausible means of demonstrating the dual theses that algorithms are mathematical objects and that computational properties like running time complexity are structural properties of these objects. The view I presented was grounded in an array of linguistic, metaphysical and technical considerations about how we employ, make reference to, and reason about procedures both inside and outside mathematics and computer science. And arguably due to the very richness of these desiderata, both the proposal I outlined and the considerations offered in its favor were fairly complex.

It is, of course, my ultimate intention to show that algorithmic realism is false. And it is to this task which this and the following two chapters will be devoted. But before commencing in earnest, it will be useful to briefly reprise several of the conclusions of Chapter 2. The most significant of these pertains to the logical form of algorithmic realism. As we have seen, this view can be taken as an ontological thesis about how discourse about algorithms can be uniformly reinterpreted as discourse about mathematical objects. In particular, I suggested that the realist's central theses would be vindicated if it could be shown that there exists *some* means of systematically identifying individual algorithms with mathematical objects in a manner which simultaneously preserves their computational properties while also maintaining intuitions about algorithmic identity and non-identity as they are reflected in computational practice.

It thus follows that the claim that algorithmic realism is *false* is equivalent to the

negative universal statement that *no* such mode of identification exists. And from this it follows that in order to demonstrate that this view is not tenable, it is not sufficient to simply refute individual proposals which have been (or might be) put forth with regard to how such a system of identifications can be set up. Rather, we must attempt to argue for the general claim that no such proposal could ever be adequate. And since it seems unlikely that this can be done in exhaustive manner – i.e., by anticipating all potential modes of identification which an algorithmic realism might put forth in favor of his view – it may seem proportionally improbable that we can construct a truly general argument against algorithmic realism.

One way around this problem would be to demonstrate that the background, informal notion of algorithm was somehow paradoxical in something like the same way as the "naive" notion of set. But I argued in Chapter 2 that there are good reasons to think this is unlikely. Primary among them are a variety of results in recursion and type theory which may be interpreted as showing that no inconsistency arises when algorithms (here interpreted, e.g., as Turing machines or untyped lambda terms) are allowed to apply to themselves. In a typical context, self-applicability is achieved by defining a map c from a domain of procedural entities $\mathcal{D}_2$ into a domain of $\mathcal{D}_1$ of objects on which the members of $\mathcal{D}_2$ operate. For instance, if $\mathcal{D}_2$ is taken to be the set of all Turing machines, c can be taken to be a form of godel-numbering which maps (structural descriptions of) Turing machines into natural numbers. A given machine T can then "operate on itself" by taking as input $c(T)$. But it can be shown that not only does this result not lead to any sort of contradiction, it in fact leads to a positive result in the form of the Kleene Recursion Theorem.[1] Since most well-known semantic or logical paradox (e.g., the Liar, Cantor's Paradox, Russell's Paradox, Curry's Paradox, etc.) are typically analyzed as arising due to the possibility of self application made possible by a mixing of logical "levels" (e.g., those that of object and function or object language and metalanguage), the fact that the self applicability of

[1]I.e. for all Turing machines $T : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, there exists a Turing machine R such that for all n, $\varphi_T(c(R), n) = \varphi_R(n)$ (where φ_M denotes the (extensional) function computed by machine M). The Recursion Theorem is a fundamental tool in recursion theory as it allows us to obtain a fixed point of any effective function t mapping (descriptions of) Turing machines to (descriptions of) Turing machines – i.e. a Turing machine R such that for all n, $\varphi_R(n) = \varphi_t(c(R))(n)$. The paid for self-applicability of this form is the existence of effectively computable but non-total functions.

procedures does not lead to paradox is at least suggestive of the fact that the "naive" notion of procedure I have sought to analyze in previous chapters suffers from the same sort of logical difficulties which plague the analogous naive notions of set and truth or satisfaction.[2]

If algorithmic realism turns out to be unsustainable, it thus seems unlikely that this is because of what might be broadly described as *logical* grounds. Rather, I will argue that the flaw which infects the notion of algorithm derives essentially from the fact that this notion is called upon to play too many distinct theoretical functions in our computational practices. As I have discussed in previous chapters, these roles may be divided into two broad categories: 1) that of serving as the abstract machine- and language- independent entities which are expressed or implemented by mathematical items like programs or machine models; 2) that of serving as the direct bearers of computational properties such as running time complexity and provable correctness.

The argument which I will present in general form in Chapter 5 is meant to show that any means of associating individual procedures with mathematical objects that satisfies the first class of requirements will of necessity fail to satisfy at least certain instances of the second class, and conversely, any such association satisfying the second requirement will, of necessity, fail to satisfy at least certain instances of the first. I will argue that a consequence of this is that it turns out to be impossible to construct a mathematical theory of algorithms which simultaneously ensures that individual algorithms have the computational properties which we take them to have in practice and also ensures that various robust intuitions about procedural sameness and difference are preserved. It will thus be my contention that the Achilles heel of algorithmic realism is its acceptance of the view that there is a stable notion of *algorithmic identity* of the sort, which I have argued in Chapter 2, is implicit in our willingness to treat procedures as objects.

In order to bolster this view, I will ultimately argue in Chapter 5 that it is impossible to provide a definition of a bisimulation relation defined over an acceptable class of implementations $\mathcal{M}$ which comports with the full range of theoretical and conceptual constraints which must be placed on a general theory of algorithm such as the theory T_p described in

[2]For general discussion of the relationship between self reference in computability theory and the semantic paradoxes, cf., e.g., [41], [158], [49].

Chapter 3.3 As a precursor to this, in Chapter 4 I will attempt to systematically delimit the forms the class $\mathcal{M}$ could potentially take based on a variety of background constraints deriving both from computational practice and more from the theoretical role I argued implementations must play in Chapter 3.2. Both of these arguments will be relatively involved as they rely on a variety of technical considerations deriving both from the practice of theoretical computer science in general and also from the specific details of proposals which have been put forth by the theorists who explicitly attempt to argue in favor of algorithmic realism.

These exegetical complications aside, the general form of the argument which I will offer in Chapter 5 is relatively straightforward. In particular, suppose that we have agreed that any acceptable theory by which algorithmic realism might be vindicated must resemble the sort of general theory of algorithms I described in Chapter 2 whose primary feature I will now briefly review. Primary amongst these features is the recognition that all reference to procedure must be mediated by reference to programs or implementations. This means in particular that whenever we take ourselves to succeed in referring to an algorithm A, this must be because there exists either a specific program Π or a implementation M such that A has been introduced either as

(3.1) a) $a =_{df}$ the procedure expressed by π

b) $a =_{df}$ the procedure implemented by m

where a is a proper name (like "Euclid's algorithm') which we take to denote A and π and m are (typically complex) mathematical description of Π and M. I argued in Chapter 2 that programs only allow us to refer to procedures in conjunction with an operational semantics for the language over which they are stated. But since the result of interpreting a program relative to such a semantics is itself an implementation (typically drawn from an antecedently defined class of "abstract machines"), I also argued that schema (3.1a) can be assimilated to (3.1b).

Even after taking this simplification into account, however, there is still another substantial problem in understanding how (3.1b) allows us to refer to procedures. For as we have seen, the relationship between machines and algorithms is typically *many-one* – i.e.,

for every algorithm A, there will generally exist an open class of distinct implementations $M_1, M_2, \ldots$, each of which can be taken to implement A with equal plausibility. It is for this latter reason that I argued in that any foundational theory which wants to accommodate even the most general contours of our everyday discourse about algorithms must adapt to what I referred to as an *abstractionist* theory about the nature of algorithms.

This view can be readily understood in opposition to a more straightforward *reductionist* approach according to which individual algorithms are directly indentified with individual mathematical objects. According to this view, for instance, the algorithm denoted by the name MERGESORT might be directly identified with a particular RAM machine, lambda expression or (more plausibly) set of recursive equations. In contrast to such a proposal, the abstractionist view seeks to intepret the expression "the algorithm implemented by m" as denoting a mapping from individual implementations onto an abstract domain of algorithms. According to such a view, a statement such as (3.1b) should thus be treated as having the logical form

(3.2) $A = imp(M)$

where imp is intended to denote a function from a domain of implementations to the domain of algorithms.

Such a view is designed to accommodate the fundamental observation recorded above that the relationship between implementation and algorithms is typically many-one. In the abstractionist setting, this is allowed since we may have distinct implementations M_1, M_2 such that $imp(M_1) = imp(M_2)$. However, once an algorithmic realist acknowledges that such a situation is possible – i.e. that two implementations may be used to refer to the same algorithm – it then become incumbent upon him to say how this can be so. Since according to the abstractionist strategy, algorithms *just are* objects which inhabit the range of the function denoted by imp, this task must presumably be accomplished in terms of the structural features of implementations themselves. If we assume that the latter sort of entities form a well-defined mathematical domain $\mathcal{M}$, then one way in which this can be accomplished is by defining an equivalence relation $\underline{\leftrightarrow}$ over this class which holds between the implementations denoted by M_1 and M_2, just in case these objects implement the same

algorithm.

These observations naturally motivate the formulation of a so-called *abstraction principle* for algorithms with the following form:

(BP) $imp(M_1) = imp(M_2) \Leftrightarrow M_1 \underline{\leftrightarrow} M_2$

For reasons which will emerge below, I will refer to a specific relation $\underline{\leftrightarrow}$ which might be proposed to figure in such a principle as a *bisimulation relation* and to (BP) itself as the *bisimulation principle*. I argued in Chapter 2 that a proponent of abstractionism will wish to employ (BP) to fill the same theoretical role to which latter-day interpreters of Frege's theory of natural numbers have wished to appoint the analogously formed statement often known as *Hume's Principle*. This statement has the form

(HP) $\#F = \#G \leftrightarrow F \simeq G$

where F and G are variables over *Fregean concepts* (which for present purposes may be thought of as classes), $\simeq$ denotes the (second-order-definable) relation of equinumerosity and # is intended to denote a function mapping concepts into numbers. Frege himself did not take (HP) as a definition of natural number, but rather sought to derive this statement from what he took to be a more basic principle about the identity of sets. However since this principle (known as Basic Law V) was later shown to be inconsistent, followers of the so-called *neo-Fregean* movement in the philosophy of mathematics have sought to isolate a consistent subtheory of Frege's logic which treats (HP) as an axiom. In this context, various theorists (most notably Wright [154] and Hale [155]) have attempted to argue that (HP) performs two important theoretical purposes: 1) it serves as an implicit definition of the function # thereby explicating the ontological status of natural numbers as the items in its range; and 2) in virtue of its form, (HP) also fixes the truth conditions of statements of the form "the number of Fs = the number of Gs" and can thereby be seen as a means of fixing the identity conditions of natural numbers.

I argued that algorithmic realists must ultimately look to a principle like (BP) to achieve the same two functions with respect to the general notion of algorithm. I argued in Chapter 2.3 that, modulo a number of technical and conceptual issues, such a view holds a certain degree of intuitive plausibility. For since there is no well-established background theory which

answers even basic metaphysical and logical questions about our computational practices, our best hope for systematically understanding our discourse about algorithms appears to reside in the hope that our current theoretical practices involving reasoning about implementations can be taken to be embodied in an implicit definition. But since we have also agreed that there is an independent argument to the effect that reference to algorithms must be mediated in the manner of (3.1b), it appears that such a definition must take the form of a principle like (BP) which attempts to fix their identity conditions of algorithms in terms of the implementations by which we have specified them.

The considerations leading to this conclusion are admittedly complex and in some cases not as clearly elucidated in the technical literature of computer science as one might hope. However the burden of Chapter 2 was to argue that these practices are sufficiently conventionalized to entail that *if* it proves possible to develop a mathematical theory of algorithms of the sort required to sustain algorithmic realism, then we would have little choice but to proceed in the manner just outlined. But of course this is merely a methodological observation about what sort of positive theory of algorithms a realist might hope to construct. As such, the conclusion that abstractionism is the most promising theoretical framework in which to pursue a vindication of algorithmic realism itself does little to determine how the relation $\underline{\leftrightarrow}$ must be defined so that (BP) entails statements about implementation which are consistent with our extant practices. And perhaps more seriously yet, the mere observation that reference to procedures must be secured by applying the function denoted by *imp* to concrete implementations does little to determine how its domain $\mathcal{M}$ is formally determined. For these reasons, it seems reasonable to conclude that the central theoretical burdens of algorithmic realism are precisely those of defining $\underline{\leftrightarrow}$ and $\mathcal{M}$ so that (BP) can play the two roles described above.

While I will suggest below that providing proper definitions of $\underline{\leftrightarrow}$ and $\mathcal{M}$ corresponds to substantial (and in my own view, unsurmountable) hurdles which algorithmic realism must clear, we also do not start from scratch in constructing these definitions. In particular, the definition of $\mathcal{M}$ which the realist seeks to provide ought to serve as a general analysis of what I have been referring to as an *implementation* of an algorithm – i.e,. an instance of a model of computation of the sort briefly described in Chapter 2.2 and which we will study

more extensively in Chapter 4. Accordingly, the definition a realist offers of $\underline{\leftrightarrow}$ ought to correspond to an analysis of whatever relation of computational equivalence we take to hold between a pair of machines M_1 and M_2 in virtue of which we are willing to regard them as implementations of the same algorithm.

In Chapter 2.3 I attempted to push these observations further by arguing that $\mathcal{M}$ ought to be analyzed in terms of a notion I refered to as a *transition system*. Roughly speaking, a transition system M is a collection of objects referred to as *computational states* Σ_M which are meant to correspond to states of a machine-like model during the course of its computation together with a so-called *transition function* $\Delta_M : \Sigma_M \to \Sigma_M$ which is meant to correspond to the rule by which such states are updated or transformed during the course of M's computations. Variants of this notion which have been employed by a variety of theorists as a generalized form notion of model of computation. And for this reason, there is an outstanding question as to how an algorithmic realist might define $\mathcal{M}$ with sufficient generality to ground (BP). I also offered some considerations in Chapter 2 in favor of taking $\underline{\leftrightarrow}$ to be among a family of definitions related to the notion of bisimulation which have been studied extensively in process algebra (cf., e.g., [86]) and modal logic (cf., e.g., [8]). However these remarks must also be seen as provisional in advance of a more definite analysis of the notion of implementation itself.

In Chapter 4, I will present arguments in favor of adopting the transition system model as a conceptually adequate definition of $\mathcal{M}$, and some form of bisimulation as a conceptually adequate definition of $\underline{\leftrightarrow}$. To reiterate, I take not only the adoption of the general abstractionist framework, but also these particular choices for $\mathcal{M}$ and $\underline{\leftrightarrow}$ and how they should be defined within this framework as decisions which are essentially forced on the algorithmic realist by various features of our practices and intuitions about algorithms. Once these decisions are made, however, we can finally enter into a detailed analysis of whether this view is viable in the sense of making predictions which are consistent with these practices and intuitions.

In Chapter 5, I will argue that algorithmic realism is *not* sustainable in this sense. In particular, I will argue it is impossible to choose precise definitions for $\mathcal{M}$ and $\underline{\leftrightarrow}$ so that the resulting version of (BP) is consistent with the overarching aim of the algorithmic

realist to offer a systematic reconstruction of our theoretical practices involving algorithms in a completely mathematical language. The specific version of this argument I will offer there is grounded in technical details about how recursive models of computation should be assimilated to transition systems. However, the form of this argument is more general than this specific application might appear to suggest. And for this reason it will be useful to both describe the form of this argument by considering a simpler example in detail. And it is to this purpose which the present chapter is devoted.

As an initial step, note that among the statements of ordinary computation discourse for which an algorithmic realist will presumably wish to account are those stating that a particular implementation M implements a particular algorithm A_1. For instance A_1 might be the algorithm MERGESORT and M_1 might be some particular RAM machine which has been constructed explicitly as an implementation of this algorithm. As per the preceeding discusion, we can assume that reference to A must be mediated by reference to implementations such as M_1, but that M_1 is not the unique implementation with this property. Suppose, for instance, that we had also constructed another distinct implementation M_2 as another implementation of A_1. Then, using the terminology adopted above, we would have

(3.3) 1. $A_1 = imp(M_1)$.

2. $A_1 = imp(M_2)$.

But now also suppose that A_2 is an algorithm which we regard in practice as distinct from A_1. For instance, if A_1 was MERGESORT, then A_2 might be like INSERTIONSORT which has a different running time complexity or differs in some other computationally significant manner from A_1. In such a case we should expect the following two statements to also turn out to be true:

(3.4) a) $A_1 \neq A_2$

b) $A_2 = imp(M_3)$

Since we will soon enounter an example of the sort of situation just outlined, I will take it to be uncontroversial that such instances do arise in the course of computational practices.

However, it should already be evident that the existence of pairs of distinct algorithms and triples of implementations with the properties just described places substantial constraints on how $\underline{\leftrightarrow}$ may be defined so as to ensure that the principle (BP) by which the realist intends to give the identity conditions of algorithms is satisfied. For note that if a definition of $\mathcal{M}$ has been given such that this class contains M_1, M_2, M_3 then it follows that any adequate definition of $\underline{\leftrightarrow}$ over this class must satisfy

(3.5) a) $M_1 \underline{\leftrightarrow} M_2$

b) $M_1 \underline{\nleftrightarrow} M_3$

c) $M_2 \underline{\nleftrightarrow} M_3$

For note that it follows from (3.3a,b) that $imp(M_1) = imp(M_2)$. And thus (3.5a) follows from the left-to-right direction of (BP). Similarly, (3.5b) follows from (3.4a) together with (3.4b) and (3.3a) and the right-to-left direction of (BP).

I will call a definition of $\underline{\leftrightarrow}$ which satisfies the statements of identity and non-identity which are embedded in computational practice an *extensionally adequate* definition of bisimulation. This is clearly a requirement which must be met by any definition of $\underline{\leftrightarrow}$ which will be acceptable to an algorithmic realist. For recall that in Chapter 2, I argued that a commitment to the determinateness of algorithmic identity statements is entailed by the realist's underlying belief that algorithms themselves constitute a domain of abstract objects with properties from implementations or programs. And since at least some identity and non-identity statements about algorithms are entrenched within our computational practices, no definition of $\underline{\leftrightarrow}$ which violated such a statement will acceptable to a realist.

But it should also be clear that the precise definition of $\underline{\leftrightarrow}$ adopted by an algorithmic realist must also satisfy a variety of additional properties. Perhaps the most significant of these is that such a definition ought to function as a plausible conceptual analysis of what we take it to mean that two implementations implement the same algorithm. For note that if the realist hopes to regard (BP) as serving the purpose of fixing the identity conditions of algorithms, then the condition appearing as its right-hand side ought to serve to analyze what it means for the algorithms determined as $imp(M_2)$ and $imp(M_2)$ to be the same algorithm. Suppose A_1 and A_2 are the algorithms determined as the denotation

of these terms in some model of (BP) taken in conjunction with the theory T_p by which the realist proposes to axiomatize our computational practices. As per the observations cataloged above, A_1 and A_2 must be determined by *some* such implementations. And from this it follows that for (BP) to perform the function of determining whether A_1 and A_2 are the *same* algorithm, $\underline{\leftrightarrow}$ must be defined so that it holds between any arbitrary pair of implementations just in case they implement the same algorithm.

The intention behind (BP) is thus semantic in character in the sense that finding a successful definition of $\underline{\leftrightarrow}$ would render the corresponding version of (BP) analytic. And for this reason, I will call a definition of $\underline{\leftrightarrow}$ which succeeds in giving the meaning of algorithmic identity in terms of the structural properties of implementations an *intensionally adequate* definition of bisimulation. However I also acknowledged in Chapter 2 that our intuitions about the properties of algorithms may not be sufficiently robust to uniquely determine a relation which serves this function. But taken in conjunction with the rest of contemporary computational practice, such intuitions are strong enough to impose a number of non-trivial adequacy conditions on $\underline{\leftrightarrow}$.

The most substantial of these follows from the fact that our intuitions pertaining to when M_1 and M_2 implement the same algorithm presumably derive from whatever informal understanding we have about their mode of operation. Whatever is meant by this will, of course, vary with the choice of the class $\mathcal{M}$ from which these structures are drawn. But in the case where $\mathcal{M}$ coincides with (or even subsumes) the transition system model mentioned above, it is clear what form a general analysis of coincidence in mode of operation should take. In particular, if M_1 and M_2 are transition systems, then to say that they operate in the same manner is to say that a relation between their states may be set up so that for all inputs, the transitions of M_1 are thereby correlated with transitions of M_2 (and conversely). As we will see below, this sort of definition naturally has the form of a bisimulation relation in the technical sense of modal logic and process algebra mentioned above.

This can be taken to mean that the definition of $\underline{\leftrightarrow}$ must be given in a certain way in terms of the features of the class chosen as $\mathcal{M}$. For instance, if the members of $\mathcal{M}$ corresponded to transition systems in the manner described above, then $\underline{\leftrightarrow}$ ought to hold between M_1 and M_2 just in case there existed an appropriate way of correlating their states

so that transitions in the former were paired with structurally analogous transitions of the latter. For note that it is precisely in virtue of the existence of such a relation linking states (or more generally, transition-linked sequences of states) which perform the same computational function relative to the computational problem solved by M_1 and M_2 that we are generally willing to regard M_1 and M_2 as implementations of the same algorithm.

Considerations of this of this kind thus lead to a rough initial characterization of the form which an intensionally adequate definition of $\underline{\leftrightarrow}$ should take. As we will see below, however, such intuitions are far too schematic to determine its precise formulation. In order to determine whether such a definition is indeed possible, we may also look to computation as source of additional intensional constraints on the definition of $\underline{\leftrightarrow}$. In particular, it follows from our practice of ascribing computational properties like running time complexity directly to algorithms that $\underline{\leftrightarrow}$ ought to be a *congruence* with respect to these properties. For suppose that $\Phi(X)$ denotes a property such that our practices suggest that $\Phi(A)$ holds for some particular algorithm A (e.g., if A were MERGESORT, then $\Phi(X)$ might be "X has running time $O(|x| \log_2(|x|))$), then it ought to be the case that for all implementations M_1 of A, if $M_1 \underline{\leftrightarrow} M_2$, then $\Phi(imp(M_2))$. This requirement can be stated more generally as follows:

(3.6) For all implementations M_1 and M_2 and for all computational properties Φ, if $M_1 \underline{\leftrightarrow} M_2$, then $\Phi(imp(M_1))$ if and only if $\Phi(imp(M_2))$.

Viewed as an adequacy on a definition of $\underline{\leftrightarrow}$, (3.6) states that such a relation may only link implementations which determine algorithms with the same computational properties. For instance, this requirement would rule out a candidate definition of $\underline{\leftrightarrow}$ which held between implementations M_1, M_2 such that M_1 implemented a linear time algorithm and M_2 implemented a quadratic time algorithm.

The central arguments of both this chapter and Chapter 5 will be aimed at showing that once we have fixed a particular definition of $\mathcal{M}$, it then becomes impossible to define a relation $\underline{\leftrightarrow}$ which is simultaneously intentionally and extensionally adequate. The version I will consider in this Chapter is based on fixing $\mathcal{M}$ to correspond to the class of single tape, single head Turing machines. We will see in Chapter 4 that this model is too simplistic

to be taken seriously as an analysis of the general notion of implementation on which the definition of $\mathcal{M}$ ought to be based. But it is sufficiently expressive so as to allow us to formulate a wide variety of plausible implementations of two intuitively distinct algorithms for deciding whether a string is a palindrome. Taking these procedures to be A_1 and A_2, I will then construct two classes of implementations $\boldsymbol{S}$ and $\boldsymbol{U}$ over $\mathcal{T}$ such that there are $S_1, S_2 \in \boldsymbol{S}$ and $\boldsymbol{U_1} \in \boldsymbol{U}$ which I will argue play roles analogous to M_1, M_2 and M_3 in statements (3.3) and (3.4). From this it follows that any extensionally adequate definition of $\underline{\leftrightarrow}$ must satisfy the corresponding instance of (3.5). And on this basis I will argue that any intensionally adequate definition of this relation will either fail to satisfy either (3.3a) (by virtue of being "too fine grained") or one of (3.5b,c) (by virtue of being "too coarse grained").

Even though the choice of algorithms and implementations is designed in this case to be as straightforward and familiar as possible, there are still many matters of detail which must be laid before the latter argument can be made. This will be accomplished over the course of Section 2 and 3 where I will first introduce the palindrome decision problem and the algorithms which will serve as A_1 and A_2 (Section 2) and then go about constructing the classes $\boldsymbol{S}$ and $\boldsymbol{U}$ and argue that they have the appropriate properties (Section 3). Section 4 will then be devoted to the exigencies which an algorithmic realist would have to face in constructing $\underline{\leftrightarrow}$.

Many of the considerations which arise in the course of this exposition are matters of detail which may appear to pertain to the particular choice of algorithms and implementations involved. However, I will argue over the course of Chapter 4 and 5 that many of these concerns generalize even when we attempt to construct a maximally general definition of $\mathcal{M}$. And it is precisely for this reason why it is useful to study a simple example of the sort I am about to present in detail.

3.2 The decision problem for palindromes

Many of the problems which arise when we attempt to carry the abstractionist programme are most readily appreciated by considering specific examples of algorithms whose

status as mathematical objects a realist will presumably wish to account. I have postponed presenting such an example until this point due to the variety of technical details which arise immediately when we start to consider even the simplest algorithms and classes of implementations. But if the algorithmic realist is to convince us of the plausibility of his programme, the onus lies squarely on his shoulders to show us that these details can be sorted out.

The example I will develop is comprised of four components whose definition we must supply: i) a formal language L whose decision problem we wish to design algorithms to solve; ii) two distinct, informally specified algorithms A_1 and A_2 for deciding membership in L; iii) a formal definition of a class of implementations $\mathcal{M}$ containing disjoint subclasses $\mathcal{M}_1$ and $\mathcal{M}_2$ which may plausibly be taken to implement A_1 and A_2; and iv) a series of proposed definitions of a bisimulation relation $\underline{\leftrightarrow}$ defined on $\mathcal{M}$. I will initially attempt to emulate the strategy of a potential algorithmic realist by showing how the judgment that A_1 and A_2 are distinct mathematical objects may be borne out by showing that for all $M_1, M_2 \in \mathcal{M}_1$, $M_1 \underline{\leftrightarrow} M_2$ and for all $N_1, N_2 \in \mathcal{M}_2$, $N_1 \underline{\leftrightarrow} N_2$ but for no pair $M \in \mathcal{M}_1$ and $N \in \mathcal{M}_2$ is it the case that $M \underline{\leftrightarrow} N$. However, once we have fixed L, A_1 and A_2 and $\mathcal{M}$, my ultimate goal will be to show that it is impossible to construct an intensionally adequate definition of bisimulation satisfying these constraints.

For L, I will take the language L_{pal} of palindromes over $\{0,1\}^*$ – i.e., finite strings of even length comprised of 0 and 1 which read the same both backwards and forwards. For instance, the strings ϵ (the empty string), 00, 1001 and 0101001010 are all members of L_{pal}, while the strings 0, 1, 000, 101101101 are not. Recall that we say that an algorithm A is a *decision procedure* for L_{pal} just in case for all $w \in \{0,1\}^*$, when applied to w, A produces the output `yes` if $w \in L_{pal}$ and `no` otherwise.

Next consider the following two procedures for deciding membership in L_{pal}, which I will henceforth refer to as Pal1 and Pal2. Pal1 consists of taking a string $w = a_1 \ldots a_n$ and comparing symbols inwards from its two ends – i.e., making the comparisons $a_1 \stackrel{?}{=} a_n, a_2 \stackrel{?}{=} a_{n-1}, \ldots, a_{\lfloor n/2 \rfloor} \stackrel{?}{=} a_{\lfloor n/2 \rfloor + 1}$ – and responding `no` as soon as one of the comparisons fails and `yes` otherwise. Pal2 consists in first writing down the string w^R corresponding to the *reversal* of w – i.e. $w^R = b_1 \ldots b_n$ such that $b_i = a_{n-(i-1)}$ for all $1 \leq i \leq n$ –

and then comparing w and w^R symbol by symbol – i.e., making the comparisons $a_1 \stackrel{?}{=} b_1, a_2 \stackrel{?}{=} b_2, \ldots a_n \stackrel{?}{=} b_n$ – and responding `no` as soon as one of the comparisons fails and `yes` otherwise.

Our first task is to assess whether there is any objective way of determining whether PAL1 and PAL2 are indeed distinct procedures. Since I have already indicated that accounting for the identity conditions of informally presented algorithms is one of the primary technical obstacles confronting the algorithmic realist, it is obviously important that this question not be begged at the outset. We must hence seek to determine whether there is an objective basis for regarding PAL1 and PAL2 as distinct algorithms. The first thing to note in is that, my choice of distinct appellation notwithstanding, we have no prior history of using the *names* PAL1 and PAL2. And there is thus no explicit component of our present computational practices which forces us to regard the procedures described in the previous paragraph as distinct algorithms.[3] And thus if we to wish isolate a principled reason for distinguishing between these procedures, it seems that we have little choice but to start down the path of converting the informal descriptions given above into more precise specifications so that we might examine their properties in more detail.

As we have seen in previous chapters, a typical first step in this process would be to regiment the natural language descriptions given above into the idiom of pseudocode. The outcome of this process would be something like the following:

(3.7)

PAL1(w)

Step 1: Let n be the length of w.

Step 2: For $i = 1$ to $\lceil n/2 \rceil$ do

if $a_i \neq a_{n-(i-1)}$, then output `no` and halt.

Step 3: Output `yes` and halt.

[3]This situation should be compared with that of sorting or graph algorithms of the kind described in Chapter 2. Here I argued that not only do our practices explicitly reflect that we distinguish different algorithms for solving the same computational problem, but that a principled basis for this could be developed by considering how these procedures are analyzed in terms of their complexity theoretic properties, use of data structures, etc..

PAL2(w)

Step 1: Let n be the length of w.

Step 2: For $i = 1$ to n do

(3.8) Let $b_i = a_{n-(i-1)}$.

Step 3: For $i = 1$ to n do

if $a_i \neq b_i$, then output `no` and halt.

Step 4: Output `yes` and halt.

In previous chapters, I have freely employed pseudocode listing of this sort in an attempt to more precisely describe procedures which, like PAL1 and PAL2, have been initially introduced via informal descriptions. As discussed in Chapter 2, this practice is widely employed in the analysis of algorithms and is generally justified on the basis of the fact that pseudocode provides a sort of expressive intermediary between natural language and formal programming languages. This level of detail forces us to adopt an explicit ordering of steps (which may not be intended in a natural language description), but it allows us to explicitly express looping constructions and variable scope which are difficult to clearly indicate in natural language. In this sense, pseudocode serves something like the intermediate level of regimentation into which Quine [111] urged us to render informally stated theories prior to their formalization in first-order logic.

One may, of course, challenge the use of such regimentations as a basis for judging the distinctness of PAL1 and PAL2. For in order to employ pseudocode listings in this manner it appears we must possess a prior understanding of what makes one listing rather than another a faithful expression of an informally described procedure. There are, however, are a number of features which can be cited in favor of the distinctness of PAL1 and PAL2 which do not turn on the details of how this process is carried out. For instance, I have mentioned several times in previous chapters that it is generally possible to grasp an algorithm in the sense of understanding its informal description to the point where one can carrying it out "by hand" without at the same time grasping what function it computes. For this reason, I have also stressed the importance of *proving* that an algorithm A_1 is correct with respect to a mathematical function f before we can use A_1 to learn about the values of f. But since finding such proofs is often non-trivial, it is conceivable that even after having proved A_1

correct with respect to f, one could be presented with another specification of an algorithm which also computes f and yet fail to realize that A_1 and A_2 computed the same function. If such a situation is indeed possible in this instance where we have grasped A_1 and A_2 by distinct pseudocode specifications p_1 and p_2, then it seem that we have little choice but to acknowledge that p_1 and p_2 express *different* algorithms for computing f.[4]

We may now ask whether such a situation can arise with respect to PAL1 and PAL2. In other words, we may inquire as to whether is it possible for someone to recognize that PAL1 as expressed by (3.7) was a correct decision procedure for L_{pal} without thereby coming to realize that PAL2 as expressed by (3.8) had the same property. Note in this regard that informal proofs of the correctness of these procedures are likely to turn respectively on the observations that i) $w = a_1 \dots a_n$ is a palindrome if and only $a_1 = a_n \dots a_{\lfloor n/2 \rfloor} = a_{n-(\lfloor n/2 \rfloor - 1)}$ and ii) $w = a_1 \dots a_n$ is a palindrome if and only if $w = w^R$. But it certainly *does* seem possible that one could recognize the truth of one of these equivalences without recognizing the truth of the other. And for this reason, it also seems possible that since the method employed by PAL2 more closely mirrors our original definition of a palindrome (i.e., a string such that $w = w^R$), one might come to know that this algorithm computed the characteristic function of L_{pal} without thereby realizing the analogous fact about PAL1. And if this is indeed the case, then it seems that we have little choice but to regard PAL1 and PAL2 as distinct algorithms.

Additional support for this conclusion may also be garnered by moving from the intuitive to the formal domain. For note that if we could find some definite computational property like running time complexity, or the use of data structures, which was possessed by one of these algorithms but not the other, then this would also provide a basis for concluding that they were distinct which did not require us to make judgments about the prerequisite epistemic conditions for grasping them. However proceeding in this manner also requires that we have an objective means of distinguishing the properties possessed intrinsically by the algorithms from those which might occur as artifacts of their pseudocode

[4]For note in particular that if p_1 and p_2 expressed the same algorithm, then grasping the correctness proof for A_1 ought to enable us to see that A_2 was correct with respect to f. But this seems not to be the case.

representations. Note, for instance, that when we assess the running time complexity of PAL1 and PAL2 directly in terms of their pseudocode specifications (i.e. by measuring the length of their executions by counting the number of times each informally demarcated step of (3.7) or (3.8) is executed), then then they differ in the sense that PAL1 has exact running time complexity $2 + \lfloor n/2 \rfloor$ whereas PAL2 has exact running time complexity $2 + 2n$. And thus if we were able to additionally argue that this was a genuine difference in procedural properties (as opposed to an artifact of how we have chosen to represent them), then we conclude that PAL1 and PAL2 were distinct by this route as well.[5]

3.3 A provisional model

Having reached the provisional conclusion that PAL1 and PAL2 are indeed distinct algorithms, our next task is to consider what it means to implement these procedures as formal instances of a formal model of computation. As we have seen, this requires formulating a definition for a class of implementations $\mathcal{M}$ which meets the sorts of adequacy conditions discussed in Section 1. Note, however, that relative to the sort of complex mathematical procedures which are routinely used in contemporary mathematical practice, PAL1 and PAL2 are both quite simple, both in their manner of operation and in that they operate over a simple class of data-type – i.e., finite strings of 0s and 1s.

It will be useful to start out by considering a correspondingly rudimentary model. As announced previously, for this purpose I will employ the class $\mathcal{T}$ of deterministic, single-tape, single-head Turing machines. There will, of course, be intuitively acceptable implementations of PAL1 and PAL2 drawn from many other models of computation. In the next chapter we will see that there are straightforward formal reasons why the algorithm realist will want to employ a class as simple as $\mathcal{T}$ as part of his preferred definition of $\mathcal{M}$. Note, however, that we have also seen in Chapter 2 that traditional arguments in favor of Church's

[5]Other complexities enter in here since there is surely some flexibility in the process of regimentation by which we go from descriptions of procedures in natural language to pseudocode specifications. Note, for instance, if we had rendered the instruction contained in our original description of PAL2 to reverse w as a single, non-iterative step, then we would calculate that PAL2 had exact running time $2 + n$. This sort of relativity of computational properties on the details of regimentation makes it difficult to construct a purely formal argument for or against the distinctness of procedures that were initially presented informally. In the sequel I will thus mostly treat PAL1 and PAL2 as if they were introduced by (3.7) and (3.8).

Thesis suggest that $\mathfrak{T}$ is already rich enough to contain at least one intuitively acceptable representative of every effective procedure. And since this class also has the advantages of being simple to define and mathematically streamlined, it will be convenient to start out by examining what happens when we attempt to implement PAL1 and PAL2 relative to $\mathfrak{T}$.

Before we may begin in earnest, it will be useful recall the definition of a Turing machine:

Definition 3.3.1. A *Turing machine* is a tuple $T = \langle K, \Sigma, \delta, \underline{\mathsf{s}}, \underline{\mathsf{h}} \rangle$ such that

i) $K \subset \mathbb{N}$ is a finite set;

ii) Σ is a finite set of symbols;

iii) $s, h \in K$;

iv) $\delta : K \times \Sigma \rightarrow K \times (\Sigma \cup \{\blacktriangleright, \blacktriangleleft\})$.

The members of K are the *states* of T with $s, h \in K$ respectively identified as the *start* and *halt* state. Σ is known as the *alphabet* of T and can be comprised of any finite set of symbols which can be written on its tape. δ is known as the *transition function* of T and maps pairs of states and symbols into actions – i.e., symbols to be marked on the tape at the current head position or instructions to move the head right or left as denoted by $\blacktriangleright, \blacktriangleleft$.

The class $\mathfrak{T}$ which I will consider in this section and the next is defined to be the set of Turing machines T with symbol sets $\Sigma \subseteq \{0, 1, -, *, \#\}$ where $-$ will be used as the blank symbol and $*$ and $\#$ as auxiliary delimiters. Following a convention proposed in [5], we can display such machines as labeled graphs whose vertices correspond to the states in K and whose edges correspond to pairs of symbols and actions. Formally, if $T = \langle K, \Sigma, \delta, \underline{\mathsf{s}}, \underline{\mathsf{h}} \rangle$, the corresponding graph consists of the set of vertices K such that there is an edge between $k_i, k_j \in K$ labeled $\langle \sigma, \alpha \rangle$ (for $\sigma \in \Sigma$, $\alpha \in \Sigma \cup \{\blacktriangleright, \blacktriangleleft\}$) just in case $\delta(k_i, \sigma) = \langle k_j, \alpha \rangle$.

Now consider the two machines S_1 and U_1 depicted in Figures 3.1 and 3.2 which are respectively intended to implement the procedures PAL1 and PAL2. While there is no apparent sense in which these machines can be claimed to be the unique "canonical" representations of PAL1 and PAL2 as Turing machines, a strong heuristic case can be presented that S_1 implements PAL1 and that U_1 implements PAL2. Such observations are premised on the observation that while in neither case is there a one-to-one correspondence between the stages occurring in executions of PAL1 or PAL2 and those executed by S_1 and U_1 on the

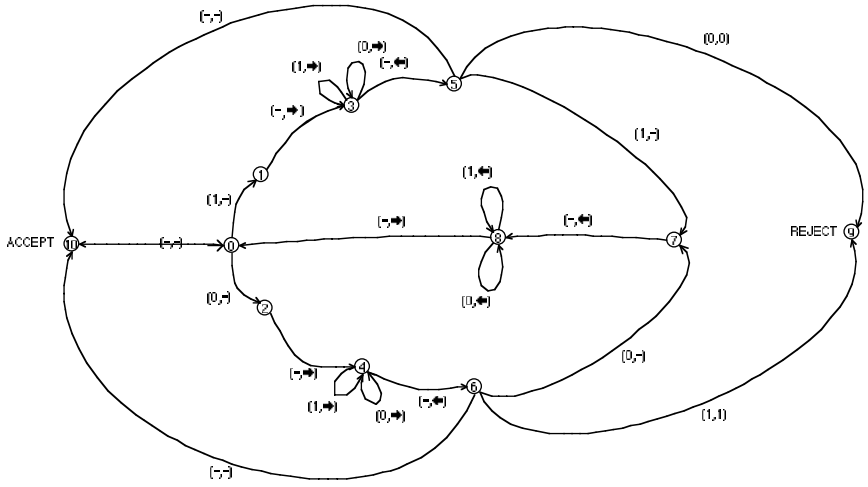

Figure 3.1: The machine S_1 depicted as a graph. When executed with input string w, S_1 indicates that it accepts the input by halting in the state label ACCEPT – i.e. 10 – and that it rejects the input by halting in the state labeled REJECT – i.e. 9. These correspond respectively to output `yes` and `no` to the decision question $w \in L_{pal}$.

same input, such a correspondence can be set up between individual stages of the former and sequences of steps of the latter.[6]

This fact is best appreciated by considering the execution of PAL1 and of S_1 on a particular string $w \in \{0,1\}^*$ – say $w = 1001$. Here we face the initial problem that since PAL1 has been presented as an informal procedure, no precise definition of what it means to execute it on a particular input has yet been given. But based on its pseudocode specification, it is reasonable to describe its mode of operation on the string 1001 as follows:

[6]The term "step" is typically used to denote both an individual instruction in a pseudocode or formal program and the class of occasions on which such an instruction is carried out in the course of executing the procedure denoted by such a listing. In order to distinguish the two usages in the case of informal procedures like PAL1 and PAL2, I will henceforth use the term *stage* to denote an execution of a single *step* understood in the former sense. Since no such confusion arises for non-linguistic means of specifying procedures I will also reserve *step* to denote transitions (as formally defined below) in the execution of a particular Turing machine or other form of implementation.

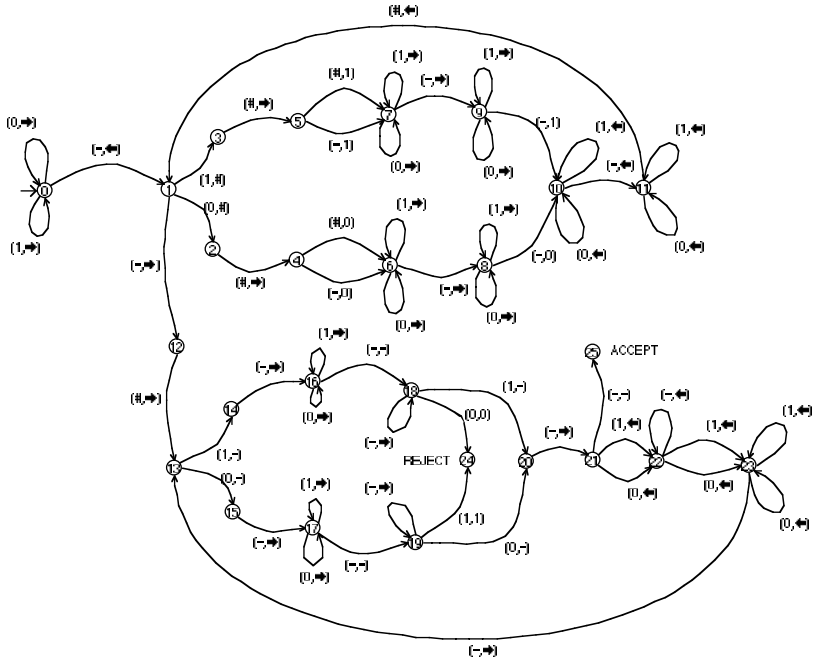

Figure 3.2: The machine U_1 depicted as a graph.

Stage 1: Executing Step 1, PAL1 assigns the value 4 to the variable n (corresponding to the length of w).

Stage 2 : Executing Step 2, PAL1 assigns the value 1 to the loop variable i. It then compares the values $a_1 = 1$ and $a_{n-(i-1)} = a_4 = 1$. Since they are equal, it continues.

Stage 3: Executing Step 2, PAL1 assigns the value 2 to the loop variable i. It then compares the values $a_2 = 0$ and $a_{n-(i-1)} = a_3 = 0$. Since they are equal, it continues.

Stage 4: Executing Step 2, since $i = 2 = \lceil n/2 \rceil$, PAL1 breaks out of the loop and continues with Step 3.

Stage 5: Executing **Step 3**, PAL1 outputs `yes` and halts.

When started on other an otherwise blank tape with its head at the leftmost symbol of the string, the operation of S_1 on w can be described in a similar manner:[7]

[7]It is conventional to denote Turing machine states by natural numbers, and we are considering machines

Step 1: Beginning in its starting state $\underline{0}$, S_1 reads the first symbol (a 1) of w, stores this fact in its state by entering state $\underline{1}$ (as opposed to state $\underline{2}$, which it would have entered had it been reading a 0) and writes over this symbol with a $-$.

Step 2: In state $\underline{1}$, S_1 reads a $-$, moves its head one square to the right, and enters state $\underline{3}$.

Steps 3 - 6: In state $\underline{3}$, S_1 moves its head rightward successively reading 0 or 1 until it reaches a $-$ symbol, signifying that it has moved past the right end of the current tape contents. At this point, it moves its head leftward one square so that it is reading the current rightmost non-blank symbol and enters state $\underline{5}$.

Step 7: In state $\underline{5}$, S_1 does one of three things according to whether it is currently reading a $1, 0$ or $-$: i) if it is currently reading a 1, it overwrites this symbol with a $-$ and enters state $\underline{7}$ (this corresponds to the situation in which the current rightmost symbol of the string matches the previous rightmost symbol which S_1 has overwritten and stored its state, in which case the computation should be continued); ii) if it is currently reading a 0, it enters state $\underline{9}$ (this corresponds to the situation in which the current rightmost symbol of the string does *not* match the previous leftmost symbol which S_1 has stored its state, in which the computation can be halted and the reject state $\underline{9}$ entered); iii) if it is reading a $-$, S_1 enters state $\underline{10}$ (this corresponds to the situation in which w was of odd length and there is thus no symbol to match against the one overwritten in Step 1, meaning that all other symbols have been successfully matched and the string can be should be rejected by entering state $\underline{9}$). Since in the case under consideration, S_1 is reading a 1, it operates as per i).

Step 8: In state $\underline{7}$, S_1 reads a # (corresponding to the symbol with which it has just overwritten the last 1 of w) and enters state $\underline{8}$.

Steps 9-11: In state $\underline{8}$, S_1 moves its head leftward while successively reading 0 or 1 until it reaches a $-$ symbol, signifying that it has moved past the left end of the current

---

over an alphabet which contains numerals. In order to distinguish between states and symbols, I will henceforth adopt the convention of denoting the former as numeral with underscores (as in $\underline{0}$ and $\underline{1}$).

tape contents. At this point, it moves its head rightward one square so that its head is reading the current leftmost non-blank symbol and re-enters state $\underline{0}$.

Step 11: This time in state $\underline{0}$, S_1 is reading a 0. Analogously to Step 1, it stores this symbol in its state, this time by entering state $\underline{2}$ and overwrites it with a $-$.

Step 12: Analogous to Step 2, in state $\underline{2}$ S_1 reads a $-$ and moves its head one square to the right and enters state $\underline{4}$.

Steps 13-15: Analogous to Steps 3-6, S_1 moves its head to right (this time reading 0) until it reaches a $-$ symbol. At this point it moves its head leftward one square so that it is reading the current rightmost non-blank symbol and enters state $\underline{6}$.

Step 16: Analogous to Step 7, S_1 compares the currently scanned symbol (in this, case a 0) with that stored in its state (in this case, also a 0). Since they match, S_1 moves into $\underline{7}$.

Step 17: In state $\underline{7}$, S_1 reads a $-$ (corresponding to the symbol with which it has just overwritten the last 1 of w) and enters state $\underline{8}$.

Step 18: Analogous to Steps 9-11, in state $\underline{8}$, S_1 moves its head rightward and enters $\underline{0}$ since it is already scanning a $-$.

Step 19: In state $\underline{0}$, since it reading a $-$, S_1 moves into the accept $\underline{0}$ and halts.

On the basis of this example, we can see that while there cannot be an exact correlation of the steps taken by S_1 on a given input and the stages of PAL1, there are obvious similarities in, as we might put it informally, "how they work." One way to begin making this observation precise is to note that S_1 and PAL1 make the same sequence of comparisons in determining whether w is a palindrome – i.e., they work inwards by comparing elements at the ends of w, breaking out of this loop and rejecting as soon as a mismatch is found and accepting only if the loop is carried out until all bits have been compared. This observation can be used to give a correspondence between the stages of PAL1 and sequences of steps of S_1.

We must first note that since S_1 accesses bits of w by moving its head over them as opposed to by index, it never explicitly requires access to the length of w. Thus S_1 has no explicit analogue to Stage 1 of PAL1. But it should be evident that the sequence of steps comprised by transitions between states $\underline{0}, \underline{1}, \underline{3}, \underline{5}, \underline{7}, \underline{8}$ and $\underline{0}, \underline{2}, \underline{4}, \underline{6}, \underline{7}, \underline{8}$ are respectively responsible for carrying out iterations of the loop expressed by Step 2 of PAL1 and carried out over the course of Stages 2 and 3 in its execution. In particular, the steps corresponding to the transitions from states $\underline{5}$ and $\underline{6}$ to state $\underline{7}$ are respectively responsible for performing the comparison of bits performed in the antecedent of the condition in the body of this loop. And the other states in these sequences perform a task analogous to arithmetically updating the index i which controls its operation in a sense which may be formalized by considering the position of S_1's head on its tape. Finally, the transition to the accepting state $\underline{10}$ from $\underline{0}$ in the execution of S_1 plays a role analogous to passing from Step 2 to Step 3 in the execution of PAL1. Although the relationships just exhibited between sequences of S_1 and stages in the executions of PAL1 only hold for their executions on the input 1001, it should be clear that not only can an analogous set of correlations between stages and sequence of steps be set up for arbitrary inputs $w \in \{0,1\}^*$, but also that such a correlation is parameterized uniformly in w.

It should also be clear that a similar, albeit somewhat more complex, characterization of the relationship between the operation of PAL2 and that of U_1 may also be given. For instance, it can be seen that states $\underline{0}$-$\underline{11}$ are responsible for constructing the string w^R to the right of w on U_1's tape. This is achieved by first moving to the right end of w (state $\underline{0}$) and then copying bits from the beginning of the string w^R (states $\underline{1}$-$\underline{11}$) using the auxiliary symbol # to keep track of its position in the former string. The bits of w and w^R are then compared from the left, bit by bit, rejecting as soon as a mismatch is found and accepting only when all bits have been successfully compared (states $\underline{13}$-$\underline{23}$). In any given execution of U_1 and PAL1, sequences of states which progress through $\underline{1}$-$\underline{11}$ may be correlated with the stages corresponding to the execution of Step 2 in PAL2. And similarly, sequences of states which progress through the $\underline{13}$-$\underline{23}$ may be correlated with the stages corresponding to the execution of Step 3 in PAL2.

Relative to the informal notion of implementation discussed in chapter 3, these observations suffice to demonstrate that S_1 implements PAL1 and U_1 implements PAL2. For on the one hand, these machines bear exactly the sort of step-by-step operational affinities to PAL1 and PAL2 which was described in Chapter 2.3 as forming the basis of our intuitions about the implementation relation. And on the other, both machines are sufficiently straightforward in their design that it is reasonable to assume that they would have been constructed by a theorist who set out to implement PAL1 and PAL2 as Turing machines. And for similar reasons in each case, it is also reasonable to conclude that S_1 ought *not* to be counted as implementing PAL2 and U_2 ought *not* to be counted as implementing PAL1.

It is, of course, one thing to assert that these facts hold in virtue of the existence (or, respectively, lack of existence) of the sort of informal operational affinities just illustrated, and quite another to provide a formal characterization of a relation which S_1 bears to PAL1 and which U_1 bears to PAL2 in virtue of which these machines implement these algorithms. Note, however, that it is one of the potential virtues of the abstractionist programme that such an account does not need to provided. For according to the abstractionist, PAL1 and PAL2 should not be thought of as "given" directly. The abstractionist rather holds that we ought to think of PAL1 precisely as the algorithm implemented by S_1 and PAL2 as the algorithm implemented by U_1.[8]

But although an abstractionist may thus avoid having to characterize the relationship between implementations explicitly, we have seen that he must still do so implicitly by providing a definition of $\underline{\leftrightarrow}$ which characterizes what S_1 and U_1 have in common with

[8]We might also think that such a theorist can also start out by claiming that PAL1 and PAL2 can be thought of as corresponding to the sorts of thing which are expressed by pseudocode specifications such as (3.7) or (3.8). But I argued in Chapter 2.4 that in order to work out this alternative, we must assume that we have a prior informal understanding of what it means to carry out a procedure by interpreting the instructions appearing in such specifications. This problem may be circumvented by presenting formal programs Π_1 and Π_2 which are also claimed to express these algorithms in a manner which more directly reflects their structure than S_1 or U_1. As I argued there, however, programs do not express algorithms intrinsically (i.e., in virtue of some prior notion of "procedural meaning" which can be assigned to their constituents) but do so (if at all) in virtue of an operational semantics for the language over which they are defined. Relative to such a semantics, Π_1 and Π_2 would determine machines M_1 and M_2 drawn from some model of computation $\mathcal{M}$. If Π_1 and Π_2 were constructed over an appropriately "high-level" programming language, it might be that M_1 and M_2 differ from S_1 and U_1 in some manner which made them appear better suited to implementing PAL1 and PAL2. For instance, they might employ mathematical operations which more closely resembled those appearing in (3.7) and (3.8). Note, however, that regardless of the individual features of these machines, the general problem of accounting for their relationship with PAL1 and PAL2 will still be the same as that which we face in with respect to S_1 and U_1.

other machines we also take to be implementations of PAL1 and PAL2. The first fixed points of data we have to constrain such a definition follows from the conclusion that PAL1 and PAL2 are distinct algorithms. For note that once we have agreed to the statements which would be formalized in the language discussed in the prior Section as PAL1 $\neq$ PAL2 and $S_1 = imp(\text{PAL1})$ and $U_1 = imp(\text{PAL2})$, it follows from (BP) that $\underline{\leftrightarrow}$ must be defined so that it does not hold between S_1 and U_1 – i.e. so that $S_1 \underline{\nleftrightarrow} U_1$.

But negative data of this sort compromise only one class of constraints to which a proponent of abstractionism must be sensitive in formulating an extensionally adequate definition of $\underline{\leftrightarrow}$. In order to provide a definition which is even minimally responsive to our intuitions about procedural identity, he must also consider the classes of Turing machines $S_2, S_3, \ldots$ and $U_2, U_3, \ldots$ which we would also intuitively accept as implementing PAL1 and PAL2. And for these classes he would have to show that for each pair S_i, U_j we have $S_i \underline{\nleftrightarrow} U_j$ and that for each pair S_i, S_j and U_i, U_j that $S_i \underline{\leftrightarrow} S_j$ and $U_i \underline{\leftrightarrow} U_j$.

In order to see that PAL1 and PAL2 have open classes of implementations of this sort, it suffices to observe that the machines derived by making certain trivial modifications of either S_1 or U_1 presumably bear equal title to serve as implementations of these algorithms. For instance, even the most cursory reflection on our actual computational practices suggests that we will generally accept two Turing machines $T = \langle K, \Sigma, \delta, s, h\rangle$ and $T' = \langle K', \Sigma', \delta', s', h'\rangle$ as "essentially the same" if they differ only in the labeling of states – i.e. if $K = K'$, $\Sigma = \Sigma'$ and there exists a permutation $\rho : K \to K$ such that $\rho(s) = s'$, $\rho(h) = h'$ and for all $k_1, k_2 \in K$, $\delta(k_1, \sigma) = \langle k_2, \alpha\rangle$ if and only if $\delta'(\rho(k_1), \sigma) = \langle \rho(k_2), \alpha\rangle$. It thus follows that $\underline{\leftrightarrow}$ should be defined so that it holds between any two machines T, T' which are related in this sense.

It should not be particularly surprising that there exist a number of similar transformations of Turing machines which preserve, at least intuitively, the algorithm which they implement. For instance, T' might be derived from T by adding or deleting states or transitions which (provably) are not accessed for any input. Similarly, T' could be derived from from T by adding or deleting sequences of transitions which (provably) counteract one another (such as moving the head left and then immediately moving to the right, writing over a symbol and immediately writing it back, etc.). And additionally T might be derived

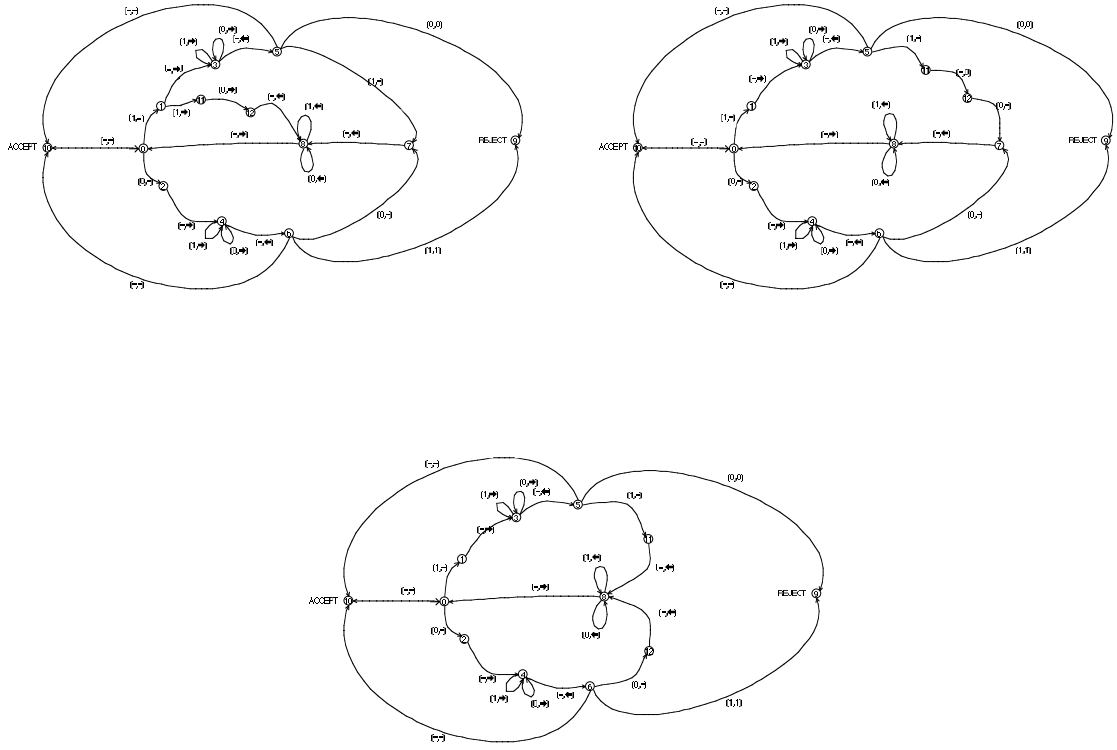

Figure 3.3: The machines S_2, S_3 and S_4 depicted as graphs.

from T' by splitting (or combining) the function performed by one (or two) state(s) and its (their) incoming and outgoing transitions into two (one) state(s).[9] Examples of machines S_2, S_3 and S_4 derived from S_1 in each of these ways are given in Figure 3.3. Since the transformations applied to S_1 to yield each of these machines is straightforward, it should not also not be surprising that in each case an argument similar to the ones above can be constructed leading to the conclusion that each of S_2, S_3 and S_4 implement Pal1. And on this basis, it also seems as if we ought to accept as additional data constraining the definition of $\underline{\leftrightarrow}$ that $imp(S_2) =$ Pal1, $imp(S_3) =$ Pal1 and $imp(S_4) =$ Pal1.

[9]These observations recapitulate a claim made by Kreisel [69] which was discussed briefly in Chapter 2. Recall that in arguing for a form of Extended Church's Thesis, Kreisel claims that Turing's work establishes that "to each mechanical rule or algorithm is assigned a more or less specific programme, *modulo trivial conversions*, which can be seen to define the same computation process as the rule" (my italics). Two questions that arise in the current context (and which are not even broached by Kreisel) are as follows: 1) is it possible to formulate a structural criterion which distinguishes trivial from non-trivial conversions of machines?; and 2) given machines T and T', can we effectively determine whether T' has been derived from such a conversion? As we will shortly see, I believe that both questions should be answered in the negative.

Before investigating further how $\underline{\leftrightarrow}$ can be defined so as to accommodate these facts, it will also be useful to consider a different class of implementations of PAL1 and PAL2 which cannot be seen as arising through straightforward modifications of S_1 and U_1. Consider, for instance, the machine S_5 depicted in Figure 3.4. As the reader is invited to verify, if we suppose that the current non-blank contents of the tape are $a_1 \ldots a_n$, then the operation of this machine may be described as follows:

1) Over the course of transitions mediated by states $\underline{0}$ - $\underline{14}$, a_1 is "pushed" leftward by swapping its value successively with that of each symbol to its right.
2) When the right end of the non-blank portion of the tape is reached (as signaled by the fact that the head is reading a $-$), S_5 moves its head one square to the left, entering state $\underline{15}$.
3) S_5 then decides whether the currently scanned symbol (which will correspond to a_1) is a 0 – in which case it is overwritten with a $-$ and state $\underline{16}$ is entered – or a 1 – in which case it also overwritten with a $-$ and state $\underline{17}$ is entered.
4) The head is moved leftward again and the symbol just "remembered" in the state is compared over the course of the transition from states $\underline{16} - \underline{18}$ (or $\underline{17} - \underline{19}$) or with the currently scanned symbol (which now corresponds to a_n).
5) If a_1 and a_n disagree, then S_5 rejects by entering state $\underline{23}$.
6) If they agree, a_n is overwritten with a $-$ and S_5's head is returned to the leftmost non-blank symbol (over the course of transitions mediated by states $\underline{20}$ and $\underline{21}$) and the process is started over again for the string $a_2 \ldots a_{n-1}$.

It should be evident that the same sort of heuristic case given above to argue that S_1 is an implementation of PAL1 can also be used to argue that S_5 implements this procedure. For note that S_5 may also be described as "working inwards" on $w \equiv a_1 \ldots a_n$ in the sense that it makes comparisons in the order $a_1 \stackrel{?}{=} a_n, a_2 \stackrel{?}{=} a_{n-1}, \ldots, a_{\lfloor n/2 \rfloor} \stackrel{?}{=} a_{\lfloor n/2 \rfloor + 1}$. And since this is the same order in which symbols are compared during the executions of PAL1 it is possible to set up the same sort of correlations between stages in operation of PAL1 and sequences of steps in the operation of S_5 on the same input.

There is, of course, a clear difference in how we might describe the operation of S_1 and S_5 from the perspective of Turing machines. For note that S_1 stores the contents of the

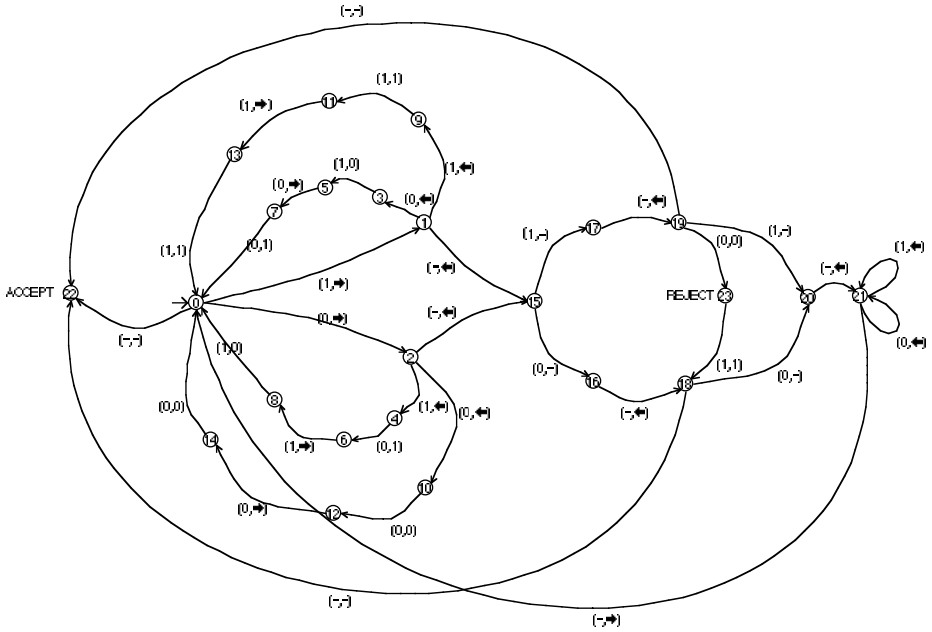

Figure 3.4: The machines S_5 depicted as a graph.

current leftmost symbol a_i in its state by selectively entering either state $\underline{1}$ or $\underline{2}$ and then moving to the right end of the string. But in order to achieve what from the outside we would describe as the same function, S_5 transports this symbol to the right end of the tape and compares it directly with $a_{n-(i-1)}$. These different operations clearly require different sorts of implementations in the sense of specific Turing machine configurations. And for this reason, there is no apparent sense in which S_5 can be viewed as a "trivial" variant of S_1 (or vice versa).

We may thus ask whether the existence of "machine level" differences of this sort ought to influence our decision to view either of S_1 or S_5 as implementations of PAL1. To see that they should not, it is sufficient to observe that although our initial specification of PAL1 tells us that during the $i+1$st stage of the execution of this procedure, symbols a_{i-1} and $a_{n-(i-1)}$ are compared, it does not tell us anything about *how* this comparison is transacted. In particular, the pseudocode specification (3.7) does not tell us if such comparisons are to be made "directly" (i.e. by accessing a_{i-1} and $a_{n-(i-1)}$ by index) or in some other manner

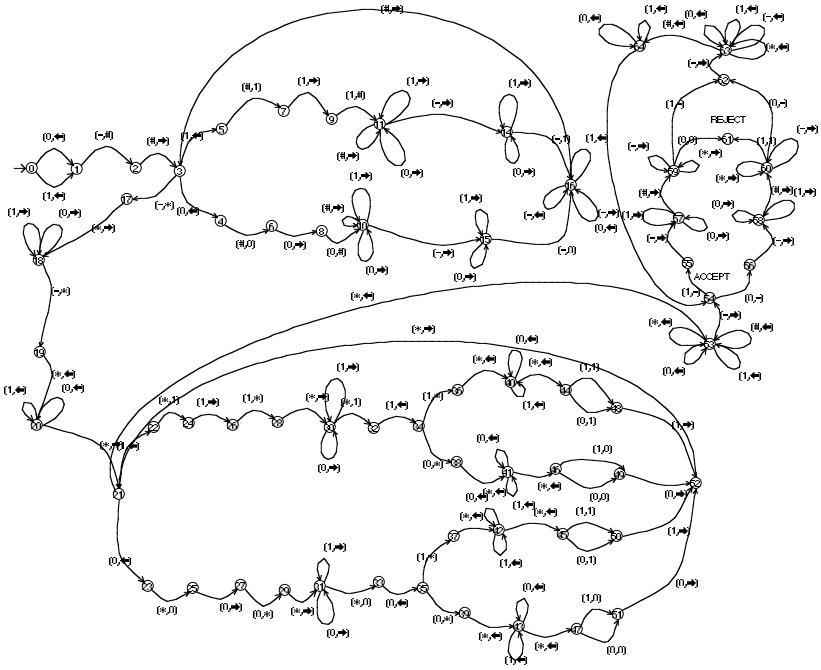

Figure 3.5: The machines U_2 depicted as a graph.

which is itself procedurally mediated. And thus there is no principled basis on which one might claim that the mechanism employed by one of S_1 or S_5 to achieve this purpose is a more faithful or transparent implementation of Step 2 of PAL1 than that employed by the other.

As one might expect, it is also possible to construct a machine U_2 which bears a similar relationship to U_1 as S_5 does to S_1. Such a machine is depicted in Figure 3.5. Although the details are more involved, an account similar to that given in the preceding paragraphs can be told about the relationship between U_2 and PAL2. Again, the reader is invited to investigate this for himself, but the high points are as follows: 1) over the course of the transitions mediated by states 0 - 16, U_2 makes a copy w' of w to its right on the tape using the auxiliary symbol # as a divider to keep track of its work; 2) during the transitions mediated by states 21 - 52, U_2 reverses w' so as to obtain a string w'' which equal is to w^R using the auxiliary symbol $*$ as a divider; and 3) during the transitions mediated by states 53 - 64, U_1 compares w and w'' bit by bit from left to right, rejecting as soon as a mismatch

is found and accepting only after all bits have been compared without finding a mismatch.

Although U_2 looks a good deal more complicated than U_1, it may again be argued that it is an equally faithful implementation of PAL2. In this case, the salient difference between the machines is how they perform the function specified by Step 2 of PAL2 – i.e., For $i = 1$ to n { Let $b_i = a_{n-(i-1)}$ }. In the case of U_1, this is achieved by copying bits from the right end of w to the left end of a new string w', an arrangement which directly yields $w' \equiv w^R$. In the case of U_2, it is achieved by i) copying w bit by bit to form a new string w' immediately to its right, and then ii) reversing the bits of w' to yield a string $w'' \equiv w^R$. Clearly both subprocedures achieve the intended result of Step 2, i.e., that of producing a string $b_1 \dots b_n \equiv a_{n-(i-1)} \dots a_1$ in another location which can be compared with $a_1 \dots a_n$. The operational difference between them thus appear to be implementation-dependent. Consequently, there again appears to be no way of arguing that U_1 is an implementation of PAL2 which does not also function to show that U_2 is an implementation of this procedure and conversely.

We can now collect together the constraints which the examples surveyed thus far place on the definition of $\underline{\leftrightarrow}$. The foregoing discussion has been aimed at establishing the following "positive" facts:

(3.9) i) PAL1 and PAL2 are distinct algorithms. Thus for no machine T can we have $imp(T) =$ PAL1 and $imp(T) =$ PAL2.

ii) For $i \in \{1, 2, 3, 4, 5\}$, $imp(S_i) =$ PAL1.

iii) For $i \in \{1, 2\}$, $imp(U_i) =$ PAL2.

From this it follows that if an a algorithmic realist wishes to identify PAL1 and PAL2 with mathematical objects via (BP) in the manner described in Section 1, the definition of $\underline{\leftrightarrow}$ which he proposes must satisfy the following "positive" requirements

(3.10) i) For all $i, j \in \{1, 2, 3, 4, 5\}$, $S_i \underline{\leftrightarrow} S_j$.

ii) $U_1 \underline{\leftrightarrow} U_2$,

as well as the following "negative" requirement

(3.11) For all $i \in \{1, 2, 3, 4, 5\}$ and $j \in \{1, 2\}$, $S_i \underline{\nleftrightarrow} U_j$.

Informally described, requirements (3.10i,ii) tells us that the definition of $\underline{\leftrightarrow}$ must be sufficiently broad so as to allow Turing machines which implement certain portions of PAL1 and PAL2 in different ways to bisimulate one another. As one might guess, this constraint is particularly significant with respect to the need to accommodate $S_1 \underline{\leftrightarrow} S_5$ since the operation of these machines is based on different methods for making comparisons at opposite ends of a string. Similar remarks apply to the relationship between U_1 and U_2 with respect to methods of reversing strings. Requirements (3.10i,ii) thus appear to lobby for a definition of bisimulation which allows machines which operate quite differently from one another in a step-by-step sense to still bisimulate one another.

But in adopting a definition which can accommodate these sorts of constraints, there is an obvious risk of defining $\underline{\leftrightarrow}$ so broadly that it will hold between implementations which it should not. In particular, in defining this relation so that it can link machines which differ in structure as radically as S_1 and S_5 or U_1 and U_2 we run the risk of sanctioning "unintended" bisimulations – i.e. pairs of machines like S_1 and U_1 which we would not informally accept as implementing the same algorithm. The necessity of defining $\underline{\leftrightarrow}$ so that it satisfies the negative requirements summarized in (3.11) thus constitutes another sort of constraint on what should be counted as an extensionally adequate definition of bisimulation.

Any definition of $\underline{\leftrightarrow}$ which satisfies even these minimal standards of extensional adequacy must thus balance factors which militate for a "coarse grained" equivalence relation over implementations with those which militate for a "fine grained" one. The danger, of course, is that it will turn out that there is no stable middle ground in the sense that any reasonable definition of bisimulation will turn out to violate either one of the positive constraints (by virtue of being too narrow) or one of the positive ones (by virtue of being too broad). It will be my ultimate contention that is this not only how matters work out in the particular case of PAL1 or PAL2, but also that this situation is typical even when we consider more sophisticated algorithms and more sophisticated classes of implementation relative which we might attempt to define them.

But before we can reach a position where it is possible to argue for either conclusion directly, much more needs to be said about what should count as an intensionally adequate

definition of bisimulation. And in this regard, it seems that we have little choice but to begin by directly consulting our intuitions about what it means for particular pairs of implementations to be computationally equivalent to one another. For note that since there appears to be no means of directly fixing the class of machines which we would intuitively accept as implementing a given algorithm A, it will not suffice to start out by attempting to define $\underline{\leftrightarrow}$ simply as the relation which holds between the machines in such a class. Rather we must seek to extrapolate an abstract characterization of such a relation which by examining individual examples. And while I will ultimately argue that there can be no stable end point to such an inquiry, we are now at least in a good position to scout out possible alternatives.

3.4 Defining bisimulation

3.4.1 On bisimulation

In attempting to define $\underline{\leftrightarrow}$ over $\mathcal{T}$ so as to account for the positive constraints (3.10) as well as the negative constraints (3.11), we might start out by asking whether we can recognize any obvious invariants in the structures of the machines comprising the class $\boldsymbol{T}_1 = \{S_1, S_2, S_3, S_4, S_5\}$ which distinguish them from those in the class $\boldsymbol{T}_2 = \{U_1, U_2\}$. For instance, since S_2, S_3 and S_4 were all introduced as variants of S_1, one might initially think that it is possible to do this by simply enumerating the allowable transformations among members of $\mathcal{T}$ which preserve various "local" relationships among their states. But even a cursory examination of the formal properties of $\mathcal{T}$ demonstrates that this approach is futile. For on the one hand, our intuitions seem to provide no clear answer to the host of technical questions which arise when we attempt to precisely specify the relationships which, say, S_2, S_3 and S_4 bear to S_1.[10]And on the other, many of the questions about the relationships between machines which can be taken to formalize basic intuitions are

formally undecidable.[11]

There is, however, an even greater informal obstacle to accounting for the full range of positive bisimulation relationships exhibited even in (3.10) by directly cataloging allowable "algorithm preserving" structural transformations between machines. Note in particular that not only was S_5 not introduced as deriving from such a transformation from S_1, but it seems difficult to imagine how even a sequence of transformations of the sort considered above could lead from one machine to the other. For recall that S_5 can be informally described as implementing an intensionally distinct method for performing one of the basic operations in terms of which PAL1 was initially specified. And for this reason alone, it also seems doubtful that we can ever hope to account for the precise relationship which the operation of this machine bears to that of S_1 in terms of localized relationships among states. Also recall that the goal of the algorithmic realist is to define $\underline{\leftrightarrow}$ so that the machine in $\boldsymbol{T} = \boldsymbol{T}_1 \cup \boldsymbol{T}_2$ satisfies (3.10) and (3.11), but also so that this relation tracks our intuitions about algorithmic identity across all members of $\mathcal{T}$. And since all of the suggestions considered thus far are specifically tied to the machines in the former class, it therefore seems that a more general methodology is called for.

There is, however, another significant set source of intuitions to which we can turn in attempting to define $\underline{\leftrightarrow}$. For recall that in order to even define the classes $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, a positive account had to be given of why the various machines they contain should

---

[11] Consider, for instance, the relationship between machines S_1 and S_4. Recall that when S_1 reaches the end of its input string, it overwrites the last symbol and then moves into state 7, regardless of whether this symbol was a 0 or a 1 and then moves left enters state 8. On the other hand, when S_4 overwrites the last symbol of its input, it moves into 11 if this symbol was a 1 and into 12 if it was a 0, before moving into the state s_8. We might informally describe this as an instance in which s_7 of S_1 had been "split" between the s_{11} and s_{12} of S_4. But in attempting to make this observation precise, we are forced to ask a variety of additional questions such as the following: what does it mean to say that the "computational function" of one state is distributed among several? how many states are allowable as the product of such a splitting? must the computational state entered after such a splitting be identical to its un-split counterpart? Although it is possible to answer these questions in a manner which fits the needs of particular examples, it seems doubtful that there is a means of doing so which non-stipulatively generalizes to even readily foreseen cases.

[12] This includes, for instance, the question of determining, given a Turing machine T and one of its states k, whether there exists an input string w such that k is accessed by T during its computation on a given input w. It also includes the question of given two machines T and T' containing initial states s and s' whether for all inputs u, the tape contents of T after executing transitions between states $k_1 \ldots k_n$ will be the same as that of T' after executing the transitions between states $k'_1 \ldots k'_{n'}$. Note that specific instances of these questions which must be answered in order to conclude that S_2 is derived from S_1 by the addition of "dummy" states which are never accessed, or whether S_3 is derived from S_1 by states which give rise to mutually counteracting transitions.

respectively be taken to implement either PAL1 or PAL2. The account provided above was based on the observation that in the case of each of the machines in these classes it was possible to correlate sequences of steps in their execution with individual stages in the operation of PAL1 or PAL2 which we would informally characterizer as playing the same functional role in the operation of the algorithm as a whole.

This observation itself is not immediately useful in helping us define $\underline{\leftrightarrow}$ over $\mathfrak{T}$ because it relates structural features of the machines in, say, $\boldsymbol{T}_1$ with those of PAL1 instead of with one another. In addition to this, it is also unclear how (or even if) we may allowably formalize the operation of algorithms which, like PAL1, have been specified informally in a manner that allows us to non-circularly identify the "functional role" of their stages. However, recall that the abstractionist's background view is that algorithms are not given to us directly (e.g. by informal linguistic descriptions) but rather as the denotation of expressions of the form "the algorithm implemented by M" where M is some explicitly defined implementation.

On this basis, one might reasonably conjecture that it was possible to use the sorts of informal operational affinities between the members of Turing machines and algorithms which were used to argue for the relationships summarized in (3.10) to define a relation which held directly between the members of $\boldsymbol{T}$. For essentially the same reasons explored in Chapter 2.4, this observation provides a background motivation for attempting to take $\underline{\leftrightarrow}$ to be some form of inter-simulation relation between Turing machines. To a first approximation, such a relation should relate $T_1, T_2 \in \mathfrak{T}$ just in case there exists a relation S linking the states of T_1 and T_2 such that if S relates states μ and ν, any transition between states μ and μ' in T_1 is mirrored by a transition between states ν and ν' of T_2 such that S relates ν and ν', and conversely for transitions of T_2 with respect to T_1.

An initial definition of bisimulation was given in Chapter 2 which was meant to capture these intuitions relative to a very general class of computational models I referred to as *transition systems*. I will argue in Chapters 4 and 5 that all models of computation can be assimilated to this definition, albeit with varying degrees of transparency and mathematical naturalness. And since it turns out to be relatively straightforward to recast Turing machines as transitions systems, it will be useful to explicitly fix a means of doing so that

we may give a definition of bisimulation for $\mathcal{T}$ which generalizes on our previous definition.

To this end, I will repeat here the definition of (deterministic) transition system from Chapter 2.3.

Definition 3.4.1. A *transition system* M is a septuple $\langle X, Y, St, \Delta, H, in, out\rangle$ whose components satisfy the following properties:

i) X and Y are sets respectively known as the *input* and *output* sets of M

ii) St is a finite or infinite set consisting of the *states* of M;

iii) $in : X \to \Sigma$ is the *input function* of M;

iv) $\Delta : St \to St$ is the *transition function* of M;

v) $H \subseteq St$ is the set of *halting states* of M;

vi) $out : St \to Y$ is the *output function* of M

Recall that relative to this definition we defined the execution of M on $x \in X$ to be the finite or infinite $\sigma_1(x), \sigma_2(x), \ldots$ defined by $\sigma_1(x) = in(x)$ and $\sigma_{i+1}(x) = \delta_A(\sigma_i(x))$ if this value is defined and $\sigma_{i+1}(x) = \sigma_i(x)$ otherwise. I will denote this sequence by $\vec{\sigma}(x)$, which will be finite or infinite depending on whether M halts on x. We may also define $len_A(x)$ to be $k + 1$ where k is least such that $\delta_M(\sigma_k(x)) = \sigma_k(x)$ and is undefined if such a k does not exist. And finally, we defined the function computed by M on X to be given by $App_M(x) =_{df} out(\sigma(x)_{len_M(x)})$ if $len_M(x)$ is defined and undefined otherwise.

Comparing (3.4.1) with the standard definition of Turing machine, it should be clear how individual Turing machines can be recharacterized as transition systems. In particular, given any machine $T \in \mathcal{T}$ which we wish to view as a decider for the language L as follows, we can define the transition system M_T via the following definition:

Definition 3.4.2. Given a machine $T = \langle K, \Sigma, \delta, \underline{\mathtt{s}}, \underline{\mathtt{h}}\rangle$, and a language $L \subseteq \{0,1\}^*$ we define $M_T = \langle X_T, Y_T, St_T, \Delta_T, H_T, in_T, out_T\rangle$ as follows:

i) $X_T = \{0,1\}^*$;

ii) $Y_T = \{\mathtt{yes}, \mathtt{no}\}$;

iii) $St_T = \{\langle \underline{\mathtt{k}}, x, y\rangle \ : \ x, y \in \Sigma^\omega \text{ and } \underline{\mathtt{k}} \in K\}$;

iv) $\Delta_T = \{\langle \mu, d(\mu)\rangle \ : \ \mu \in St_T\}$ where if $\mu = \langle \underline{\mathtt{k}}, x, y\rangle$ for $x \equiv ax'$ and $y \equiv by'$ for $a, b \in \Sigma$,

$x', y' \in \Sigma^\omega$ and $\delta(\underline{\mathtt{k}}_1, b) = \langle \underline{\mathtt{k}}_2, \alpha \rangle$ then $d(\mu) = \begin{cases} \langle \underline{\mathtt{k}}_2, x, cy' \rangle & \text{if } \alpha = c \\ \langle \underline{\mathtt{k}}_2, b \cdot a \cdot x', y' \rangle & \text{if } \alpha = \blacktriangleright \\ \langle \underline{\mathtt{k}}_2, x', a \cdot b \cdot y' \rangle & \text{if } \alpha = \blacktriangleleft \end{cases}$

v) $H_T = \{\langle \underline{\mathtt{h}}, x, y \rangle \ : \ x, y \in \Sigma^\omega\}$;

vi) $in_T(w) = \langle \underline{\mathtt{s}}, (-)^\omega, w(-)^\omega \rangle$ where $w \in \{0,1\}^*$ and $(-)^\omega$ denotes an infinite sequence of blank symbols;

vii) $out_T(\langle \underline{\mathtt{k}}, u, v \rangle) = \mathtt{yes}$ if $\underline{\mathtt{k}} = \underline{\mathtt{h}}$ and is labeled `yes` and $out_T(\langle \underline{\mathtt{k}}, u, v \rangle) = \mathtt{no}$ if $\underline{\mathtt{k}} = \underline{\mathtt{h}}$ and is labeled `no`.

Several points should be flagged about this definition. First note that the class of states St_T of M_T are taken to be tuples of the form $\mu = \langle \underline{\mathtt{k}}, x, y \rangle$ for $\underline{\mathtt{k}} \in K$ and $x, y \in \Sigma^\omega$. These structures are thus what are sometimes referred to as Turing machine *configurations* – i.e., complete temporal cross sections of a machine's state including a specification of its current state (given by $\underline{\mathtt{k}}$), the contents of its infinite two-way tape (given by its the infinite strings x and y), and its head position. The former part is encoded directly by the first component of μ, while the latter two parts are encoded indirectly in the second and third components of μ subject to the following conventions: 1) x corresponds to the contents of T's tape properly to the left of its head read in reverse order, i.e., such that the first symbol of x is directly to the left of the head and its last symbol is the leftmost non-blank symbol on the tape; 2) if $y \equiv by'$, then $b \in \Sigma$ is the symbol currently scanned by T's head and y' corresponds to the tape contents properly to the right of T's head. For the most part, I will continue to refer to the members of St_T as transition system *states*, despite the clash of names with the notion of Turing machine state (which encode neither head position or tape contents). However, when there is potential for confusion, I will employ the terms *local state* to denote a state of T (i.e., a member of K) and *global state* to denote a state of M_T (i.e., a member of St_T). I will also continue to employ the symbols $\underline{0}, \underline{1}, \ldots$ (possibly with subscripts indicating the machine in which they occur) to denote local states and use variables $\mu, \nu, \ldots$ to refer to the global states.

Note that as defined by Definition 3.2, the transition system M_T associated with any Turing machine T will have infinitely many global states μ, each of which will correspond

to an infinite structure. But the course of execution of M_T is still finitely determined in the sense that the value of the transition function Δ_T applied to μ is determined by a finite amount of "local" information and effects only a finite "local" change of state in the following sense: for all states $\mu_1 = \langle \underline{\mathsf{k}}, a_1 x_1, b y_1 \rangle$, $\mu_2 = \langle \underline{\mathsf{k}}, a_2 x_1, b y_2 \rangle$, if there exist $\mu_3 = \langle \underline{\mathsf{j}}, a_3 x_3, b_3 y_3 \rangle$ and $\mu_4 = \langle \underline{\mathsf{l}}, a_4 x_4, b_4 y_4 \rangle$ such that $\langle \mu_1, \mu_3 \rangle, \langle \mu_2, \mu_4 \rangle \in \Delta_T$, then $\underline{\mathsf{j}} = \underline{\mathsf{l}}$ and either $a_3 = a_4$ or $b_3 = b_4$. Also recall that according to our convention for interpreting machines $T \in \mathfrak{T}$ as deciders for languages over strings $w \in \{0,1\}^*$, T will always be started with a state of the form $\langle \underline{\mathsf{s}}, (-)^\omega, w \cdot (0)^\omega \rangle$. Thus at any point during its computation, it will be in a state of the form $\langle \underline{\mathsf{k}}, (-)^\omega \cdot u^R, v \cdot (-)^\omega \rangle$ for *finite* string u, v such that uv is the contents of its tape from the leftmost to its rightmost non-blank symbol. As a notational expedient, we can thus represent the elements of St as strings of the form $u\underline{\mathsf{k}}v$ for $u, v \in \Sigma^*$ (where the blank symbol may occur in either u or v) subject to the convention that the head is assumed to be reading the first symbol of v, or if $v = \epsilon$, the head is scanning a blank symbol and all symbols to its right are also blank.

We may now also begin to refine the informal characterization of inter-simulation given above so to provide a formal definition of what it means for two transition systems M_1 and M_2 to be computationally equivalent. Recall that the idea such a definition is supposed to formalize is that M_1 and M_2 ought to be held to be computationally equivalent (or more generally to "work the same way") just in case these systems have similar step-by-step behavior for all inputs on which they operate. In the strictest sense, this can be taken to mean that any state $\mu \in St$ must be correlated with a state in $\nu \in St$ which is claimed to be equivalent and conversely, although we will consider several ways in which this requirement might be relaxed below.

The formal notion of inter-simulability which I will employ to attempt to formalize the sort of computational equivalence relation just described is a variant of the notion of *bisimulation* which is a formal of structural equivalence applied in process algebra and modal logic. In order to simplify notation, it will we useful to henceforth write $\xi \xrightarrow{i} \xi'$ to denote that $\langle \xi, \xi' \rangle \in \Delta_i$ (for $i \in \{1,2\}$). A so-called *bisimulation relation* between two transition systems M_1 and M_2 with same class of states St can be generally characterized as a relation $S \subseteq St \times St$ which satisfies the two following properties:

(3.12) a) If $\mu \xrightarrow{1} \mu'$ and $\mu S \nu$ for some $\nu \in St$, then there exists $\nu' \in St$ such that $\nu \xrightarrow{2} \nu'$ and $\nu S \nu'$. And conversely, if $\nu \xrightarrow{2} \nu'$ and $\mu S \nu$ for some $\mu \in St$, then there exists a state $\mu' \in St$ such that $\mu \xrightarrow{1} \mu'$.

b) There is an equivalence relation $R \subseteq St \times St$ whose definition is fixed independently of M_1 and M_2 such that if $\mu S \nu$ then the states μ and ν bear R to one another.

I will call these requirements respectively the *transitional* and *representational* requirements on a bisimulation relation. The transitional requirement may be seen as requiring that S is large enough to ensure that all transitions M_1 are mirrored by some transition in M_2 and conversely, that all the transitions of M_2 are mirrored by transitions of M_1. The representation requirement is most naturally understood relative to the assumption that the non-relational properties of the states of M_1 and M_2 (i.e., those properties that the states μ and ν possess in virtue of the definition of these classes as opposed to the transitions in which they figure) encode information about the mathematical structures on which these systems operate. In this case, the relation R should be formulated so that states μ and ν are R-related just in case they correspond to the same intermediate step in an appropriately abstract description of the executions of M_1 and M_2. (For instance, in the case of the machines in $\boldsymbol{T}$, we might attempt to define R so that $\mu R \nu$ held just in case μ and ν contained similar fragments of the initial string w.)

Although (3.12) goes some distance towards constraining the properties which our prior investigations suggest that a definition of $\underline{\leftrightarrow}$ must possess in order to ensure that $\underline{\leftrightarrow}$-related transition systems can plausibly be taken to implement the algorithm, it remains for an algorithmic realist to transform these conditions in a precise definition of bisimulation. This is straightforward in the case of the transitional requirement which can be stated more succinctly as consisting of the requirement that

(13a) $\forall \mu \forall \mu' \forall \nu [(\mu \xrightarrow{1} \mu' \wedge \mu S \nu) \rightarrow \exists \nu' (\mu' S \nu' \wedge \nu \xrightarrow{2} \nu')]$

This condition is often referred as the *forth* direction in a definition of bisimulation since it expresses how transitions in M_1 must be mirrored by transitions in M_2. The symmetric *back* condition may be expressed as

(13b) $\forall\nu\forall\nu'\forall\mu[(\nu \xrightarrow{2} \nu' \wedge \mu S\nu) \rightarrow \exists\mu'(\mu' S\nu' \wedge \mu \xrightarrow{1} \mu')]$

a definition which is intended to reflects the fact that transitions in M_2 should be mirrored by transitions in M_1.

Although we will find reasons to substantially modify (3.13a,b), these condition express in as straightforward a form as possible the intuition that $\underline{\leftrightarrow}$-related implementations ought to operate similarly in the sense of making analogous transitions between statements on all inputs. But it is difficult to find as straightforward a means of formalizing the representational requirement. This is not surprising, since the definition of transition system imposes no conditions on the form that the classes of states of such a system may take, meaning that the sort of equivalence relation which is appropriate will vary at least with the general class of transition systems from which M_1 and M_2 are drawn.

To illustrate this fact, note that in the most familiar settings in which the notion of bisimulation is applied is that in which the transition systems in question correspond to models $\mathfrak{M}_1$ and $\mathfrak{M}_2$ of propositional modal logic. In this case, the members of St can be taken to correspond to valuations v and v' on a common class $\boldsymbol{A} = \{A_i \ : \ i \in \mathbb{N}\}$ of atomic propositions. And in this setting, the relation R is commonly taken to be that of agreement on $\boldsymbol{A}$ – i.e. $R(v, v')$ if and only if for all $A \in \boldsymbol{A}$, $v(A) = v'(A)$.[13] However in most of the cases which we want to study, the structures corresponding to states will be more complex. One important tradition in theoretical computer science suggests that states of this sort can be taken to be first-order structures for a fixed mathematical signature $\mathfrak{L}_M$.[14]In this setting, there are a variety of well known notions of structural equivalence which we might take for R – e.g. isomorphism, first- or higher-order equivalence, n-ary back and forth

equivalence, etc.

But as formalisms needed to describe computational states are largely orthogonal to generalizations we wish to state about transition systems, it is also evident that none of these well known "off the shelf" relations has the right properties to characterize the classes of states which we will want to relate. Thus despite the fact that it would be highly desirable for an algorithmic realist to be able to characterize this relation abstractly – i.e. in a manner which does not rely on the details of the class of implementations from which M_1 and M_2 are drawn – for the moment, I will leave R as a free parameter in the definition of bisimulation which, for the record, I will fix as follows:

Definition 3.4.3. Let M_1 and M_2 be transitions systems defined as above and R an equivalence relation over $St \times St$. A relation $S \subseteq St \times St$ is said to be an *R-bisimulation* between M_1 and M_2 just in case it satisfied the forth and back conditions (3.13a,b) and if $\mu S \nu$ then $R(\mu, \nu)$. Additionally, M_1 and M_2 are said to be *bisimilar* just in case such a relation exists.

Before returning to the question of how we can apply this definition to the class of

[14]In modal logic and process algebra there are many well known results which connect the bisimilarity of transition system-like structures and various notions of "local" language equivalence. For instance if $\mathfrak{M}_1 = \langle W_1, R_1, v_1 \rangle$ and $\mathfrak{M}_2 = \langle W_2, R_2, v_2 \rangle$ are models of propositional modal logic and S a bisimulation linking $w_1 \in W_1$ and $w_2 \in W_2$, then it can be shown that for all modal formals φ, $\mathfrak{M}_1, w_1 \models \varphi$ iff $\mathfrak{M}_2, w_2 \models \varphi$. More significantly, however, there is a partial converse to this result stating that if $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are *image finite* models (i.e. are such that all world in W_i bear R_i to at most finitely many other worlds for $i \in \{1, 2\}$), if $w_1 \in W_1$ and $w_2 \in W_2$ agree on all modal formulas, then there is bisimulation linking them. See [8] and [148] for references on this and a variety of other similar results linking bisimilarity to various forms of state-based linguistic equivalence. Note that on the basis of such results it seem promising to approach the problem of transition system equivalence by working backwards from a notion of linguistic indiscerniblity to an appropriate form of structural equivalence. There are, however, two substantial obstacles to this approach in the current context. First, note that implementations drawn from different models of computation will most naturally be described using different languages, meaning that there is no model independent way of selecting a "universal" language in which to describe the local properties of states. And second, note that results allowing us to infer from linguistic equivalence to bisimilarity generally apply only to languages that contain one or more modal operators. In the current context, the use of such operators would allow us describe properties of states which pertain not only to their "intrinsic" structural properties, but also to those of other states accessible via computational transitions. But in seeking to define R so that it bears only states structural properties, it is exactly this sort of information we are seeking to exclude.

[15]This proposal is defend by by Gurevich [51]. Note, however, that say that the members of St are $\mathfrak{L}_M$-structures entails neither that they are models of a particular first-order theory or that these classes of structures are even first-order definable. In fact, the second requirement will *not* be met in many common cases where we wish to impose the requirement that the classes of states in question have finitely many memory locations.

implementations $\mathfrak{T}$ under consideration, it will be useful to impose one other general requirement on the sort of simulation relations in which we are interested. Note that our current definition does not require that a bisimulation S linking transition systems M_1 and M_2 relate any particular pairs of their states. It is thus consistent with this definition that a bisimulation relation may be empty, from which it follows that any pair of transition systems will trivially be bisimilar. One way to avoid this problem is to require that S relate all the initial states of M_1 with the analogous initial states of M_2. In order to do this in a systematic manner, it will be useful to require that M_1 and M_2 have the same domain X, meaning that their input functions in_1 and in_2 are respectively of types $X \to St$ and $X \to St$.[16] In this case, we can define an *initial bisimulation* between M_1 to be a relation $S \subseteq St \times St$ such that for all $x \in X$, $in_1(x)\ S\ in_2(x)$. In the sequel, I will assume that all bisimulations in question are initial.

3.4.2 Refining the definition of bisimulation

To begin our assessment of how the general definition of bisimulation just outlined can be adapted so as to apply to the machines in $\boldsymbol{T}$, it will be useful to start out considering a version of Definition 3.4.3 which is based on the strictest possible choice of representation relation R. To this end, consider the relation which holds between Turing machine configurations μ and ν just in case the components of μ and ν other than that representing the state are identical. I will denote this relation by $\sim$. Formally we have that if $\mu = u_1 k_1 v_1$ and $\nu = u_2 k_2 v_2$, $\mu \sim \nu$ if and only if and $u_1 = u_2$ and $u_2 = v_2$.

With a complete definition of $\underline{\leftrightarrow}$ finally in place, we can begin to assess how well this relation fares with respect to satisfying the extensional adequacy requirements (3.10) and (3.11). First recall that per (3.10i), an algorithmic realist needs it to turn out that $S_1 \underline{\leftrightarrow} S_2$.[17] In order to see this, first note that S_2 differs from S_1 only in the presence of an

[16] While this does limit the domain to which the various definition of bisimulation which will be considered below may be applied, it is important to keep in mind that the algorithmic realist's interest in the notion of bisimulation derives from its potential role in formulating a definition of algorithmic identity. A trivial adequacy condition on such a definition is that identical algorithms A_1 and A_2 compute the same function, say of type $X \to Y$. From this it follows that any machines M_1 and M_2 which we might take to implement A_1 and A_2 will also induce a function of type $X \to Y$. There is thus no relevant loss in generality entailed by restricting the definition of bisimulation so that bisimiliar implementations must have the same domain.

[17] Although strictly speaking bisimulations are defined over transition systems and not Turing machines,

additional state $\underline{11}_2$ connected to $\underline{1}_2$. And it is easy to see that if S_2 is started in any initial state of the form $in_2(w)$, then it will never enter this state. Translating this observation into transition system talk, we can see that for no initial state of the form $\nu_1 = in_2(x)$ will there be a sequence of transitions $\nu_1 \xrightarrow{2} \nu_2 \xrightarrow{2} \ldots \xrightarrow{2} \nu_{n-1} \xrightarrow{2} \nu_n$ such that $\nu_n = u\underline{11}_2 v$ for some strings $u, v \in \Sigma^*$. Since all configurations of this form are thus inaccessible in S_2 from states in St_2, it follows that there is no requirement that such states needs to be in the range of an initial bisimulation S linking S_1 and S_2. Hence the trivial relation

$$S = \{\langle \mu, \nu \rangle \ : \ \mu = u\underline{\mathsf{k}}_1 v \wedge \nu = u\underline{\mathsf{k}}_2 v \wedge \underline{\mathsf{k}} \in \{\underline{0}, \ldots, \underline{10}\}\}$$

which simply omits state of the form $u\underline{11}_2 v$ from the range of S will be an initial bisimulation between S_1 and S_2.

Testing the adequacy of our baseline definition of bisimulation starts to become more complicated when we consider the case of S_1 and S_3. Again, the algorithmic realist wishes to formulate a definition of bisimulation so that $S_1 \underline{\leftrightarrow} S_3$. Like S_2, S_3 may be seen as deriving from S_1 through the addition of "inessential" states, this time in the sense that states $\underline{11}_3$ and $\underline{12}_3$ have the effect of immediately counteracting one another. For consider a case in which both S_1 and S_3 are operating on a string of the form $w \equiv 1u1$. Having stored the initial 1 in their state and moved their heads to the right of the tape, they will respectively enter states of the form $u\underline{5}_1 1$ and $u\underline{5}_3 1$. But whereas S_1 then overwrites the final 1 with a $-$ and then transitions immediately to a state of the form $u\underline{7}$, S_3 first overwrites this symbol with a $-$ and then, transitioning through states $\underline{11}_3$ and $\underline{12}_3$, successively overwrites the $-$ with a 0 and the 0 with a $-$ before entering a state of the form $u\underline{7}_3$. With respect to the problem of deciding whether w is a palindrome, it seems reasonable to say that the same computational function performed by S_1 in course of performing a single transition of the form $u\underline{5}_1 1 \xrightarrow{1} u\underline{7}_1$ is performed by S_3 by executing a sequence of transitions of the form $u\underline{5}_3 1 \xrightarrow{3} u\underline{11}_3 \xrightarrow{3} u\underline{12}_3 0 \xrightarrow{3} u\underline{7}_3$.

On this basis, we can now see that S_1 and S_3 are not related by the definition of $\underline{\leftrightarrow}$ as it now stands. For note that we assume that any initial bisimulation S linking S_1 and

---

we have seen that a given Turing machine uniquely induces a transition system. I will henceforth abuse notation and write for Turing machines $T, T' \in \mathcal{T}$ $T \underline{\leftrightarrow} T'$ to mean that $M_T \underline{\leftrightarrow} M_{T'}$ where M_T and $M_{T'}$ are the transitions systems associated with T and T' in the sense described above.

S_3 must be such that $u\underline{5}_1 1 \; S \; u\underline{5}_3 1$.[18] But it then follows that since $u\underline{5}_1 1 \stackrel{1}{\longrightarrow} u\underline{7}$ is the unique transitition emanating from states of the first form, then since $u\underline{5}_3 1 \stackrel{3}{\longrightarrow} u\underline{11}_3$ is the unique transition emanating from $u\underline{5}_3 1$, by (3.13a) we must have $u\underline{7}_1 \; S \; u\underline{11}_3$. But then since $u\underline{7}_3 \stackrel{3}{\longrightarrow} u\underline{12}_3 0$ is the unique transition of emanating from the former state in S_3 and $u\underline{7}_1 \stackrel{1}{\longrightarrow} u'\underline{8}_1 a$ (for $u \equiv u'a$, $a \in \{0,1\}$) is the unique transition from the former state in S_1, it follows that we must have $u'\underline{8}_1 a \; S \; u\underline{12}_3 0$. But note that clearly these two states cannot be linked by any relation satisfying the representation relation $\sim$ defined above as they have different tape components. And from this it follows that $S_1 \underline{\leftrightarrow} S_3$ cannot hold according to our current definition.

This is the first of several instances in which the need to accommodate the equivalence of a given pair of machines T_1 and T_2 appear to necessitate the revision of our baseline definition (3.4.3) of bisimulation. The general problem posed by such situations is that isolated observation that M_1 and M_2 fail to fall under our current definition generally provides only a partial indication of what form such a revision should take. In the current case, for instance, it is possible to either revise the representation requirement as given by $\sim$ so as to allow $\mu \; S \; \nu$ in cases where μ and ν differ in more than just their state component or to modify one or both of the transitional requirements (3.13a,b). The decision as to how to proceed in such situations is, of course, ultimately in the hands of a realist whose job it is to show that the details of the abstractionist programme can be carried out by defining an adequate definition of bisimulation. In arguing against the realist, it would thus be rhetorically desirable to consider all possible revisions. But since this is obvious not a feasible rhetorical strategy, I will proceed in each case by revising in the manner which seems most favorable to the realist's cause.

To this end, it seems reasonable to start out by attempting to formalize the suggestion made two paragraphs back that we should view single transitions of the form $u\underline{5}_1 1 \stackrel{1}{\longrightarrow} u\underline{7}_1$ of S_1 as mirroring sequences of transitions of the form $u\underline{5}_3 1 \stackrel{3}{\longrightarrow} u\underline{11}_3 \stackrel{3}{\longrightarrow} u\underline{12}_3 0 \stackrel{3}{\longrightarrow} u\underline{7}_3$

[18] This follows by considering the execution of these machines on any input of the form $w \equiv 1a_2 \ldots a_{n-1}1$. For note that we must then have $\underline{0}_1 w S \underline{0}_3 w$, from which it follows inductively that $\underline{1}_1 - a_2 \ldots a_{n-1} 1 S \underline{1}_3 - a_2 \ldots a_{n-1}1$, $\underline{3}_1 a_1 \ldots a_{n-1} 1 S \underline{3}_3 a_1 \ldots a_{n-1}1$, $a_1 \underline{3}_1 a_2 \ldots a_{n-1} 1 S a_1 \underline{3}_3 a_2 \ldots a_{n-1}1$, $\ldots$, $a_1 \ldots a_{n-1} 1 \underline{3}_1 S a_1 \ldots a_{n-1} 1 \underline{3}_3$ and thus finally that $a_1 \ldots a_{n-1} \underline{5}_1 1 S a_1 \ldots a_{n-1} \underline{5}_3 1$. The fact this chain of relations must obtain follows from the conditions (3.13a,b) and the fact that S_1 and S_3 are deterministic.

of S_3. To motivate this proposal more generally, note that even though in this case it is tempting to view the latter sequence of transitions as a "computational detour" in the operation of S_3, there also will be situations in which single stages in the operation of an informally described procedure like PAL1 will be carried out by a single transitions with respect to one of its implementations but multiple transitions with respect to some other. One way we can to accommodate this observation in a general setting is to relax (3.13a,b) so that if $\mu \xrightarrow{1} \mu'$ in M_1 and $\mu S \nu$ for some state ν of M_1, then the state mirroring μ' in M_2 need not be an *immediate* successor under $\xrightarrow{2}$ of ν. We could, for instance, instead require merely that there existed a stated ν' and sequence of intermediate states $\nu_2, \ldots, \nu_{n-1}$ such that $\nu = \nu_1 \xrightarrow{2} \nu_2 \xrightarrow{2} \ldots \xrightarrow{2} \nu_{n-1} \xrightarrow{2} \nu_n = \nu'$ and $\mu' S \nu'$.

The simplest way to accommodate this into a definition of bisimulation is simply to revise the back and forth requirement replacing the transition relation $\xrightarrow{2}$ in (3.13a) with its transitive closure $\xrightarrow{2^*}$ and the relation $\xrightarrow{1}$ in (3.13b) with its transitive closure $\xrightarrow{1^*}$. This yields this following set of new transition requirements for our definition of bisimulation

(3.13c) $\forall\mu\forall\mu'\forall\nu[(\mu \xrightarrow{1} \mu' \wedge \mu S \nu) \rightarrow \exists\nu'(\mu' S \nu' \wedge \nu \xrightarrow{2^*} \nu')]$

(3.13d) $\forall\nu\forall\nu'\forall\mu[(\nu \xrightarrow{2} \nu' \wedge \mu S \nu) \rightarrow \exists\mu'(\mu' S \nu' \wedge \mu \xrightarrow{1^*} \mu')]$

According to this revised definition, it may readily be seen that $S_1 \underline{\leftrightarrow} S_3$. In particular, there now exists an allowable simulation relation which S such that $u\underline{5}_1 1 \; S \; u\underline{5}_3$ and $u\underline{7}_1 \; S \; u\underline{7}_3$. Note that this simulation simply omits the states of the form $u'\underline{11}_3$ and $u'\underline{12}_3$ from its range. Such a simulation thus formalizes the intuition that we may regard S_1 and S_3 as step-by-step equivalent in their operation if we think of the work performed by the transitions $u\underline{5}_3 1 \xrightarrow{3} u\underline{11}_3 \xrightarrow{3} u\underline{12}_3 0 \xrightarrow{3} u\underline{7}_3$ in S_3 as being performed by the transition $u\underline{5}_1 1 \xrightarrow{1} u\underline{7}_1$ in S_1.

It is fairly easy to see, however, that loosening the definition of $\underline{\leftrightarrow}$ in such a simple manner will have several undesirable consequences. Perhaps the most serious of these is that under this definition, the transitional requirement on a simulation relation becomes very easy to satisfy. For let T_1, T_2 be any pair of machines which determine the same mapping between classes of states. Then it follows that T_1 and T_2 will be bisimiliar regardless of our choice of representation requirement, even if we continue to take R to be $\sim$. For let M_1 and

M_2 be the transition systems based on T_1 and T_2 and such that the functions induced by their operation on their statements of states is identical (modulo $\sim$). Now define S to be the relation such that for all $x \in X$, $in_1(x) S in_2(x)$ and $App_{M_1}(in_1(x))\ S\ App_{M_2}(in_2(x))$. Since we also have $in_1(x) \sim in_2(x)$ and $App_{M_1}(in_1(x)) \sim App_{M_2}(in_2(x))$ by assumption it follows that $T_1 \underline{\leftrightarrow} T_2$.

On the basis of this example, we may see that a definition of bisimulation based on these modified requirements fails to place any substantial requirement on how the step by step behavior of M_1 is mirrored by M_2 (or conversely). In particular, this definition is so broad as to allow the existence of bisimulations linking implementations which induce the same mapping on states (up to $\sim$). And from this it follows there will exist an "unintended" bisimulation linking machines such as S_1 and U_1 even though this is explicitly ruled out by the one of our extensional adequacy requirements (3.11).[19]

The situation just described suggests that if wish to define a bisimulation relation which is compatible with (3.10) and (3.11) then we must seek to formulate a more sophisticated characterization of what it means for a sequence of transitions of M_1 to mirror a sequence of transitions of M_2. On the basis of the cases we have considered thus far, it might seem that since it is only sequences of transitions of S_3 of length two that are mirrored by single transitions of S_1, it would suffice to replace $\xrightarrow{2^*}$ in (3.13c,d) with $\xrightarrow{2^2}$ (denoting the relation obtained by iterating Δ_2 two or fewer times). But of course it is easy enough to construct implementations similar S_3 which are such that single transition of $u\underline{5}_1 1 \xrightarrow{1} u\underline{7}_1\ S_1$ is mirrored by a sequence of mutually counteracting transitions of arbitrary length. And for this reason it is evident that merely replacing $\xrightarrow{2*}$ with $\xrightarrow{2^n}$ for any fixed n will not lead to a satisfactory definition.

The example of S_1 and S_3 does, however, suggest another more sophisticated way in which (3.13a,b) could be modified in order to accommodate not just this case but a variety

[19] A related problem with basing a definition of bisimulation on (3.13c,d) is that since these conditions allow arbitrary length sequences of M_1 to be mirrored by single transitions of M_1 (and vice versa), it is clear that a relation satisfying these conditions need not preserve running time complexity when measured directly in terms of the individual transitions of M_1 and M_2. As I will discuss further below, this makes it unlikely that such a definition will satisfy the intensional adequacy condition that $\underline{\leftrightarrow}$ be a congruence with respect to complexity theoretic properties as expressed by (3.6) (taking φ to be, e.g. "has running time $O(f(|x|))$").

of other similar ones. For note that although the cases we have just considered suggest that any way of adapting this definition so that single transitions may be mirrored by sequences of transitions of arbitrary length will make the resulting definition too broad, it is also possible to define S so that it directly relates sequences of transitions with a particular form rather than individual states. In order to understand the motivation for undertaking this sort of modification, it will also be useful to examine the relationship between the pairs of implementations S_1, S_5 and U_1, U_5 in greater detail. For recall that while it may be natural to describe S_3 as a "trivial" variant of S_1 derived by purposefully adding superfluous states, we have seen that the relationship between S_1 and S_5 cannot be described in such a simple manner. Nonetheless, if we describe the relationship between the operation of these two machines more abstractly, it does seem possible to provide a positive characterization of the relationship which they bear to one another in virtue of which we take them to implement PAL1.

To get a concrete impression of this, let us again consider the operations of these machines on the input $w \equiv a_1a_2a_3a_4 \equiv 1001$. Using our standard conventions for representing configurations and transitions, the operation of these two machines on w is summarized in Figure (3.6). Upon examining these executions, it should be evident that there can be no way of setting up a one-to-one correlation of configurations between S_1 and S_5 in original the sense of (3.13a,b). For note that not only are lengths of the executions of these machines different (18 transitions versus 38 transitions), but it is also not possible to identify a sequence of transitions of one these machines which we would say is "collapsed" into a single transition of the other. Thus despite the fact that these machines fall under the overly permissive definition of bisimulation based on (3.13c,d), they do so only because of the existence of "unintended" bisimulations of the sort we have considered above.

But it should also be evident on the basis of Figure 3.6 that there is a systematic means of correlating the transitions of S_1 with those of S_5 in a manner that preserves the "functional" descriptions of their operation provided in the rightmost column. Note, for instance, that both S_1 and S_5 compare the outermost values a_1 and a_4 before they compare the inner values a_2 and a_3. These tasks are respectively performed by the transitions $\mu_7 \xrightarrow{1^*} \mu_9$ and $\mu_{16} \xrightarrow{1^*} \mu_{18}$ of S_1 and $\nu_{19} \xrightarrow{5^*} \nu_{23}$ and $\nu_{33} \xrightarrow{5^*} \nu_{37}$ of S_5. Similarly, the task of returning

S_1 transitions	States	Description
$\underline{0}1001 \xrightarrow{1} \underline{1}-001 \xrightarrow{1} \underline{3}001 \xrightarrow{1} 0\underline{3}01 \xrightarrow{1} 00\underline{3}1 \xrightarrow{1} 001\underline{3} \xrightarrow{1}$	$\mu_1 - \mu_6$	remember a_1 in state and move right to the right end of the input
$00\underline{5}1 \xrightarrow{1} 00\underline{7} \xrightarrow{1}$	$\mu_7 - \mu_8$	compare a_1 and a_4
$0\underline{8}0 \xrightarrow{1} \underline{8}00 \xrightarrow{1} \underline{8}-00 \xrightarrow{1}$	$\mu_9 - \mu_{11}$	move head left
$\underline{0}00 \xrightarrow{1} \underline{2}-0 \xrightarrow{1} \underline{4}0 \xrightarrow{1} 0\underline{4} \xrightarrow{1}$	$\mu_{12} - \mu_{15}$	remember a_2 in state and move right to the right end of the input
$6\underline{0} \xrightarrow{1} \underline{7} \xrightarrow{1}$	$\mu_{16} - \mu_{17}$	compare a_2 and a_3
$\underline{8} \xrightarrow{1}$	μ_{18}	move head left
$\underline{0} \xrightarrow{1} \underline{10}$	$\mu_{19} - \mu_{20}$	accept and halt

S_5 transitions	States	Description
$\underline{0}1001 \xrightarrow{5} 1\underline{1}001 \xrightarrow{5} \underline{3}1001 \xrightarrow{5} \underline{5}0001 \xrightarrow{5} 0\underline{7}001 \xrightarrow{5} 0\underline{0}101 \xrightarrow{5} 01\underline{1}01 \xrightarrow{5} 0\underline{3}101 \xrightarrow{5} 0\underline{3}101 \xrightarrow{5} 0\underline{5}001 \xrightarrow{5} 00\underline{7}01 \xrightarrow{5} 00\underline{0}11 \xrightarrow{5} 001\underline{1}1 \xrightarrow{5} 00\underline{9}11 \xrightarrow{5} 00\underline{1}111 \xrightarrow{5} 001\underline{13}1 \xrightarrow{5} 00\underline{1}01 \xrightarrow{5} 00111 \xrightarrow{5}$	$\nu_1 - \nu_{18}$	push a_1 to the right end of the input
$001\underline{15}1 \xrightarrow{5} 00\underline{117} \xrightarrow{5} 00\underline{19}1 \xrightarrow{5} 00\underline{20} \xrightarrow{5}$	$\nu_{19} - \nu_{22}$	compare a_1 and a_4
$0\underline{21}0 \xrightarrow{1} \underline{21}00 \xrightarrow{5} \underline{21}-00 \xrightarrow{5}$	$\nu_{23} - \nu_{25}$	move head left
$\underline{0}00 \xrightarrow{5} 0\underline{2}0 \xrightarrow{5} \underline{10}00 \xrightarrow{5} \underline{12}00 \xrightarrow{5} 0\underline{14}0 \xrightarrow{5} 0\underline{0}0 \xrightarrow{5} 00\underline{2} \xrightarrow{5}$	$\nu_{26} - \nu_{32}$	push a_1 to the right end of the input
$0\underline{15}0 \xrightarrow{5} 0\underline{16} \xrightarrow{5} \underline{18}0 \xrightarrow{5} \underline{20} \xrightarrow{5}$	$\nu_{33} - \nu_{36}$	compare a_2 and a_3
$\underline{21} \xrightarrow{5}$	ν_{37}	move head left
$\underline{0} \xrightarrow{5} \underline{22}$	$\nu_{38} - \nu_{39}$	accept and halt

Figure 3.6: Transitions executed by S_1 and S_5 on input 1001.

the head to the left end of tape after these comparisons are performed respectively by the transitions $\mu_9 \xrightarrow{1^*} \mu_{12}$ and $\mu_{18} \xrightarrow{1^*} \mu_{19}$ of S_1 and $\nu_{23} \xrightarrow{5^*} \nu_{25}$ and $\nu_{37} \xrightarrow{5^*} \nu_{38}$ of S_5.

On the basis of such observations, it seems reasonable to attempt to revise (3.4.3) so that a bisimulation S relation between transitions system M_1 and M_2 is now taken to relate finite sequences of transitions among states instead of individual states. The intention, again, is that by defining bisimulation in this manner it will be possible to formalize uniform regularities in the operations of machine such as S_1 and S_5 which we intuitively wish to fall under a definition of computational equivalence, but whose state-to-state transitions

cannot be directly correlated. However, as soon as we consider how to adapt our current definition of bisimulation to accomodate this possibility, a number of conceptual, technical and notational complications arise. So as to movitate the definition on which I settled below, I will begin by flagging several of these concerns.

i) Since we now wish to view two transition systems M_1 and M_2 as being computationally equivalent in virtue of a relation linking not individual states but sequences of states occurring in their executions, the type of a simulation relation S must be redefined accordingly. To this end, let St^* denote the set of finite sequence of states over St. I will use variables $\vec{\mu}, \vec{\nu}, \ldots$ to vary over such sequences. A simulation relation between M_1 and M_2 will thus should now be of type $S^* \subseteq St * \times St^*$. I will refer to such a relation between as a *sequence simulation*.

ii) As we have already noted, correlating sequences instead of states does not automatically rule our "unintended" simulations where complete executions of one transition system are correlated with those of another operationally distinct system. In order to do this, we must find some way of uniformly decomposing an arbitrary finite executions $\vec{\xi}(x)$ of a transition system M into subcomponents $\vec{\xi_1}(x) \cdot \vec{\xi_2}(x) \cdot \ldots$ such $|\vec{\xi_i}(x)| < |\vec{\xi}(x)|$ and each $\vec{\xi_i}$ corresponds to a functional delimited porition of M's operation on x (and where if $\vec{\xi}(x)_i \equiv \langle \xi_{i,1}(x), \ldots, \xi_{i,m_i} \rangle$ then $\Delta_(\xi_{i,j}) = \xi_{i,j+1}$ for $1 \leq j < m_i$ and if $\xi_{i,m_i} \notin H_M$, then $\delta(\xi_{i,m_i}) = \xi_{i+1,1}$).

iii) Then the manner in which a desired decomposition $\vec{\xi}(x) = \vec{\xi_1}(x) \cdot \vec{\xi_2}(x) \cdot \ldots \cdot \vec{\xi_m}(x)$ will often depend on the input x. For note that if we consider the operation of S_1 as summarized in Figure 3.6, the length of, say, the sequence of transitions corresponding functionally as "remember a_1 in state and move right" will depend on the length of x. Such a parameterization can be achieved by providing what I will call a *sequence decomposition function* for M – i.e. a function $g : X \times \mathbb{N} \to \mathbb{N}^+$ which decomposes $\vec{\xi}(in(x))$ into subsequences such that $\vec{\xi_i}(x)$ so that $|\vec{\xi}(x)_i| = g(x, i)$.[20]If we additionally define a function $h : X \times \mathbb{N} \to \mathbb{N}$ such that for all $x \in X$, $h(x, 0) = 1$ and for $j > 0$, $h(x, j) = \sum_{i=1}^{j} g(x, i)$ then the complete execution $\vec{\xi}(x)$ will be uniformly decomposed as $\vec{\xi}(x) = \langle \xi_{h(x,0)}(x), \ldots, \xi_{h(x,1)-1}(x) \rangle \cdot \langle \xi_{h(x,1)}(x), \ldots, \xi_{h(x,2)-1}(x) \rangle \cdot \ldots$.

iv) If a sequence simulation S^* relates the subsequences $\vec{\mu}_i(x)$ and $\vec{\nu}_j(x)$ appearing in a given decompositions of the executions $\vec{\mu}(x)$ and $\vec{\nu}(x)$ of transition systems M_1 and M_2, then our new definition of $\underline{\leftrightarrow}$ ought to formalize the fact that $\vec{\mu}_i(x)$ plays the same role in operation of M_1 as does $\vec{\nu}_i(x)$ in M_2 and conversely. This can again be decomposed into a transitional requirement pertaining to the relationship which $\vec{\mu}_i(x)$ and $\vec{\nu}_j(x)$ bear to other subsequences in these decompositions and a representational requirement pertaining to these subcomponents either individually or relative to one other state in the subsequence.

v) The most straightforward way of formalizing the transitional requirement is to modify the original back and forth requirements (3.13a,b) so that they specify that consecutive subsequences occurring in executions of M_1 relative to a decomposition given by g_1 are related to consecutive subsequences of M_2 relative to a decomposition given by g_2. Let $j \in \{1, 2\}$ and g_j be a sequence decomposition function for M_j. I will write $\vec{\xi} \overset{j,g_j}{\Longrightarrow} \vec{\xi}'$ to symbolize the fact that there exists an $x \in X$ and $i \in \mathbb{N}$ such that $\vec{\xi} = \langle \xi_{h(i)}(x), \ldots, \xi_{x,h(i+1)-1}(x) \rangle$ and $\vec{\xi}' = \langle \xi'_{h(i+1)}(x), \ldots, \xi'(x)_{x,h(i+2)-1}(x) \rangle$. Then the back and forth requirements should take the form

(13e) $\forall\vec{\mu}\forall\vec{\mu}'\forall\vec{\nu}[(\vec{\mu} \overset{1,g_1}{\Longrightarrow} \vec{\mu}' \wedge \vec{\mu} S^* \vec{\nu}) \rightarrow \exists\vec{\nu}'(\vec{\mu}' S^* \vec{\nu}' \wedge \vec{\nu} \overset{2,g_2}{\Longrightarrow} \vec{\nu}']$

(13f) $\forall\vec{\nu}\forall\vec{\nu}'\forall\vec{\mu}[(\vec{\nu} \overset{2,g_2}{\Longrightarrow} \vec{\nu}' \wedge \vec{\mu} S^* \vec{\nu}) \rightarrow \exists\vec{\mu}'(\vec{\mu}' S^* \vec{\nu}' \wedge \vec{\mu} \overset{1,g_1}{\Longrightarrow} \vec{\mu}')]$

vi) A representational requirement can now be taken to be an equivalence relation R^* which is an equivalence relation on $St^* \times St^*$. Such a relation is intended to formalize the fact that S^*-related subsequences $\vec{\mu}_i(x)$ and $\vec{\nu}_i(x)$ have a similar overall structure. This may be taken to mean either that the individual states of which they are composed are pairwise similar (or, in the case where $|\vec{\mu}_i(x)| \neg |\vec{\nu}_i(x)|$, an appropriate generalization of pairwise), or that the structure these sequences are similar some other respect which is local in the sense of not pertain to the other sequences related by S^*. I will call a sequence simulation S^* R^*-*allowable* (or just *allowable* if R^* is clear from context) if for all $\vec{\mu}, \vec{\nu} \in St^* \times St^*$, if $\vec{\mu} S^* \vec{\nu}$, then $\vec{\mu} R^* \vec{\nu}$.

vii) We must additionally require that however the executions of M_1 and M_2 are decomposed by g_1 and g_2, a bisimulation relation should link the initial subcomponents of

executions M_1 with those of M_2 determined by the same input. To this end, define a *initial* sequence simulations for g_1 and g_2 to be a relation $S^* \subseteq St^* and St^*$ such that all $x \in X$, if $\vec{\mu} = \langle \mu_{h_1(x,0)}, \ldots, \mu_{h_1(x,1)} \rangle$ and $\vec{\nu} = \langle \mu_{h_2(x,0)}, \ldots, \mu_{h_2(x,1)} \rangle$, then $\vec{\mu} S^* \vec{\nu}$.

Based on the foregoing observations and definitions, we may now formulate a revised definition of bisimulation as follows:

Definition 3.4.4. Let M_1 and M_2 be transition systems and $R^* \subseteq St_1^* \times St_2^*$ be a representation requirement. A initial sequence simulation $S^* \subseteq \Delta_1^* \times \Delta_2^*$ is said to be an *initial sequence bisimulation* just in case it is R^*-allowable and there exist decomposition functions g_1, g_2 such that S^*, g_1 and g_2 jointly satisfy the transitional requirement given by (3.13e,f). We will continue to say that M_1 and M_2 are *sequence bisimiliar with respect to* S^*, g_1, g_2 and simply that they are *sequence bisimiliar* (which I will continue to write as $M_1 \underline{\leftrightarrow} M_2$) if such a relation and functions exist.

Although definition is still parameterized in the representation relation R^*, it is evident that sequence bisimularity will be an equivalence relation on St^* as long as is R^*.

In attempting to determine what form such a requirement should take, it seems that we have little choice but to start out by examining a situation in which we wish to view two machines as bisimiliar for purposes of satisfying one of the constraints (3.10). We have already undertaken such an analysis by analyzing the operation of S_1 and S_5 in the manner displayed in Figure 3.6. In this case, I have argued that the algorithmic realist will hold that the fact that S_1 and S_5 are implementations of the same algorithm in virtue of the fact that their operations can be decomposed into subsequences of the sort given in the left hand columns of Figure 3.6 which can in turn be aligned with the informally delimited stages in the operation of PAL1 as described in Section 3. It thus follows that a realist who also embraces the (BP), will wish to analyze the fact that $imp(S_1) = imp(S_2)$ in terms of the relationship between the subsequence of states given in the left hand column of Figure 3.6. In concrete terms, this means that a realist will wish R^* to be defined so that there exists an allowable sequence simulation $S^*_{1,5}$ relating not only the individual $\langle \mu_1, \ldots, \mu_6 \rangle$

[20]Formally, such a function should meet the following conditions: i) g is strictly monotone; ii) if the complete execution $\vec{\xi}(x)$ is of length n, then $g(x,i) \leq m$ for all i; iii) there exists $q \leq n$ for that $\sum_{i=1}^{q} g(x,i) = n$; and iv) for all $i > q$, $g(x,i) = 0$.

and $\langle \nu_1, \ldots, \nu_{18} \rangle$, $\langle \mu_7, \mu_8 \rangle$ and $\langle \nu_{19}, \ldots, \nu_{22} \rangle$, etc. which arise in operation of S_1 and S_5 on $w \equiv 1001$, but also other similarly structured subsequences which arise for other inputs $w \in \{0,1\}^*$.

In order to accommodate this possibility under (3.4.4), we first need to show that there exist decomposition functions g_1 and g_2 which give rise to the partitioning of states given in Figure 3.6. If, for simplicity, we restrict attention to the case where the input string $w \in L_{pal}$,[21] then such functions may be given as follows:

(3.14) a) $g_1(w,i) = \begin{cases} |w| - 2(\lfloor i/3 \rfloor) + 2 & \text{if } i \equiv 1 \bmod 3 \\ 1 & \text{if } i \equiv 2 \bmod 3 \\ |w| - 2(\lfloor i/3 \rfloor) + 2 & \text{if } i \equiv 0 \bmod 3 \end{cases}$

b) $g_2(w,i) = \begin{cases} 5(|w| - 2(\lfloor i/3 \rfloor - 1) + 2 & \text{if } i \equiv 1 \bmod 3 \\ 3 & \text{if } i \equiv 2 \bmod 3 \\ |w| - 2(\lfloor i/3 \rfloor) + 2 & \text{if } i \equiv 0 \bmod 3 \end{cases}$

If we assume $w \in L_{pal}$, then relative to this pair of decomposition functions, any allowable initial sequence bisimulation $S^*_{S1,S5}$, will have to relate subsequences $\vec{\mu}$, $\vec{\nu}$ of $\vec{\mu}(w)$ and $\vec{n}(w)$ just in case they result from a partitioning analogous to that in Figure 3.6. This, of course, is a necessary condition to inscribe S_1 and S_5 under a definition of bisimulation. But we still need to formulate a representation requirement R^* which, taken in conjunction with Definition (3.4.4) will serve as an appropriate sufficient condition. As I have already mentioned, one of the paramount concerns in formulating such a definition is to rule out relations like $S^*_{S1,U1} = \{\langle \vec{\mu}(w), \vec{\rho}(w) \rangle \ : \ w \in \{0,1\}^*\}$ which relate complete executions of S_1 and U_1 on all inputs.

One way to characterize the reason a realist will wish to rule out $S^*_{S1,U1}$ as an allowable bisimulation is because we cannot use such a simulation as part of an argument that S_1 and U_1 operate in the same way as we can in the case of S_1 and S_5 relative to $S^*_{S1,S5}$. But in

[21] It is straightforward to extend the given definition of g_1 and g_2 to cover functions g_1' and g_2' which cover cases where $w \notin L_{pal}$. If $w \equiv a_1 \ldots a_n$, then this extension involves the addition of several cases which describe the operation of S_1 and S_5 when they encounter the least i such that $a_i \neq a_{n-(i-1)}$. But since the details of the definition of g_1' and g_2' will not effect our discussion below, I will not give their definitions here.

both these cases, we start out with a functional decomposition of the operation each of the machines in question, and it only relative to this means of understanding the operations of the machines in question that we are able to make these judgments. In attempting to frame a definition of R^* which embraces $S^*_{S1,S5}$ but rejects $S^*_{S1,U1}$ as an allowable bisimulation, we may assume that the realist must avoid reliance on such pre-specified decompositions. For note that decompositions of the sort provided in (3.6) are derived via a prior initive understanding of what it means for these machines to implement the algorithms PAL1 and PAL2. And it is exactly such an alliance on such intuitions which the realist presumably wishes to eliminate in attempting to characterize algorithms via (BP).

But in seeking to formulate general conditions which a definition of R^* must meet in order to correctly partition $\mathcal{T}$ into equivalence classes of co-implementing algorithms, it seems that the realist can do little better than to extrapolate the lessons learned by considering cases of the sort of we have been considering. One preliminary observation which we can make on his behalf is that one way in which $S^*_{S1,U1}$ differs from $S^*_{S1,S5}$ is that the *length* of the sequences related by the former is longer than those related by the latter. In particular, in the former case, it follows immediately from the definition of this relation that if $\vec{\mu}S^*_{S1,U1}\vec{\rho}$, then $|\vec{\mu}| = len_{S1}(w)$ and $|\vec{\rho}| = len_{U1}(w)$ (where w if $\mu_1 = \langle \underline{\mathsf{k}}, x, y\rangle$, then $w = x^R y$). If we continue to assume $w \in L_{pal}$ and let $|w| = n$, then we may explicitly compute that $len_{S1}(n) = (n^2 + 5n + 6)/2 - 2$ and $len_{U1}(n) = (3n^2 + 6)/2$. And from this it follows that $S^*_{S1,U1}$ relates sequences which are of length $O|w|$. On the other hand, it may be seen directly from (3.14) that $S^*_{S1,S5}$ relates only sequences which are of length at most $O(|w|)$.

On the basis of this example, it seems reasonable to suppose that the realist might seek to formulate the definition of R^* so as to rule out $S^*_{S1,U1}$ on the basis of the fact that it related subsequences which are of length quadratic to the input size of $S1$ and $U1$. There are, however, a number of problems involved with making this proposal compatible with our current definition of bisimulation. The first of these depends on the fact that since even $S^*S1, S5$ will relate subsequences of arbitrary length and thus there is no way to fix this requirement in terms of the absolute length of the sequences related. Rather, in order to formalize the current suggestion, we must thus look for some parameter of arbitrary states

sequences $\vec{\sigma}$ which can be taken as a measure the quantity relative to which we wish to formalize the relevant prohibition. This is easy in the case of $S^*_{S1,U1}$, since the initial states of the related sequences $\vec{\mu}(x), \vec{\rho}(x)$ directly encode the input string. But we can just as easily imagine another "unintended" sequence simulation relating S_1 and U_1 where there appears to be a uniform way of identifying the parameter in question in a manner that does not depend on the specific details of their operation.[22]

This problem, however, is just one of the many which an algorithmic realist is likely to face in attempting to formulate a definition of R^* so that the resulting definition of bisimulation will satisfy the extensional adequacy conditions (3.10) and (3.11). For note that although in the case just considered, it seems there is a good rational for ruling out sequence simulations which relate sequences of states which are of length $O(|w|^2)$, there are other instances in which it appears that such simulations must be tolerated in order to accommodate these constraints. This may be seen, for instance, by considering again the operation of the machines U_1 and U_2 which, per (3.10ii), ought to be related under a satisfactory definition of $\underline{\leftrightarrow}$. Recall that I initially argued for this on the basis of the fact that both of these machines can reasonably be taken to be implementations of the algorithm PAL2 on the basis of an informal comparison of stages and state sequences.

But like S_1 and S_5, neither of these machines can be described as a simple variant of the other. And this means that if we wish to assimilate the operation of U_1 and U_2 under an appropriate sequence simulation relation $S^*_{U1,U2}$, we must again look for a way of correlating stages in their execution in a manner similar that given for S_1 and S_5. The decomposition on which such a relation would be based will be considerably more than the analogous case involving S_1 and S_5 which we have just considered. But luckily the general point which I wish to frame does not require that it be given explicitly. For even without fully decomposing the executions of U_1 and U_2, we know that $S^*_{U1,U2}$ ought to relate these machines in virtue of linking sequences which implement individual stages in the operation

[22]For instance consider the sequence simulation which related all first halves $\vec{\mu}_1(w)$ of executions of $\vec{\mu}(w)$ of S_1 to those of first halves $\vec{\zeta}_1(w)$ of executions $\vec{\zeta}_1(w)$ of S_5 and similarly for second halves $\vec{\mu}_2(w)$ and $\vec{\zeta}_2(w)$ (rounding lengths as necessary). This simulation is also unintended in that such a decomposition is arbitrary from the standpoint of the operation of these machines. Also note that the components of each execution will still have length $O(|w^2|)$. The problem in this case, however, is that there appears to be no non-arbitrary way of extracting $|w|$ from $\vec{\mu}_2(w)$ or $\vec{\zeta}_2(w)$.

of PAL2 uniformly for all inputs.

But now note that in order for $S^*_{U1,U2}$ to satisfy this requirement, it must, for all inputs w link the initial segments of the execution of U_1 and U_2 which are responsible for transacting the operation described by Step 2 of PAL1. Recall that this step is expressed in pseudocode as

(3.15) For $i = 1$ to n do

Let $b_i = a_{n-(i-1)}$

and has the effect of constructing a reversed copy of input string $w \equiv a_1 \ldots a_n$ in the registers indexed by $b_1, \ldots, b_n$. But it is also apparent that Turing machines cannot directly perform operations like copying symbols directly by accessing tape cells by index. In order to perform the operation just described U_1 and U_2 must thus proceed indirectly. In the case of U_1, this is achieved by constructing a new string to the right of w by successively copying symbols from the right of w. And in the case of U_2, however, it is is accomplished by first constructing a copy of w imedidate to its right using states. This copy is then reversed by directly swapping symbols from its left to its right ends. For ease of reference call the submachines of U_1 and U_2 responsible for carrying out the procedures just described U'_1 and U'_2.[23]

The problem which the case of U_1 and U_2 illustrates is as follows. In order to conform with our prior conclusion that these machines both implement PAL2, we wish to allow that subsequences of their overall executions induced by the operation on U'_1 and U'_2 (call these $\vec{\rho}'(w)$ and $\vec{\tau}'(w)$) must be somehow correlated by $S^*_{U1,U2}$ for all w. This can be achieved in one of two ways. First we may define this relation so that it links such subsequences directly – i.e. without further decomposing them into shorter subsequences. And second, we may attempt to decompose $\vec{\rho}'(w)$ and $\vec{\tau}'(w)$ into shorter subsequences which are then linked by $S^*_{U1,U2}$.

Although it may seem most in keeping with the goal of the realist to align the operations of U_1 and U_2 as tightly as possible to pursue the second strategy, a number of new problems

[23]More precisely, U'_1 is the machine $\langle K'_1, \{0,1\}, \delta'_1 \underline{1}_1, \underline{11}_1 \rangle$ where $K'_1 = \{\underline{1}_1, \ldots, \underline{11}_1\}$ and δ'_1 is derived from δ_1 by restricting to K'_1. U'_2 is similarly defined as the machine $\langle K'_2, \{0,1\}, \delta'_2 \underline{1}_2, \underline{52}_2 \rangle$ where $K'_2 = \{\underline{1}_2, \ldots, \underline{16}_2\}$.

emerge when we attempt to do so. In particular, the structure of the individual states comprising these sequences is quite distinct. For on the one hand U_1' moving its head from the right end of w to the right of new string it is constructing, using an auxiliary symbol $(*)$ as a place holder to keep track of its position in w. And on the other hand U_2' first moves its back and forth repeatedly over w to create a copy of this string to its right, using $*$ as place folder, and then proceeds to move back and forth repeatedly over the copied string using # as a place holder. Certainly it is possible to try to either correlate these states individually in grouping shorter than $\vec{\rho}'(w)$ and $\vec{\tau}'(w)$ themselves. But there seems to be no way of doing so in an canonical or even defensibly non-arbitrary manner.[24] It thus seems that realist will be better served by adopting the view that an "intended" bisimulation linking U_1 and U_2 – i.e. one that characterizes their operation in terms of subsequences which carry out the individual steps of the procedures which they are claimed to implement – must link $\vec{\rho}'(w)$ and $\vec{\tau}'(w)$ directly.

But if we conclude that R^* must be defined so as to allow $\vec{\rho}'(w)$ that $\vec{\tau}'(w)$ can be linked directly, the realist faces another significant problem. For note that we have that $len_{U'}(n) = 2n^2 + 9n + 2 = |\vec{\rho}'(|w|)|$ and $len_{U2'}(n) = \lceil 5n^2/2 \rceil + 16n + 8 = |\vec{\tau}'(|w|)|$ where $n = |w|$. And thus if these sequences are to be related by $S^*U1, U2$, then R^* must be formulated so as to allow quadratic length sequences to simulate one another. However, this contradicts the conclusion reached above that if we wish to exclude the sequence simulation $S^*_{S1,U1}$ which links $S1$ and $U1$, then R^* must be defined to prohibit such simulations.

This is the first instance in which we have seen explicitly that the desire to satisfy the positive requirements (3.10) and the negative requirements (3.11) come into direct conflict. For note that the problem we have encountered has arisen precisely due to the competing constraints to define $\underline{\leftrightarrow}$ so that it is broad enough to accomodate $U_1 \underline{\leftrightarrow} S_2$ and narrow enough to accomodate $U_1 \underline{\nleftrightarrow} U_2$. As I announced in Section 1, however, I believe this problem to be pandemic to the enterprise of attempting to defend algorithmic realism

[24]The central problem here is that the structure of states comprising $\vec{\rho}'(w)$ and $\vec{\tau}'(w)$ is sufficiently different that any particular partitioning into subsequences must be rationalized on the basis of a specific provision built into R^*. For if we were to allow any partitioning of these sequences then under R^*, then the realist will be again be hard put to block various other unintended simulations which will exist between otherwise unrelated machines.

via the abstractionist strategy. But for a variety of reasons, the current example may not be taken as a satisfactory demonstration of the universality of this problem. At the very least, the reader might still reasonably suspect that the realist can still provide a definition of bisimulation which will satisfy all of the extensional adequacy conditions by further modification of the representation requirement R^*.

This is, of course, true in the trivial sense that it is possible to simply define $\underline{\leftrightarrow}$ "by hand" on the machines in $\boldsymbol{T}$ so as to satisfy (3.10) and (3.11). In order to do this he would have to proceed as follows: 1) define arbitrary sequence decomposition functions $g_1, \ldots, g_5$ for $S_1, \ldots, S_5 \in \boldsymbol{T}_1$ and $h_1, h_2 \in \boldsymbol{T}_2$; 2) define S^* to be the smallest equivalence relation which held between the subsequences induced by the operation of $S_1, \ldots, S_5$ as respectively partitioned by the $g_1, \ldots, g_5$ and also the subsequences induced by the operation of U_1, U_2 as respectively induced by h_1, h_2; 3) define $R^* = S^*$. Such a definition is, of course, stipulative in the extreme since the definitions of neither S^* nor R^* are based on more general principles about either the structure of the machines in $\boldsymbol{T}$ or background intuitions about computational equivalence. And thus while it is the desirable of satisfying the extensional adequacy conditions (3.10) and (3.11), it has the presumably disastrous feature of ruling all pairs of machines T_1, T_2 which are not members of $\boldsymbol{T}$ inequivalent.

The considerations adduced thus far suggest that if the realist adopts $\mathcal{T}$ as his choice of computational model $\mathcal{M}$, then he will be hard pressed to define $\underline{\leftrightarrow}$ so that it even satisfied the extensional adequacy criteria (3.10) and (3.11). But as things stand, it is as yet unclear whether this is due to a fundamental problem with the abstractionist strategy or merely reflects the limitations of the model $\mathcal{T}$ with which we have attempted to work it out in this chapter. Although I believe that the difficulty can ultimately be seen to reside with the abstractionist strategy itself, we have already noted several properties of $\mathcal{T}$ which appear to suggest that it is in fact a poor choice of computational model with respect to which to implement the abstractionism.

This observation stands somewhat in contrast to my initial observation that despite its simplicity, the one-tape, one-head Turing machine model does seem well tuned to providing implementations of algorithms which solve decision problems over strings. But it should now be clear that there are clear operational constraints on the members of $\mathcal{T}$ which prevent

them from directly carrying out what otherwise seem like simple computational operations such as storing or retrieving a value from a storage location accessed by numerical index. As we have, this it is essentially this feature which prevents the machines in $\boldsymbol{T}$ from carrying out single operations such as checking the symbols a_i and $a_{n-(i-1)}$ for equality. Since such operations are subsumed into a single stages in the operation of Pal1 and Pal2, this necessitates the use of various indirect mechanisms for performing such checks. And as we have seen, the execution of these mechanisms is computationally expensive in the sense of requiring long sequence of intermediate Turing machines steps. And more germanely to the problems particular to the abstractionist strategy, it also necessitates that we go to elaborate lengths to subsume the co-implementing machines under the same definition of computational equivalence.

On the basis of the computational overhead entailed by adopting $\mathcal{T}$ as our choice of background model, the reader might reasonably surmise that this class of implementations can be ruled out as a reasonable choice for $\mathcal{M}$ on the basis of some precisely characterizable limitation on its ability to carry out certain kinds of intuitively straightforward computations. In the next chapter, I will cite a variety of formal results which suggests that this is in fact the case. But before abandoning the current setting entirely, however, I wish to attempt to illustrate the depth of the difficulty we face in accounting for the relationship between the machines in $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ on the basis of one final example.

By way of motivation, note that all of the examples of "unintended" simulations we have seen thus far have been trivial in the sense that they attempt to correlate all (or almost all) of an execution of one of the machines in $\boldsymbol{T}_1$ with one of the machines in $\boldsymbol{T}_2$. Thus far, I have only argued that the realist faces a substantial problem in stating general principles which allow even such trivial simulations from being excluded by an appropriately general definition of bisimulation. But it is also possible to come at the situation from the opposite direction and demonstrate that there are indeed simulations between the machines in these two classes which correlate fine-grained subsequences of states in a manner which has at least the initial appearance of demonstrating a true form of computational equivalence.

In order to see this, it will be useful to consider a new Turing machine T which is not in either of $\boldsymbol{T}_1$ or $\boldsymbol{T}_2$ but which can be shown to serve as a sort of computational interporlant

between them. (3.7). T is given in Figure (3.7). As should be immediately apparent, we

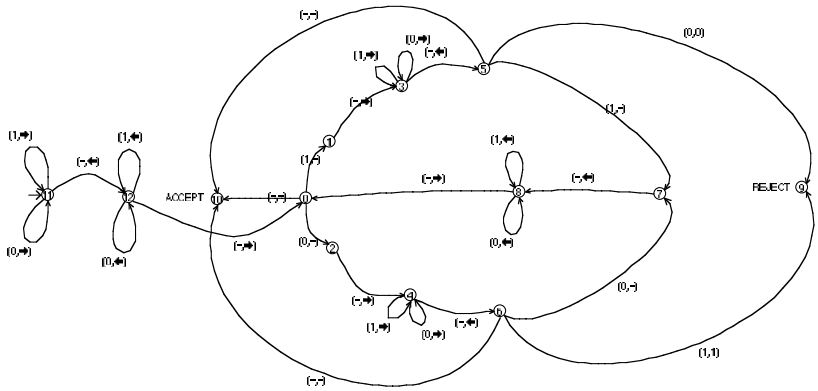

Figure 3.7: The machines T depicted as a graph.

may look upon it as being derived from S_1 by adding states $\underline{11}_6$ and $\underline{12}_6$, which respectively move T's head to the right end of the tape and then back to the left prior to start the normal operation of S_1. As such, T is most naturally described as a trivial variant of S_1 containing a computational detour in the style of S_3.

For this reason, it follows that any extensionally adequate definition of bisimulation ought to satisfy the additional the positive requirements that

(3.10′) $S_i \underline{\leftrightarrow} T$ for all $i \in \{1, 2, 3, 4, 5\}$

And in order to see that this condition can indeed be accommodated relative to both our current definition of bisimulation and the constraints we thus far contemplated placing on R^*, first note that an initial sequence of any execution $\vec{\nu}(w)$ of T will consist in a sequence of states of the form

(3.16)

$$\vec{\tau}'(w) \equiv \underline{11}a_1 \ldots a_n \longrightarrow a_1\underline{11}a_2 \ldots a_n \longrightarrow \cdots \longrightarrow a_1 \ldots a_{n-1}\underline{11}a_n \longrightarrow a_1 \ldots a_n\underline{11} \longrightarrow$$
$$a_1 \ldots a_{n-1}\underline{12}a_n \longrightarrow \ldots \longrightarrow \underline{12}a_1 \ldots a_n \longrightarrow \underline{12} - a_1 \ldots a_n \longrightarrow \underline{0}a_1 \ldots a_n$$

Over the course of this sequence, T moves its head rightward until it encounters a blank symbol, then leftward until it until encounters another blank symbol, and then into the state $\underline{0}a_1 \ldots a_n$ which corresponds to the initial state of S_1.

It is now easy to see how to construct a sequence simulation $S^*_{S1,T}$ linking S_1 and T. In particular, first define the decomposition function g_1 for S_1 as $g_1(w,i) = 1$ for i and $g_2(w,i) = 2|w| + 1$ if $i = 0$ and $g_2(w,i) = 1$ otherwise. We may now define the relation $S^{*,0}_{S1,T}$ so that it relates initial sequences of S_1 of the form $\langle \underline{1}_1 1 a_2 \ldots a_n, \underline{2}_1 - a_2 \ldots a_n \rangle$ with sequences of T of the form $\vec{\tau}(1a_2, \ldots a_n) \cdot \underline{1} - a_2 \ldots a_n$ and initial sequences of S_1 of the form $\langle \underline{1}_1 0 a_2 \ldots a_n, \underline{2}_1 - a_2 \ldots a_n \rangle$ with initial sequences of T of the form $\vec{\tau}(0a_2, \ldots a_n) \cdot \underline{2} - a_2 \ldots a_n$. $S^*_{S1,T}$ can now be defined as the union of $S^{*,0}_{S1,T}$ and the relation $S^{*,1}_{S1,T}$ which for all $\mu_1, \mu_2 \in St$ is defined by condition

(3.17) if $\mu_1 \xrightarrow{1} \mu_2$ then $\langle \mu_1, \mu_2 \rangle \; S^*_{s1,t} \; \langle \hat{\mu_1}, \hat{\mu_2} \rangle$ where if $\mu \equiv u \underline{\mathsf{k}}_1 v \in St_1$, then

$\hat{\mu} = u \hat{\underline{\mathsf{k}}} v \in St_T$ where $\hat{\underline{\mathsf{k}}}$ is the member of St_T with index corresponding to that of $\underline{\mathsf{k}}$.

Described informally, $S^*_{S1,T}$ correlates single initial transitions in executions of S_1 (which will always be from state $\underline{0}_1$ to either state $\underline{1}_1$ or state $\underline{2}_1$) with length $2|w|+1$ sequences of transitions in the execution of T which correspond to its initial back and forth movement across its input string transacted in states $\underline{11}_T$ and $\underline{12}_T$. Since after going though these transitions, T behaves identically to S_1, all other transitions may be correlated one a one-to-one basis as given by $S^{*,1}_{S1,T}$. And finally we may note that although $S^*_{S1,T}$ relates $O(1)$ length sequences with $O(|w|)$ length sequences, such a simulation must presumably be allowable under any representation requirement meeting the conditions discussed above.

Although slightly more involved, it is also straightforward to see that T bisimulates with at least one of the machines in $\boldsymbol{T}_2$. This is most readily seen with respect to U_1 by defining another sequence simulation $S^*_{U1,T}$. In order describe this simulation, first note that complete executions $\vec{\tau}(w)$, $\vec{\rho}(w)$ of T and U_1 on the input can be naturally decomposed as $\vec{\tau}(w) = \vec{\tau}_0(w) \cdot \vec{\tau}_1(w) \cdot \ldots \cdot \vec{\tau}_i(w) \cdot \ldots \cdot \vec{\tau}_m(w)$ and $\vec{\rho}(w) = \vec{\rho}_0(w) \cdot \vec{\rho}_1(w) \cdot \ldots \cdot \vec{\rho}_0(w) \cdot \ldots \cdot \vec{\rho}_m(w)$. Here $\vec{\tau}_0(w) = \vec{\tau}'(w)$ (as defined in (3.16) and $\vec{\nu}_0(w) = \vec{\nu}'(w)$ (as defined relative to the operation of the submachine U' above) and for $i > 0$, $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ respectively denote the subsequences of $\vec{\tau}(w)$ and $\vec{\rho}(w)$ during which the symbols a_1 and $a_{n-(i-1)}$ are compared by T and S_1. Note that if all of these pairs of symbols match we will have $w \in L_{pal}$ and hence also $m = n$. If, however, j is least such that $a_j \neq a_{n-(j-1)}$, then we will have $m = j$.

There are thus four distinct structures which the sequences $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ may have

depending on the values of a_i and $a_{n-(i-1)}$. These may be enumerated as follows:

(3.18) i) If $a_i = a_{n-(i-1)} = 1$, then

$$\vec{\tau}_i(w) = (-)^{i-1}\underline{1}_1 1 a_i \ldots a_{n-i} 1 (-)^{i-1} \xrightarrow{*T} (-)^{i-1}\underline{8} - a_{i+1} \ldots a_{n-i}(-)^i \text{ and}$$

$$\vec{\rho}_i(w) = \#(-)^{i-1}\underline{13}_1 1 a_{i+1} \ldots a_n (-)^i 1 a_{n-(i-1)} \ldots a_1 \xrightarrow{1*} \#(-)^i \underline{23}_1 a_{i+1} \ldots a_n (-)^{i+1} a_{n-i} \ldots a_1.$$

ii) If $a_i = a_{n-(i-1)} = 0$, then

$$\vec{\tau}_i(w) = (-)^{i-1}\underline{1}_1 0 a_i \ldots a_{n-i} 0 (-)^{i-1} \xrightarrow{*T} (-)^{i-1}\underline{8} - a_{i+1} \ldots a_{n-i}(-)^i \text{ and}$$

$$\vec{\rho}_i(w) = \#(-)^{i-1}\underline{13}_1 0 a_{i+1} \ldots a_n (-)^i 0 a_{n-(i-1)} \ldots a_1 \xrightarrow{1*} \#(-)^i \underline{23}_1 a_{i+1} \ldots a_n (-)^{i+1} a_{n-i} \ldots a_1.$$

iii) If $a_i = 1$ and $a_{n-(i-1)} = 0$, then

$$\vec{\tau}_i(w) = (-)^{i-1}\underline{1}_1 1 a_i \ldots a_{n-(i-2)} 0 (-)^{i-1} \xrightarrow{*T} (-)^i a_i \ldots a_{n-(i-2)} \underline{9}_1 0 (-)^i$$

$$\vec{\rho}_i(w) = \#(-)^{i-1}\underline{13}_1 1 a_{n-1} \ldots a_n (-)^i 0 a_{n-i} \ldots a_1 \xrightarrow{1*} \#(-)^{i+1} a_{n-i} \ldots a_n (-)^{i+1} \underline{24}_1 0 a_{i+1} \ldots a_1.$$

iv) If $a_i = 0$ and $a_{n-(i-1)} = 1$, then

$$\vec{\tau}_i(w) = (-)^{i-1}\underline{1}_1 0 a_i \ldots a_{n-(i-2)} 1 (-)^{i-1} \xrightarrow{*T} (-)^i a_i \ldots a_{n-(i-2)} \underline{9}_1 0 (-)^i \text{ and}$$

$$\vec{\rho}_i(w) = \#(-)^{i-1}\underline{13}_1 0 a_{n-1} \ldots a_n (-)^i 1 a_{n-i} \ldots a_1 \xrightarrow{1*} \#(-)^{i+1} a_{n-i} \ldots a_n (-)^{i+1} \underline{24}_1 1 a_{i+1} \ldots a_1.$$

These four cases respectively correspond to sequences of states undertaken by T and S_1 during their ith pass over the input string w during which the symbols are a_i and $a_{n-(i-1)}$ are compared. As the reader is invited to confirm, in each case not only are $|\vec{\tau}_i(w)|$ and $|\vec{\rho}_i(w)|$ of length linearly proportional to $n - i$, but the exact length of these sequences can be effectively encoded in sequence decomposition functions $g_1(w, i)$ and $g_2(w, i)$.

On the basis of the foregoing observations, we may define the sequence simulation $S^*_{T,U1}$ to be the smallest relation $S^* \subseteq St_T \times St_{U_1}$ which holds between $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ for all $w \in \{0, 1\}$ and $i \leq |w|$. It should also be evidence that in conjunction with g_1 and g_2, this simulation will satisfy Definition 3.4.4 together with all of the conditions on R^* considered above. For as noted, each of the sequences $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ for $i > 1$ will be of length $O(n)$. Without adducing additional constraints on R^* so as to rule out a bisimulation which relates $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ on the basis of some other features of their structure, it seems that the realist will have no choice but to admit that T and S_1 will be bisimilar.

The moral of this final example can now be summed up as follows. Collecting what we have just seen about the relationship between T, S_1 and U_1, note that it seems inescapable that we will have $T \underline{\leftrightarrow} S_1$ and $T \underline{\leftrightarrow} U_1$. But since we have been careful to ensure that

our refined definition of bisimulation is transitive, then without additional modification to this definition, we will also have $S_1 \underline{\leftrightarrow} U_1$, in direct contradiction of (3.11). We have now reached this conclusion on several distinct occasions with increasingly strict definition of bisimulation. In the prior instances, however, the relation which witnessed $S_1 \underline{\leftrightarrow} U_1$ was manifestly unintended in the sense that it failed to meaningfully decompose the executions of S_1 and U_1. In the current case, however, we can explicitly define a bisimulation S^* linking S_1 and U_1 by composing those linking S_1 and T and U_1 and T. As the reader is invited to confirm, S^* is closely related to the relation S^*_{T,U_1} defined above. In particular, it links functionally related subsequences in the executions of S_1 and U_1 essentially like those given in (3.18).

We may note in conclusion that the existence of such a relation does not automatically scuttle the what hope the algorithmic realist may have left in defining $\underline{\leftrightarrow}$ over $\mathfrak{T}$ so as to non-stipulatively satisfy (3.10) and (3.11). For in particular, it is open to him to attempt to refine the representation requirement R^* so as to rule out the simulation S^*. In this sense, S^* cannot be ruled as intuitively unintended, at least relative to the standards we have considered thus far. In seeking to disallow such a simulation, a realist will have to find some basis in which to prohibit linkages between state sequence such as those given in (3.18).

This may, of course, be accomplished citing such some feature particular to the individual machines S_1 and U_1 or their executions. But in this case there is again no guarantee that it will yield correct results outside the class of $\boldsymbol{T}$. And it may also be done by citing some more general structural features of the subsequences $\vec{\tau}_i(w)$ and $\vec{\rho}_i(w)$ – e.g. the fact that the members of $\vec{\rho}(w)$ but not $\vec{\tau}(w)$ contain an internal $-$ symbol separating sequences of 0s and 1s. Ultimately, however any specific proposal for how this might be accomplished has to be justified on the basis of more general principles not specifically tied to S_1 or U_1. And equally importantly, any prohibition of this sort also has to be shown to prohibit the existence of a simulation between pairs of machines such as S_1 and S_5 or U_1 and U_2 which implement the same algorithm but which we have seen to operate in a very different manner in a step-by-step sense. While a case study of the sort which we have just undertaken may itself be insufficient to rule out the possibility of giving such a definition in principle, it is at

least adequate to demonstrate the in-principle problems which stand in the way of giving one.

Chapter 4

On implementations

4.1 Lessons and morals

One of my principle goals in Chapter 3 was illustrate how easy it is to construct a wide class of formally distinct implementations of the same algorithm over a single model of computation. This is not necessarily a surprising observation, since many common models of computation are advertised precisely on the basis of their putative operational universality – i.e. their inclusion of machines whose executions mimic the operation of any effective procedure. But since many distinct procedures on one so-called "level of abstraction" can be viewed as implementing the same procedure described at a different level, it is hardly surprising that a computational model designed to directly reflect as many of the lower level procedures as possible will contain multiple intuitively adequate representations of the operations of a higher level procedure.

But it is not so much the general phenomenon of multiple implementability which is significant to prospects of algorithmic realism, but rather what it entails about the formulation of an adequate definition of bisimulation. For note that once the realist has nominated a class of implementations $\mathcal{M}$ relative to which he proposes to represent algorithms in accordance with the "abstractionist" program outlined in Chapter 2.3, he is then responsible for formulating a definition of bisimulation which satisfies both intensional and extensional adequacy conditions over this class. But the more heterogeneous the class of implementations $M_1, M_2, \ldots \in \mathcal{M}$ which can plausibly be claimed to implement a single algorithm A_1, the broader the definition of bisimulation will have to be. And the broader such a definition gets, the greater the danger that it will hold between one of the M_i and some other algorithm N_j which can plausibly be claimed to implement a distinct algorithm A_2.

Or so went the argument of Chapter 3.4. But however persuasive the reader may

have found this argument in the case of the particular algorithms and implementations studied in Chapter 3, there are also substantial reasons to think that the conclusion which I tentatively drew from it – i.e. that the realist will be unable to define *any* intensionally and extensionally adequate definition of bisimulation – has been drawn far too hastily. The fundamental problem is not that of attempting to generalize from a case study. For if indeed all the premises on which I based this argument are accepted – i.e. PAL1 and PAL2 really are distinct algorithms, $S_1, \ldots, S_5$ implement PAL1, U_1, U_2 implement PAL2, etc. – then the argument would indeed demonstrate the non-existence of an extensionally adequate definition of bisimulation. And thus given my prior argument that the so-called "abstractionist" strategy is in fact the only one open to the algorithmic realist, this set of affairs would be sufficient to show the view to be false.

But as I acknowledged early on in Chapter 3, the example to be considered was a indeed a toy, optimized for familiarity and ease of exposition. Its artificiality is evident in at least two respects: 1) not only is the decision problem for the language L_{Pal} computationally "easy," but the algorithms PAL1 and PAL2 are themselves far simpler than most procedures which are employed in computational practice to derive mathematically significant results; 2) the single-head, single-tape Turing machine model $\mathcal{T}$ which was used to implement these algorithms is also far simpler than most models of computation which are currently employed in computational practice. And it is thus fair to conclude that while the argument of Chapter 3 certainly does raise several concerns to which the algorithmic must reply, it is as yet unclear how substantial they are.

The purpose of this chapter and the next is to demonstrate that while the example of Chapter 4 certainly does suffer from these limitations, its moral for algorithmic realism is still generalizable. In this regard, we see immediately that the first of two artifacts about this example noted in the previous paragraph does not appear to rhetorically weigh in favor of the realist. Of course PAL1 and PAL2 certainly are far simpler to formulate than most algorithms which are of genuine utility in mathematical practice (e.g. Strassen's matrix multiplication algorithm, the AKS primality algorithm test, CYK context free parsing algorithm, Dijkstra's shortest path algorithm, etc.) and they are also far easier to prove correct then these other algorithms. However implementations of such non-trivial algorithms will

be substantially more complex than those of PAL1 and PAL2 both in the sense that the control structures they employ are more complicated and also in that they operate on more complex data structures. As these control and data structures must themselves must be implemented relative to a given choice of computational model, the greater complexity of these algorithms gives rise to a greater potential for multiple forms of implementation overall. And thus rather than simplifying the task of formulating an adequate definition of bisimulation, it appears that the need to accommodate less trivial algorithms will only exacerbate many of the problems encountered in Chapter 3.[1]

But in seeking to determine whether the conclusions of Chapter 3 can indeed be generalize, we must also weigh carefully the contribution of the second assumption mentioned – i.e. the fact that the example considered was worked with respect to the model $\mathcal{T}$ of single head, single-tape Turing machines. In taking stock of the significance of this assumption, it will be useful to briefly recall the purpose which the choice of a class of implementations $\mathcal{M}$ is supposed to play with respect to the abstractionist strategy. Recall in this regard that we concluded in Chapter 2 that all reference to algorithms must be mediated by instances of the schema "the algorithm implemented by M" which we have long since agreed to formalize via the function $imp(\cdot)$. The ostensible purpose of this function is to map implementations into the algorithms which they implement. But I argued that our intuitions about the values of this function were sufficiently limited to offer little hope that it could be explicitly defined. And I therefore concluded that our best chance of understanding how it was possible to refer to individual algorithms was to view this function as implicitly defined by a so-called abstraction principle.

In Chapter 2, I originally formulated the statement which I suggested the algorithmic realist employ in this capacity as follows:

(4.1) $imp(M_1) = imp(M_1) \Longleftrightarrow M_1 \underline{\leftrightarrow} M_2$

[1] Although these remarks appear quite qualitative, they can be made precise along several dimensions. For instance, we may note that there is no nesting of iterative control structures in PAL1 and PAL2 (i.e. an occurrence of a control structure like While do ... od within another such structure. However, implementations of Strassen's algorithm will typically have at least two such nestings. We may similarly note that although PAL1 and PAL2 operate directly on the input strings which are supplied as their inputs, many other algorithms use more complex data structures. For instance,the CYK algorithm uses techniques from dynamic programming which require the maintenance of an auxiliary table of strings, and Dijkstra's algorithm requires the maintenance of an auxiliary Fibonacci heap.

And I also argued that such a principle could be taken to have the conceptual function analogous to that of traditional abstraction principles such as that for the governing *direction* – i.e.

(4.2) $Dir(\ell_1) = Dir(\ell_2) \Longleftrightarrow \ell_1 \parallel \ell_2$

I argued in Chapter 2 that such a statement can be taken as serving the dual rule of fixing the truth conditions of statements of the form $Dir(\ell_1) = Dir(\ell_2)$ and also of delimiting the domain of directions as the range of the function $Dir(\cdot)$. In this setting, it is standard to speak of the relation of parallelism as *grounding* the abstract sortal *direction* on the base of the fact parallelism bears some conceptual or semantic relation to that of direction.

Of course spelling out exactly what relation parallelism must bear to the concept direction in order for (4.2) to serve these roles is something of a vexed problem itself (as witnessed by the dispute about the putative analyticity about the corresponding principle for natural numbers – i.e. Hume's principle). And of course one of the morals of Chapter 3 is that if we want to treat *algorithm* as an abstract sortal in a similar manner, then it will be difficult to identify a relation which entails the correct identity conditions for statements of the form $imp(M_1) = imp(M_2)$. But what I want to concentrate on initially, however, is not this aspect of justification of (4.1), but rather on the proposed formulation of this statement itself. For note that in (4.1), the variables M_1 and M_2 are implicitly assumed to vary across implementations just as the variables ℓ_1 and ℓ_2 are assumed to vary across lines in (4.2). But of course if we want to consider adjoining either (4.1) or (4.2) to a formal theory such as T_p as discussed in Chapter 2.3, we must obviously modify them so that their scope is appropriately restricted.

In the case of (4.1), this is most readily achieved by introducing a predicate $Impl(x)$ which is meant to hold of a mathematical structure x just in case x is of the appropriate type to serve as an implementation. In this case (4.1) could be restated in the form

(4.3) $\forall x \forall y[(Impl(x) \wedge Impl(y)) \rightarrow (imp(x) = imp(y) \Longleftrightarrow x \underline{\leftrightarrow} y)]$.

If we assume that this principle is to be evaluated over a mathematical domain (say that of a model $\mathfrak{A}$ of set theory), then the idea is that the predicate $Impl(x)$ is supposed to hold

of just those structures which correspond to potential implementations.[2] The minimum qualification for a definition of such a predicate is that it hold of just structures which we treat as implementations in the course of our computational practices (i.e. individual Turing machines, RAM machines, etc.) and which thereby allow us to refer to algorithms. But per Chapter 2.3.2, this class appears to be open ended in the sense we often define new forms of implementations. It thus seems that providing an enumerative definition of the notion of implementation based on extant model of computation is thus likely to provide only a temporary, stipulative answer to this question.

One problem which the realist faces in regard to defining $Impl(x)$ is if this class is defined too narrowly, the realist will be unable to account for the status of expressions of the form $imp(M)$ where M corresponds to an implementation which does not fall under a given definition. However there is a more substantial background problem. For if the class of implementations somehow turns out to resist a precise mathematical definition, then the ability of a principle like (4.3) to delimit a well-defined class of abstract objects may also be called into question. For if the domain $\mathcal{M}$ of $imp(\cdot)$ is not well-defined, such a principle can hardly be taken to implicitly determine a class of objects as its range.

The upshot of these considerations is that if (4.3) is to actually serve the role which is envisioned by the algorithmic realist, the definition of $Impl(x)$ must function as something like a conceptual analysis of the notion of implementation. *Pace* Chapter 2.2.3.2, we cannot simply look to theoretical computer science to provide such a definition. For, in particular, the notion of implementation is generally only determined by providing examples of models of computation without at the same time specifying what they have in common by virtue of which their members are properly classified in this manner. However, one of my purposes in this chapter will be to dispel the impression that the realist's situation in this regard is hopeless. For in particular, as I hope to show, sufficient common structure *can* be found among common models of computation such that we can at least attempt to extract a general definition.

---

[2]The same effect of limiting the scope of objects whose identity conditions are given by (4.1) can also obviously be achieved by other devices – e.g. by adopting a multi-sorted language or defining $imp(x)$ so it that was undefined on values of x which were not implementations. But note that in both cases, we would still face the task of delimiting the domain of objects suitable to serve as implementations.

Returning now to the significance of the fact that in Chapter 3, $\mathcal{M}$ was stipulated to be $\mathcal{T}$, one problem can be immediately observed. For even though we do not yet have a good sense as to how $Impl(x)$ should be defined and thus also cannot explicitly define $\mathcal{M}$ at this point, it is quite evident that $\mathcal{T}$ will comprise a proper subset of this class. With respect to the relevant examples, for instance, it is easy to envision implementations of PAL1 and PAL2 which are other than Turing machines – for instance we will see below that these algorithms are naturally implementable as simple forms of RAM machines operating on binary strings. And since the necessity of taking other forms of implementation of these algorithms into account will presumably induce different pressures on the appropriate definition of $\underline{\leftrightarrow}$, it may be that the essentially negative conclusion of Chapter 3 that no non-stipulative extensionally inadequate definition is possible will need to be rethought.

In order to get an impression of why this may be so, it will be useful for the time being to suppose that while reference to algorithms PAL1 and PAL2 can in principle be effected by expressions of the form $imp(M)$ for M drawn from an arbitrary model of computation $\mathcal{Q} \subseteq \mathcal{M}$, it is up to realist to choose which model is actually to be employed for this purpose. Under this assumption, the difficulties we encountered in defining $\underline{\leftrightarrow}$ over $\mathcal{T}$ may be seen as demonstrating not that no such definition is possible in general, but merely that $\mathcal{T}$ is not an appropriate model by which to mathematically ground our understanding of these algorithms.

More precise technical reasons can be cited to support this view. For observe that much of the trouble which arose in defining $\underline{\leftrightarrow}$ so that it correctly accounted for the operational affinities among the members of the classes $S_1, \ldots, S_5$ and U_1, U_2 were due precisely to the fact that Turing machines are unable to access squares of their tape by numerical index in the manner indicated by PAL1 and PAL2. This is to say that while we informally describe PAL1 as proceeding on input $a_1, \ldots, a_n$ by making the comparisons $a_1 \stackrel{?}{=} a_n, \ldots, a_{\lfloor n/2 \rfloor} \stackrel{?}{=} a_{n-(\lfloor n/2 \rfloor - 1)}$, these comparisons cannot be transacted directly by members of $\mathcal{T}$. Rather, machines in this class must use an indirect method to compare these symbols, such as storing the leftmost symbol in its local state and moving to the right end of the tape (in the case of S_1) or "pushing" the leftmost symbol to the right-hand end (in the case of S_5). Although the comparisons described above are transacted by PAL1

in a single step, machines like S_1 or S_5 require $O(n)$ steps to transact the sort of indirect methods just described. Since n comparisons must be made overall, this means that these machines will have overall running time $O(n^2)$, as opposed to the $O(n)$ running time of PAL1 itself.

These calculations obviously apply only to S_1 and S_5. But the fundamental observation on which they are based – i.e. that a member of $\mathfrak{T}$ cannot compare the values of symbols written on arbitrary squares of distance of n from one another in fewer than $O(n)$ steps – can be used to prove a general theorem. In particular, it may be shown that for all machines $T \in \mathfrak{T}$, if T decides L_{Pal}, then the worst case running time of T is at least $\Omega(n^2)$.[3] This result, first obtained by Hennie and [61], is one of several well-known *lower bound* results for $\mathfrak{T}$. Results of this sort show that individual Turing machines cannot be used to compute certain functions in asymptotic running time less than $O(f(n))$. For instance, an $\Omega(O(n \log(n)/(\log(\log(n)))^2)$ lower bound has been established for computing multiplication "on line" which holds not just for $\mathfrak{T}$, but also for the class of multiple-tape Turing machines which I will discuss below.[4]

Although these are purely technical results, they potentially have substantial foundational significance to how we should understand the notions of implementation and model of computation. For note that not only are there informally described decision algorithms for L_{pal} with $O(n)$ running time – e.g. PAL1 and PAL2 – but we will see below that there are models of computation which contain concrete implementations of these algorithms which preserve this complexity. And similarly, there are known to be multiplication algorithms which have sufficiently clever implementations which have running time $O(n)$.[5]

One obvious consequence of these results is that the value of the function $imp(\cdot)$ applied

---

[3]In somewhat more detail, it can be shown that for any machine $T \in \mathfrak{T}$, if T decides L_{pal}, then there exists a string w such that $len_T(w) > k|w|^2$ for some $k \geq 1$. This result was first shown using a combinatorial argument by Hennie. It may also be proven by the so-called *incompressibility method* from Kolmogorov complexity – cf. [75].

[4]In the on line computational paradigm, a machine T computing a function $f(n)$ whose inputs and outputs are represented in binary must produce the nth digit of its output before it scans the $n+1$st digit of its input. The result cited is due to Cook and Aandreaa [22].

[5]For instance, Schönhage [122] showed that there is a storage modification machine which multiplies in time $O(n)$. Several other $O(n)$ multiplication algorithms are now know which are based on fast Fourier Transforms – cf. [24].

to a machine $T \in \mathcal{T}$ cannot be taken to be the identity function. For since the realist presumably wishes to hold that $imp(S_1) = \textsc{Pal1}$, it would following from the fact that he also (presumably) wishes to maintain that "Pal1 has running time $O(n)$" is true, we would be able to conclude that "S_1 has running time $O(n)$" contrary to the quadratic lower bound cited above. But this result is hardly surprising since we have already noted that for arbitrary implementations M, we generally cannot have $imp(M) = M$ as there will typically be distinct implementations M_1, M_2 for which we are willing to accept $imp(M_1) = imp(M_2)$.

More generally, however, lower bound results point to the fact that while the class $\mathcal{T}$ is obviously universal in the extensional sense that it contains a member which computes every function determined by an intuitively effective algorithm, it may not be operationally universal in the sense of containing a machine which directly mirrors the mode of operation of every informally specified algorithm A. It is, of course, difficult to state such a claim precisely because it is seems difficult to formalize the relevant notion of operational universality. I have, however, argued that the attribution of asympoptic running time complexity to individual algorithms is one of the most important components of our practices of taxonomizing informally specified algorithms according to their operational properties. And thus the fact that $\mathcal{T}$ does not contain a machine which matches the running time of even simple algorithms like Pal1 and Pal2 serves as fairly strong evidence that $\mathcal{T}$ is not universal in this sense.

Of course running time complexity is only one of many properties which we attribute directly to algorithms in practice. As I have previously noted, for instance, we also speak of individual algorithms as operating on or employing various sorts of data structures (e.g. "Strassen's algorithm takes a matrix as input," "Heapsort uses a priority queue,"), and computational techniques ("Mergesort is recursive," "The CYK parsing algorithm uses dynamic programming," "Kruskal's algorithm is greedy"). It is, of course, something of an idealization to imagine that we can construct an exhaustive list of such properties. And some additional explanation is required in order to ensure that a property of this sort is appropriately applied to an algorithm rather than an implementation.[6]

[6]For instance, it is unclear whether the analogously formed statement "Dijkstra's algorithms uses a Fibonacci heap" should be taken as a claim about a particular algorithm (i.e. Dijkstra's method of finding

An operationally universal model of computation in the sense alluded to above should correspond to one in which contains a member $Q \in \mathfrak{Q}$ which reflects all of the computational properties of every informally specified algorithm. In making such a characterization more precise, it will be useful to retain the logical terminology developed in Chapter 2. For recall that I suggest there that the use of such statements such as those appearing in the previous paragraph is sufficiently regular and conventionalized that we can at least imagine formulating an axiomatic theory T_p in which these statements would correspond to theorems of the form $\theta(A)$ where $\theta(X)$ corresponded to a particular computational property such as having running time $n\log(n)$ or using a priority queue and A is a term denoting an individual algorithm. Using this terminology, the operational universality of $\mathfrak{Q}$ can be characterized as follows:

(4.4) For all informally stated algorithm $\mathcal{A}$ and for all $\mathfrak{L}_p$ definable properties $\theta(X)$, if $T_p \vdash \theta(a)$ (where a is a $\mathfrak{L}_p$-term denoting A), then there exists a $Q \in \mathfrak{Q}$ such that $T_p \vdash \theta^*(q)$ (where q is an $\mathfrak{L}_p$-term denoting Q and $\theta^(x)$ is an appropriate mathematical analysis of the mathematical property $\theta(x)$ with respect to $\mathfrak{Q}$).

The formal necessity of replacing $\theta(X)$ with $\theta^*(x)$ derives from the fact that $\mathfrak{L}_p$ was originally described as a two-sorted language with variables $X_1, X_2, \ldots$ ranging over algorithms and $x_1, x_2, \ldots$ ranging over mathematical objects such as implementations. We may note more generally, however, that many of the computational properties which we apply directly to algorithms will need to be mathematically reformulated before they can be applied to members of $\mathfrak{Q}$ in a manner which may depend on the features of this model itself. For instance, it would be natural in practice to speak directly of Kruskal's algorithm as having the property of greediness. [7] Of course when we speak in this manner, we attach a definite meaning to this property (i.e. that the algorithm in question operates by making

---

shortest paths in a weighted directed graphs $G = \langle V, E\rangle$) or merely an implementation thereof. For note that pseudocode specifications of this algorithm typically only specify that this algorithm operate on a priority queue Q. However, if we assume that Q is employed in the standard manner such that extracting its minimum takes $O(|V|)$ steps, then Dijkstra's algorithm runs in time $O(|V|^2+E)$. However, this algorithm is often reported to have running time $O(|E| + |V|\log(|V|))$, although only if we assume that Q is itself implemented as a Fibonacci heap.

[7] e.g.: "Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight." [24], p. 504

a locally-optimal choice at each stage with the hope of finding the global optimum), but it is to the algorithm itself which we attribute the property.

But consider a given implementation of this procedure as a `MIX` machine M which represents G as an adjacency matrix A which, as per [72], is itself implemented as a doubly linked list of weights (i.e. wherein each element $A[j,k]$ belongs to the lists $A[j,1],\ldots,A[j,n]$ and $A[1,k],\ldots,A[n,k]$). In such a case, it obviously makes no sense to apply the predicate "greedy" directly to M as it will have no predetermined sense relative to such a machine. But we can imagine what it would take to analyze our informal understanding of the $\mathcal{L}_p$ predicate $\gamma(X)$ (i.e. "X is greedy") into a complex open sentence $\gamma^*(x)$ over $\mathcal{L}_m$. Roughly speaking, such a sentence would say that if m denotes a `MIX` machine, then m implements an array A of the form described above which functions as its input and another array B which functions as its output. In addition, $\gamma^*(x)$ would stipulate that at every step of m's main loop, it attempts to place the entry of A corresponding to a minimal weight unattached edge of G into the array B.

The necessity of analyzing computational properties in this manner substantially complicates the characterization of what we might mean by saying that one model of computation was more operationally universal than another. For in particular, there seems to be little hope that it would be possible to construct a *uniform* translation $(\cdot)^*$ taking primitive predicates of $\mathcal{L}_p$ into open sentences of $\mathcal{L}_m$ (one which was effectively determined over the entire language $\mathcal{L}_p$) even if we allow for such a map to be parameterized in $\mathfrak{Q}$. And thus it seems that in order to formulate a principle like (4.4) will require that we construct analyses of each $\mathcal{L}_p$ predicate "by hand."

This observation detracts from the hope that a principle like (4.4) can serve as a truly robust thesis about the representability of algorithms relative to a given model of computation. However I intend to largely overlook the problem potentially posed by the necessity of adopting an ad hoc property translation function. For as I now want to briefly discuss, the possibility of demonstrating a principle like (4.4) appears to provide the realist with a promising way out of many of the difficulties encountered in the previous chapter.

For suppose that it were possible to demonstrate (4.4) with respect to a particular model of computation $\mathfrak{Q}$. In this case, the algorithmic realist would be assured that for

every informally specified A, there existed at least one implementation $Q \in \mathfrak{Q}$ which, as we might put it, *faithfully represented* A in the sense of reflecting its computational properties (as represented relative to the translation funciton $(\cdot)^*$). Now for familiar reasons, it will most likely not be open to the realist to simply identify A with Q. For it certainly does not follow from the fact that $\mathfrak{Q}$ contain *at least* one representative of A that it contains *exactly one*. And from this it follows that if the realist seeks to reduce algorithms to members of $\mathfrak{Q}$, he will still presumably have to proceed according to the general abstractionist strategy outlined in Chapters 2 and 3 – i.e. by defining an equivalence relation $\underline{\leftrightarrow}$ over $\mathfrak{Q}$ and then arguing that it has the appropriate properties.

But as I will attempt to demonstrate over the rest of this chapter, if $\mathfrak{Q}$ is chosen appropriately, then there appears to be hope that this task can be accomplished in a manner which avoids many of the problems encountered in Chapter 3. For looking back on the examples considered there, we may now note that the task of defining $\underline{\leftrightarrow}$ over $\mathfrak{T}$ was complicated precisely by the fact that the Turing machines which we considered were anything but direct implementations of Pal1 and Pal2. For as I noted above, each of these machines had to implement the task of comparing symbols at opposite ends of the input string by performing auxiliary operations which had little to do with the specification of Pal1 and Pal2 themselves. But it was due to the extra states and transitions which were required to support these operations that we were ultimately able to define "unintended" bisimulations linking pairs of machines like S_1 and U_1 which putatively implemented distinct algorithms.

If the realist were able to locate a model $\mathfrak{Q}$ whose members were able to directly support the sorts of operations and structures which appear in informal specifications of Pal1 and Pal2, the necessity of taking such computational detours could possibly be eliminated. And in this case, it is at least reasonable to hope on the realist's behalf that the classes of machines $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ which we judged to be implementations of Pal1 and Pal2 over $\mathfrak{Q}$ are substantially more homogeneous than the classes $\boldsymbol{T}_1 = \{S_1, \ldots, S_5\}$ and $\boldsymbol{T}_2 = \{U_1, U_2\}$. And in this case, we might correspondingly hope that $\underline{\leftrightarrow}$ could be defined in a manner so that it related all pairs of machines $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ without relating any pairs across these classes.

In more general terms, the strategy on offer to the algorithmic realist is thus as follows.

Rather than attempting to define $\underline{\leftrightarrow}$ across the class of *all* mathematical structures $\mathcal{M}$ which might be classified as implementations, he may take as a new goal that of identifying some proper subclass $\mathcal{Q}$ of $\mathcal{M}$ which can be shown to contains at least one "natural" or "direct" representative of every informally specified algorithm in the sense of (4.4). He can then proceed by attempting to define $\underline{\leftrightarrow}$ over $\mathcal{Q}$. And as described in Chapter 3.2, he will again be responsible for demonstrating both that this relation is extensionally adequate in the sense of tracking our intuitions about which machines implement the same algorithms and also intensionally adequate where this property may now be understood in two consistent of two components: 1) $Q_1 \underline{\leftrightarrow} Q_2$ may be understood as an analysis of what we mean by saying that the implementations Q_1, Q_2 "work the same way" in the sense of Chapter 2.3; 2) this relation serves as a congruence with respect to properties provable of $imp(Q_1)$ – i.e. if $T_p \vdash \theta(imp(Q_1)$ and $Q_1 \underline{\leftrightarrow} Q_2$, then $T_p \vdash \theta(imp(Q_2))$. And if such a definition is indeed obtainable, he may then claim to have reduced algorithms to mathematical objects in the sense of having constructed an implicit definition of the function $imp(\cdot)$, now interpreted as a mapping from members of $\mathcal{Q}$ into algorithms.

Over the rest of this chapter, I will set out to survey the realist's options in defining $\mathcal{Q}$ so that it has the appropriate properties to render this task feasible. In so doing, however, we are faced with lingering conceptual and technical problems. For on the one hand, if the realist is to adopt the strategy outlined in the previous paragraph, then it follows that he will be left without a means of accounting for the meaning of expressions of the form "the algorithm implemented by M" where M is some bona fide implementation which is not contained in $\mathcal{Q}$. As I suggested above, this is presumably a problem in the sense that although he may wish to associate algorithms with mathematical objects by taking them to be appropriate equivalence classes over $\mathcal{Q}$, it will still presumably be allowable to make reference to them by using machines drawn from other classes.

There are a variety of ways a proponent of the strategy just outlined might attempt to respond this to situation. For instance, the realist might attempt to deny the supposition that all forms of implementation really are created equal with respect to the task of making reference to algorithms. On this basis, he might argue that although Turing machines can be used to refer directly to certain simple algorithms on strings, the considerations

advanced above suggest that procedures like PAL1 and PAL2 are already beyond the power of such implementations to describe in a determinate manner. And on the other hand he might attempt to adopt a "multi-tier" theory of procedural reference whereby reference to members of one class of implementations $\mathfrak{Q}'$ (e.g. Turing machine) is claimed to sufficient to effect reference to the members of $\mathfrak{Q}$ which may be then be used to refer to algorithms, with indeterminacy possible at the first but not the second step.

In either case, however, the class $\mathfrak{Q}$ will be given a preferred position in the general theory of algorithmic reference and ontology which the realist hopes to develop. And for this reason, it will be incumbent upon him not simply to define this class, but also to offer some sort of explanation for why this should be so. For if he fails to do so, he leaves himself open to the familiar charge of ontological arbitrariness in the sense that he will not have offered any positive explanation as to why algorithms should be identified with mathematical objects relative to $\mathfrak{Q}$ as opposed to some other class of implementations. And for this reason, if he follows the strategy just outlined, he must also provide an argument that $\mathfrak{Q}$ is a maximally general or otherwise "natural" class of implementations by which to directly represent algorithms.

As we will see in the next chapter, the two most outspoken proponents of algorithmic realism – i.e. Yiannis Moschovakis and Yuri Gurevich – not only proceed according to this strategy, but they also attempt to make precisely this form of argument in favor of the classes of their preferred classes of implementations. These correspond respectively to the class of *recursors* $\mathfrak{R}$ and the class of *Abstract State Machines* $\mathcal{ASM}$. From both a technical and a conceptual perspective these models are very different. And as such, they certainly cannot *both* be the maximally general or natural medium in which to represent individual algorithms.

At this point, we are not yet in a position to see what sort of desiderata can be brought to bear to argue either for against such a claim. However, the recursor and ASM models may both be located within a more general framework for classifying model of computation according to both their formal properties and their affinities to other traditional models. Over the course of the rest of this chapter, I will attempt to develop a general framework for comparing and contrasting models of computation which will allow us to do this. In

particular, I will proceed by first attempting to provide an abstract characterization of the notion *model of computation*. And on this basis, I will then propose that virtually all extant models can be naturally classified as one of three types which I will respectively refer to as *transition-based*, *register-based* and *recursion-based* models.

Along the way, I will also offer a variety of observations about the various desiderata about models which have already come up in this section. For an arbitrary model $\mathcal{Q}$ these include the following: 1) $\mathcal{Q}$'s ability offer "direct" models of informally specified algorithms in the sense of (4.4); 2) the prospects of defining an adequate definition of bisimulation over $\mathcal{Q}$; and 3) the formal relationships in which members of $\mathcal{Q}$ stand to members of other models of computation. This will be significant for it should now be apparent that in order to carry out the modified abstractionist programme which I have just described, the realist will want to choose $\mathcal{Q}$ so that it is maximally direct in the sense of 1), admits to the narrowest possible adequate definition of bisimulation in the sense of 2), and is maximally general in the sense of assimilating other models in the sense of 3).

4.2 On models of computation

I will henceforth assume that all implementations are members of broader classes of mathematical formalisms traditionally known as *models of computation*. This notion has arisen several times previously. But in accordance with tradition, I have elected to simply illustrate the notion of a model of computation with examples rather than attempting to provide anything resembling a general definition or analysis. One of the greatest obstacles to doing so is the sheer diversity of formalisms which are conventionally characterized as models of computation. As mentioned in Chapter 2.2.3.2, many of the best known formalisms which are now accepted as falling under this concept grew out of foundational investigations of the notion of effective computation as briefly discussed in Chapter 1.3. As such, it is possible to understand what these models have in common in virtue of their initial motivation with respect to the analysis effective computability. But since the majority of the models which are now in common use were not proposed with foundational aims in mind, the task of unifying them under a single characterization is substantially more complex.

A number of authors have attempt to provide longitudinal surveys of major classes of models of computation, e.g. [118], [63], [149]. The formalisms surveyed in these sources give rise to at least three major classes of models which I consider individually below. Since many of the models are parameterized with respect to one or more of their formal components (e.g. the number of tapes or heads of a Turing machine, the number of accumulator registers of a RAM machine, etc.), in a formal sense there are at least countably many distinct models of computation currently recognized in the literature of theoretical computer science.

Although there are substantial structural difference between the major classes of models, they also share certain structural affinities which allow us to talk about them in a quasi-uniform manner. Extending the account given in Chapter 2, I will henceforth assume that a model of computation $\mathcal{Q}$ is specified by giving a class $\boldsymbol{Q}$ of mathematical structures whose members are conventionally referred to as *machines* (although we will see below that there are many instances in which this is a poor choice of terminology). As we have seen with the special case $\mathcal{Q} = \mathcal{T}$, a machine $Q \in \boldsymbol{Q}$ is to be thought of as inducing a function between sets X and Y which we can think of respectively as the class of objects which members of $\mathcal{Q}$ take as input and produce as output. The other standard component of a model of computation is what I will refer to as a definition of *application* – i.e. a specification of a function $App_{\mathcal{Q}} : \boldsymbol{Q} \times X \to Y$ which takes an individual machine $Q \in \boldsymbol{Q}$ and input value $x \in X$ and returns the value which we would normally describe as that derived by applying Q to x if this computation halts, and is undefined otherwise. Given a particular definition of $\boldsymbol{Q}$, $App_{\mathcal{Q}}$ is thus generally defined in a manner which attempts to provide an abstract mathematical characterization of the execution induced by applying Q to x.

The structure which I will refer to as the model $\mathcal{Q}$ itself will thus have the form $\langle \boldsymbol{Q}, App_{\mathcal{Q}}, X, Y \rangle$. And as in Chapter 2, I will often abuse notation by speaking of individual machines Q as being members of $\mathcal{Q}$ as opposed to $\boldsymbol{Q}$. It is generally straightforward to explicitly present various prototypical examples of models of computation in this form. And given this fact, it is also possible to go about characterizing the general notion of model of computation by simply enumerating all currently defined models characterized in this standard form.

But this approach turns out to have the serious disadvantage that this structural characterization of a model of computation does not even come close to providing sufficiently stringent conditions to rule out formalisms which are unlikely to be accepted as serving the role of such a model in practice. In particular, the characterization provided above does not to enforce that the definition of $App_{\mathcal{Q}}$ is given in a manner which analyzes what we mean informally by applying a machine to an input. In other words, a definition of this functional ought to reflect the fact that we typically think of applying a machine Q to x as corresponding to a procedurally mediated process whereby the value Q operates on x over the course of a sequence of intermediate steps. It is, for instance, such a characterization which allows us to reject structures such as $\mathcal{N} = \langle \mathbb{N}^{\mathbb{N}}, App_{\mathcal{N}}, \mathbb{N}, \mathbb{N} \rangle$ wherein the definition of application is given by extensional function application (i.e. $App_{\mathcal{N}}(f, n) =_{df} f(n)$) as serving as intuitively accepted models of computation. As such, any general characterization of this notion which does provide a pretext for doing so cannot be accepted as providing an adequate definition of this concept.

The fundamental conceptual problem with providing such a definition is, of course, characterizing the circumstances under which we are willing to accept that a definition of $\mathcal{Q}$ as characterizing the operation of a model which we would standardly speak of operating *in time*. For note that to say the definition of $App_{\mathcal{Q}}$ ought to reflect the fact that applying Q to x leads to a computational *process* is to say not only that such an application can be characterized as a sequence of states $\sigma_0(x), \sigma_1(x), \ldots$ but also that we generally view the members of such a sequence as being generated one after another. At first glance, this may seem to stand in contrast to the assumption that models of computation are themselves mathematical structures and that their members must perforce be understood as atemporal and non-spatial. It thus may initially seem puzzling on what basis we can ascribe temporal-like features to some models and not others.

There is, however, no reason to think that this is an insurmountable problem. However, in order to solve it we must take a step backwards and ask what it is that we expect implementations to be model of. The answer which the realist is presumably trying to work out is that the members of a given model of computation ought to be thought of as models of individual algorithms. And as I have observed previously in Chapter 2, we

standardly do speak of algorithms as operating in time. So in this sense, a tentative analogy may be drawn between, on the one hand, an algorithm and a model of computation and on the other, a mathematical model of a physical dynamical system, and the physical system it is used to represent.

It is important to keep in mind, however, that the execution of an algorithm A is unlike a physical or biological process in that it does not concern the evolution of a system composed of physical constituents but rather is better described as a process by which mathematical structures are sequentially transformed so as to yield a sequence of new structures. We also often speak of such transformations as occurring. This impression is encouraged by the equally conventional practice of speaking of the execution of A as being mediated by an abstract computing agent who explicitly follows instructions to perform certain transformations. But as I argued in Chapter 2, talk of this sort is best seen as figurative both in the sense that it is possible to specify an algorithm in a manner which does not consist of such instructions and also in that many of the properties attributable to actual agents are irrelevant to how we reason about algorithms in the abstract.[8]

In elucidating the use of the term "model of computation" to describe a formalism which is capable of representing the operation of an informally specified algorithm, we must start off by isolating those characteristics of our informal descriptions of algorithms which, upon more extensive reflection, we take to have genuine temporal significance. I started to do this in Chapter 2 where I argued that what is of central theoretical importance in computer science is our ability to associate with each informal procedure A and input x a metric called the *running time complexity of A on x*.

To a first approximation, this quantity can be said to measure the time it takes to carry out A on x until it halts and returns a value (presuming that it does). However in order to ascribe such properties uniformly, we must first agree on some means of determining what is to count as a primitive computational operation of A. This will result in a enumeration of mathematical "actions" $\alpha_1, \ldots, \alpha_n$ such as incrementing a counter, adjoining an edge to a

[8]For instance, we make no allowances for a computing agent getting tired or making a mistake. And in reasoning about procedures in a general setting, we are also willing to think of such an agent as being completely free of particular resource bounds.

graph, etc. which in typical cases correspond to the denotation of mathematical expressions in terms of which A has been specified. However, if we were to think of A being carried out by an actual computation agent operating on concrete representations of mathematical structures, we could then go on to assign a definite duration d_i to the task of carrying out action α_i. And on this interpretation, the running time of A on x would thus correspond to an actual temporal duration D derived by summing the d_i's in accordance to the number of times they are executed in the course of carrying out A on x. But of course this is not at all the way we actually apply the notion of running time complexity in practice. What we actually do is to treat all of the actions $\alpha_1, \ldots, \alpha_n$ as having unit cost. And on this view, the running time of A on x is a unitless quantity N derived by simply summing the number of times each of the α_i is carried during the operation of A on x.

Suppose we write $t_A(x)$ to denote this sum. Then in somewhat greater generality, the aspect of our discourse about running time which is taken to be significant concerns the running time complexity of A as a function of the size of x. The relevant notion of size will vary with the structure of the input set X as can be given by a metric $|\cdot| : X \to \mathbb{N}$. For instance, a standard size metric for strings w over a finite alphabet Σ^* is the length of w and a standard metric for natural numbers n is $\lfloor \log_2(n) \rfloor$ which corresponds to the length of n's binary representation. Subject to a choice of metric, we are generally most interested in the rate of growth of the derived function $time_A : \mathbb{N} \to \mathbb{N}$ defined as $time_A(n) = \max\{t_A(x) \ : \ |x| = n\}$ which gives A's maximal (or "worst case") running time for all inputs of size $|n|$. Such running times are generally reported using so-called asymptotic or "big-O" notation where the class $O(f(n)$ is defined to be the class of all functions $g : \mathbb{N} \to \mathbb{N}$ such that there exists an n_0 and c such that for all $n > n_0$, $f(n) < cg(n)$. As we saw in Chapter 2, it is through the use of this notation by which we standardly ascribe running times to individual algorithms when we say, for instance, "MERGESORT has running time $O(n \log(n))$" (meaning that $time_{\text{MERGESORT}}(n) \in O(n \log(n))$).

Thus although we do standardly ascribe temporal properties to individual algorithms A, this is done only in the abstract manner just described. In particular, we standardly factor out the actual durations which would be required to transact the various operations

in terms of algorithm A may be specified. And through the use of a size metric and asymptotic notation, we also abstract away from the contribution of additive and multiplicative factors which appear in $t_A(x)$. As such, it turns out that relatively little is required of an implementation Q in order to serve as a model of the temporally significant properties of A.

In order to see concretely what is required of an implementation in order to serve in this capacity, we must note one more consequence of our informal understanding of what it means to execute A. I have already noted that an execution of A on x may be taken to correspond to a sequence of stages $s_O(x), s_1(x), \ldots$ Implicit in this understanding is that stages are temporally ordered at least in the sense that it makes sense to speak of stage $s_i(x)$ as occurring before stage $s_{i+n}(x)$ for $n > 0$. Note that this is a necessary condition to impose on our analysis of an execution since an algorithm will typically be defined in such a way that the action it performs at stage $s_{i+1}(x)$ may depend on aspects of the structure of this stage which themselves will have been determined by actions carried out at stages *before* the $i + 1$ step in its execution. Also note that since we speak of the execution of A for a particular $x \in X$ as beginning with a particular stage whose structure will typically depend on x, we are also justified in recognizing every execution as having a unique *first stage*. And finally, the stages of A's execution can standardly be *discretely ordered* in the sense that if $s(x)$ precedes $s'(x)$ in the execution of A on x, then there will exist a stage $s(x)''$ occurring after $s(x)$ but before or at the same time as $s'(x)$ for which there exists no stage occurring after $s(x)$ but before $s(x)''$. This allows us to speak of one state as being the *immediate predecessor* of another. And thus in the most general setting, we can thus view A's operation on x as corresponding to a discrete partial order $<_{A,x}$ with minimal element $s_0(x)$.

If we make the additional assumption that A is also *deterministic* – i.e. for every stage $s(x)$ there is at most one stage $s(x)'$ occurring immediately after $s(x)$ relative to $<_{A,x}$ – then $<_{A,x}$ will have the structure of a discrete linear order of the form $s_0 <_{A,x} s_1 <_{A,x} s_2 <_{A,x} \ldots$.[9]Since there will generally be no a priori guarantee that A will halt on input x, such an ordering may have an infinite order-type. But even when we take this into account, we can still define a partial function $t_A : X \to \mathbb{N}$ which takes elements of X and returns

the length of the sequence of stages corresponding to A's execution on x if this sequence is finite and is otherwise undefined.

Given this characterization of what it means for a informal procedure A to operate in time, our next task is to analyze what it means to say that a member Q of a class of mathematical structures $\mathfrak{Q}$ can be used as a *model* of A's operation. As is now familiar, this may generally be accomplished by finding some means of formalizing A's stages as a class of mathematical structures St and its stage-by-stage operation as a transition function $\delta : St \rightarrow St$. To a first approximation, the execution of A on x corresponding to the sequences of stages $s_0(x), s_1(x), \ldots$ may thus be modeled in the conventional manner as a sequence of states of the form $\sigma_0(x), \sigma_1(x), \ldots$. Note, however, that whereas we would naturally use the temporal notions *before*, *immediate predecessor*, etc. to describe the sequence of stages $s_0(x), s_1(x), \ldots$, the corresponding properties of $\sigma_0(x), \sigma_1(x), \ldots$ may be defined directly in terms of the structure of Q. For instance to say that state σ occurs before σ' in the execution of Q on x is simply to say that there are i and m such that $\sigma = \sigma_i(x)$, $\sigma' = \sigma_{i+m}(m)$ and to say that σ is the immediate predecessor of σ' is to say that $\delta(\sigma) = \sigma'$.

In this sense, a model of computation $\mathfrak{Q}$ can now be characterized as a class of mathematical structures for which a formal notion of execution is definable relative to which we can interpret the temporal properties we apply when we describe the operation of an algorithm informally. But since such models are indisputably abstract, this means that temporal language can only be used to describe the operation of its members in an indirect or metaphorical sense. I have just indicated how this may be done in a particularly simple case, essentially by likening the transformation of one stage in the execution of an algorithm into another to the iteration of a transition function. However, this is by no means the only way in which to draw an analogy between the abstract mathematical operations in terms of which the machines in $\mathfrak{Q}$ are defined and the temporally interpretable actions out

[9]I adopt this restriction primarily here primarily as an expository expedient since it will allow us to considerably curtail the class of models which must be considered. The status of non-deterministic algorithms actually in computational practice is, however, quite interesting in large part precisely because it is unclear what ought to be counted as a implementation of such an algorithm. A thorough examination of this issue would require substantial digressions into both non-deterministic models of computation and their relationship to deterministic ones.

of which an algorithm's operation may be described.

The major classes of models of computation mentioned above can be seen as arising from different views about how such an analogy can be developed. These correspond to models based on a fundamental notion of state transition (of which the Turing machine model considered in the previous is paradigmatic) models based on a fundamental notion of assignment (of which the RAM machine and its variants are paradigmatic) and models based on recursion (on which Gödel-Kleene general recursive definitions are paradigmatic). One of the best known facts about these models (or more accurately, their paradigms just cited) is that they are also computationally universal in the standard extensional sense discussed above. But over the course of the next three sections, I hope to demonstrate that extensional equivalence results of this form belie operational distinctions. In particular, I will suggest that these three classes do not stand as equals with respect to satisfying the three desiderata which I suggested at the end of the last section ought to guide the algorithmic realist's selection of a model of computation by which to carry out the modified abstractionist programme described above.

4.3 Transition based models

One of the most straightforward ways to account for how an instance Q of a mathematical model of computation $\mathfrak{Q}$ can serve as a model of an algorithm A whose execution we would informally describe as taking place in time is to define the structure of Q so that it mirrors that of a physical dynamical system $\hat{Q}$ whose evolution actually does occurs in time. In order to facilitate formulating an analogy, it is useful to assume that Q may be described as a transition system. In particular, I will assume that $Q = \langle X_Q, Y_Q, \Sigma_Q, \delta_Q, H_Q, in_Q, out_Q \rangle$ and that $App_{\mathfrak{Q}}(Q, x)$ is defined as in Chapter 2.4.

In order to liken the execution of Q to the temporal evolution of the a physical system, we must demonstrate the potential existence of a concrete dynamical systems $\hat{Q}$ with the following three properties: 1) $\hat{Q}$'s phase space is described as a set $St_{\hat{Q}}$ of physical states such that for every abstract state $\mu \in St_Q$ there is a physical state $\hat{\mu} \in St_{\hat{Q}}$ whose physical structure can be taken to represent the mathematical structure of μ; 2) for every state of

the form $in(x)$ (for $x \in X_Q$), the physical state $\hat{in}(x)$ corresponds to an initial state of $\hat{Q}$; 3) when initiated from a initial state $\hat{in}_Q(x)$, the temporal evolution of $\hat{Q}$ under the physical dynamical laws may be divided into discrete steps of duration d seconds indexed by times $t = 0, 1, \ldots$ and such that evolution of $\hat{Q}$ from time t to $t+1$ is given by a function $\delta_{\hat{Q}} : St_{\hat{Q}} \rightarrow St_{\hat{Q}}$ such that $\delta_{\hat{Q}}(\hat{\mu}) = \hat{\mu}'$ if and only if $\delta_Q(\mu) = \mu'$. Where $\hat{Q}$ to be start in an initial state of the form $\hat{in}(x)_{\mathfrak{Q}} = \hat{\mu}_0(x)$, its subsequent evolution would take the course $\hat{\mu}_0(x), \hat{\mu}_1(x), \ldots$ where $\hat{\mu}_{i+1}(x) = \hat{\delta}(\hat{\mu}_i(x))$. In such a case, this sequence would unfold in actual time in the the sense that it would be stated as a definite moment t_0 with $\hat{Q}$ in state $\hat{\mu}_0(x)$ and such that the state $\hat{\mu}_{i+1}(x)$ would obtain (literally) d seconds *after* $\hat{\mu}_i(x)$.

To the extent which the relationship between Q and $\hat{Q}$ can be made precise, the existence of such a physical model of Q of this sort would provide an obvious basis for grounding the use of temporal language to describe the operation of Q. But of course it is well known to be difficult to specify even sufficient conditions for $\hat{Q}$ to represent Q in this sense and presumably harder yet to demonstrate a general statement about the circumstances under which such a system can be guaranteed to exist for all Q in a specified model $\mathfrak{Q}$.[10] But my present goal is not to show that the members of common models of computation have or even could have physical models of the sort just described but rather to explore different contexts in which it is possible to systematically justify the use of a certain kinds of mathematical formalism in describing computational processes which we informally describe as taking place in time. And in this regard, the potential availability of the set of physically inspired analogies between certain kinds of transition systems and physical dynamical systems has historically been seen as a paradigm case of such a framework.

The system of analogies which likens computational states to physical states and computational transitions to physical transitions mediated by dynamical laws can seen as providing a criterion for membership in an historically important class of models of computation which I will refer to collectively as *transition based models*. We have already met the paradigm example of this class in the form of the model $\mathfrak{T}$ consisting of single tape, single

[10]The problems which arise here are both well known and largely orthogonal to our present concerns. For a general discussion of the difficulties which arise when wish to precisely formulate the sort of relationship which must obtain between Q and $\hat{Q}$ in order for the latter to be justifably employable as a modeling of the former in the sense described in the previous paragraph, cf., e.g., [58].

head Turing machines. Starting in the 1950s, a number of models were proposed which sought to generalize this model. The earliest of these retained the finite control mechanism and possessed the same basic sort of computational medium – i.e. a read/write tape or grip divided into cells. However, they generalized beyond the specific properties of $\mathcal{T}$ by allowing such devices to have multiple heads or tapes. These abstractions led naturally to the definition of multi-tape and multi-head Turing machines and machines with two dimensional tapes as described in, e.g., [63]. These models in turn served as the basis for another round of generalizations which retained the finite control structure of a Turing machine, but generalized more radically on a Turing machine's tape by replacing it with different forms of finite labeled graphs upon which different "local" operations could be performed such as adding or deleting nodes or edges or changing the labeling of a path. This latter class includes the *Kolmogorov machines* of Kolmogorov and Uspensky [68], the *storage modification machines* of Schönage [121] and the *K-graph* machines of Sieg and Byrnes [130].

Since all of these latter day models were explicitly motivated by the creators as generalizing on various features of the definition of $\mathcal{T}$, there is a historical precedent for taking them to form a natural class. But this observation alone does not provide a basis for presenting a definition of this class of models which usefully distinguishes it from register-based models. For instance, we will see below that both transition- and register-based models can readily be represented as transition systems. And thus if we wish to distinguish these paradigms with respect to their suitability for providing a general characterization of a class of models which an algorithmic realist can take to serve of the class $\mathcal{Q}$ described above, some additional feature of transition based models must be cited which distinguishes them from register-based models.

Although I will ultimately suggest that certain aspects of this distinction must ultimately be regarded as vague in the sense that there are models which can be naturally classified as falling into either class, it is at least possible to provide a uniform basis for regarding the individual models mentioned above as forming a natural class. And as I now wish to argue, this possiblity is grounded in the plausibility with which the physically inspired analogies can be applied to these models. I will ultimately argue that it is this

property which makes them poor candidates to serve in the role of $\mathfrak{Q}$. In order to get some impression of why this is so, it will be useful to start out by briefly examining the sense in which the paradigm model $\mathfrak{T}$ is often cited as an example of a so-called "mechanistic" or machine-like model of computation.

Consider an arbitrary machine $T \in \mathfrak{T}$ and let $M_T = \langle X_T, Y_T, St_T, \Delta_T, H_T, in_T, out_T \rangle$ be the transition system based on T in the sense of Definition 4.2 of Chapter 3. As we have seen, St_T will be comprised of structures of the form $u\underline{\mathsf{k}}v$ where u and v are abstract string types over Σ, $\underline{\mathsf{k}} \in K$ represents one of finitely many local states of T's finite control mechanism (as formalized by Δ_T, which we can assume is single valued) and the position of $\underline{\mathsf{k}}$ relative to u and v encoded the position of T's head on its tape. It is a common observation that such states can be taken to reflect the spatial organization of the states of a (notional) physical states system $\hat{T}$ which consists of an unbounded physical tape on which tokens of finitely many physical symbols types $\hat{\Sigma}$ corresponding to the symbols in Σ may be written and overwritten. $\hat{T}$ will also contain a physical subsystem consisting of a finite control mechanism interpretable as a set of states $\hat{K}$ such that $|\hat{K}| = |K|$ and a physical read/write head which moves along $\hat{T}$'s tape. Characterized in this manner, the physical states of $\hat{T}$ can be represented as sequences $\hat{u}\hat{\mathsf{k}}\hat{v}$ where $\hat{u}$ and $\hat{v}$ are physical strings comprised of physical symbol tokens, $\hat{k} \in \hat{K}$ and the relative physical position of $\hat{u}$, $\hat{v}$ and $\hat{k}$ represent the spatial locates of $\hat{T}$'s tape and read/write head. Let $\hat{St}_T$ represent the class of physical structures of this form. We must additionally assume that $\hat{T}$ is such that its dynamical evolution ia determined by the physical dynamical laws is be given by a physical transition function $\hat{\Delta}_T : \hat{St}_T \rightarrow \hat{St}_T$. Finally, we must require that if $\Delta_T(u_1\underline{\mathsf{k}}_1v_1) = u_2\underline{\mathsf{k}}_2v_2$, then the system $\hat{T}$ satisfies all counterfactuals of the form: "if at time t_i $\hat{T}$ is in state $\hat{u}_1\hat{\underline{\mathsf{k}}}_1\hat{v}_1$, then at time t_{i+1} $\hat{T}$ will be in state $\hat{u}_2\hat{\underline{\mathsf{k}}}_2\hat{v}_2$."

Treating the availability of a physical interpretation of this sort as a basis for categorizing $\mathfrak{T}$ as a model of an informally described computational process which we originally described as occurring in time differs substantially from that originally provided by Turing [144]. For recall that Turing attempted to justify the properties $\mathfrak{T}$ not in terms of their physical interpretablity, but rather relative to the potential actions of an idealized computing agent. Such an agent – which following Gandy [43] I will refer to as a *computor* – can be thought of

as working in actual time but without being subject to any physical limitations other than those imposed by his own cognitive capacities. Turing thus stipulates that the computor operates in a step by step manner on a quasi-abstract (i.e. non-physical, but presumably *not* atemporal or static) computing medium with the same structural properties as a Turing machine's tape. Consequently the analogies which are open to him in explaining why we are justified in describing Turing machines as operating in time at all depend on a prior analysis of the abilities of such a notional computing agent.

As a number of commentators have pointed out, the specific constraints which Turing chooses to impose on the computor are closely tied to the specific foundational (or more broadly, epistemic) considerations which were important during in the 1930s.[11] This is most evident in his description of the computor as capable of being in only finitely many "states of mind" (which serves as the informal motivation for treating a Turing machine's state as a member of the finite set K), in the narrow "window" of symbolic configurations to which he may attend in selecting his next action (which serves as the informal motivation for allowing δ to depend only on the contents of the currently scanned square) and on the narrow class of symbolic operations he can undertake (which serves as the informal motivation for allowing δ to change only the scanned square or to move the head a single square left or right).

These decisions can readily be justified in the context of an attempt to delineate operations which conservatively fall under the pre-theoretic notion of effectivity whose proper definition was up for debate in the 1930s. But since our current desire is to characterize models of computation relative to the more general goal of explaining why we are justified in using certain mathematical formalisms as models of temporally extended computational processes, there is no reason why we must remain faithful to Turing's original motivations in characterizing $\mathfrak{T}$'s properties. In fact in this broader context, there are a number of reasons to prefer the quasi-physicalistic analogies we have been considering over Turing's quasi-psychological ones. The first of these derives from the fact that Turing's computor must still be understood as abstract at least in the weak sense of not corresponding to an

[11] For discussion, see [129], [43], [26].

actual physically embodied agent. Consequently we face the same sort of interpretive challenges in making sense of temporal discourse involving his actions as we do for statements about the operation of strictly mathematical models – e.g., we must still explain what it means for such a agent to operate in time but to be free of the sort of "performance" limitations such as fatigue, breakdown, recourse bounds, etc. which would effect the operation of an actual computing agent. And second, even if we do retain Turing's psychologistic interpretation of the features of the computor, it seems doubtful that any general description of such an agent's abilities could could ever serve to *uniquely* determine a mathematical model of computation.

It thus follows that although each of the models mentioned above can be motivated on the basis of positing computing agents with greater and greater calculating abilities, it is difficult to isolate anything about the notion of computor qua computing *agent* which serves to unify them. For although we may figuratively speak of a computor in a state μ as explicitly following a rule that tells him to assess the properties of μ by applying a predicate φ and then depending on the outcome to perform an action α to μ to yield a new state μ', the mere fact that the computor is characterized as an agent does itself entail that φ be effectively decidable or that the mapping induced by α is effective. If the computor were either mathematically omniscient or capable of directly performing various infinitary operations, then there would be nothing to prevent him from operating in a manner which could not be represented by any nominalistically possible physical process. Additional constraints must thus be imposed on how the computor operates to ensure either effectiveness or even the possiblity of physical representability.

Another problem with Turing's characterization is that not even he looked upon the specific constraints imposed on the computor's operation as constituting a principled upper bound on what such an agent should be able to do. For instance, there seems to be no *a priori* reason to suppose that the computor should only be to read or operate on the immediately adjacent tape cell or move the read/write head only one square at a time. Note, however, that if we seek to construct a revised model based on a more general characterization of such an agent's abilities, the fact that the computor is characterized as *agent* again seems to be of little help. Rather, what is required is an account of what

would make his operations suitably *mechanistic* – i.e. such that if the system of which the computor is part (which may be taken to include his "state of mind") is given by μ, then a subsequent state μ' (if any) which is brought about by his actions is determined by a finitely describable process free from creative insight or causal influences outside of the computational system. But note that once the class of allowable operations is described in these terms, it seems possible dispense with the computor in favor of a purely extension or non-psychologistic description.

Several theorists have attempted to do this by proposing a set of conditions which can be placed on the states and transition functions of a model of computation so as to guarantee that its operation is mechanistic in an intuitively plausible sense. In particular, Gandy [42], Sieg [129] and Gurevich [51] have all argued that four types of constraints are required in order to meet this requirement. I will refer to these respectively as the *discreteness*, *determinism*, *boundedness* and *locality* conditions.

Any model which can be characterized as a deterministic transition system will automatically satisfy the first two conditions.[12] For note that the discreteness of a model $\mathfrak{Q}$ is generally taken to mean that the states of its members Q may be described as abstract structures in a manner which allows their operation to be described by the iteration of a transition function Δ_Q on the class of states St_Q. This means that execution of Q is comprised of stages which may be indexed by finite ordinals, and thus that the evolution of Q in time may be likened to that of a discrete dynamical system rather than a continuous one. The determinism condition may be described informally as the constraint that a mechanistic computing device must operate so that at any point during its computation, its next state (if any) is completely determined by its current state. This can easily be enforced by requiring that Δ_Q is single valued – i.e. if $\langle \sigma, \sigma' \rangle, \langle \sigma, \sigma'' \rangle \in \Delta$, then $\sigma' = \sigma''$.

It is more difficult to provide equally general formalizations of the boundedness and

[12] All of the models which I have mentioned in this section may be straightforwardly presented as transition systems in the manner of Definition 3.3. This generally involves taking the original formulation of the model in question and showing that its computational medium (e.g. n tapes in the case of an n-tape Turing machine, a finite labeled graph in the case of storage modification machines and K-graph machines, etc.) can be combined with a relevant notion of local state so as to derive a definition of global state analagous to that defined for Turing machines in Chapter 4.3. For illustrations of how various other models can be subsumed under the definition of transition system, see [11] and [51].

locality conditions on mechanism. This is due to the fact that the properties which these constraints are intended to formalize cannot be defined solely in terms of the of the structure of a transition system. In fact, in order to even describe the constraints these conditions are intended to enforce on a model $\mathcal{Q}$, we must first assume that it is possible to define a function $cen_{\mathcal{Q}}$ and relations $Near^{a}_{\mathcal{Q}}$ and $Near^{c}_{\mathcal{Q}}$ with the following intended interpretations:

(4.5) a) The function $cen_{\mathcal{Q}}$ maps a complete states μ of a machine in $Q \in \mathcal{Q}$ into a structure $cen_{\mathcal{Q}}(\mu)$ which is a substructure of μ. The intention is that if $cen_{\mathcal{Q}}(\mu) = \nu$ then ν represents the portion of μ whose structure a machine Q may immediately query and modified in the course of a single transition. For this reason, I will refer to $cen_{\mathcal{Q}}(\mu)$ as the *center* or *computational locus* of μ. I will also denote the class of all such substructures of for all machines $Q \in \mathcal{Q}$ as $Cen_{\mathcal{Q}}$.

b) The relation $Near^{a}_{\mathcal{Q}} \subseteq St_Q \times St_Q$ is intended to hold of states μ_1 and μ_2 which are in computational proximity in the sense that μ_1 can be derived from μ_2 (or vice versa) by a single application of Δ_Q

c) The relation $Near^{c}_{\mathcal{Q}} \subseteq St_Q \times St_Q$ is intended to formalize the fact that if $Near^{c}_{\mathcal{Q}}(\mu_1, \mu_2)$ then μ_1 and μ_2 are in computational proximity in the sense that their centers $cen_{\mathcal{R}}(\mu_1)$ and $cen_{\mathcal{R}}(\mu_2)$ are in computational proximity to one another in a sense which will depend on the structure of St_Q.

The appropriate definitions of $cen_{\mathcal{R}}$ and $Near^{a}_{\mathcal{R}}$ and $Near^{c}_{\mathcal{R}}$ will obviously vary from one model of computation to another but are generally such that they can be immediately "read off" from a definition of a genuinely mechanistic model. If we assume that we have obtained a definition of these terms, then the more general intention behind the boundedness and locality constraints can now be explained as followed. The boundedness condition is supposed to ensure that operation of machines $Q \in \mathcal{Q}$ is finitary in the sense that the operation which is applied to a state $\mu_1 \in St_Q$ to yield $\mu_2 = \Delta_Q(\mu_1)$ requires only a finite amount of "local" information. This may be achieved by requiring first that $|Cen_{\mathcal{Q}}|$ be finite. And we must also ensure that $cen_{\mathcal{R}}(\mu_1)$ does indeed determine which operation α is selected by Q so as to derive $\Delta_Q(\mu)$ from μ. This can be achieved by requiring that Δ_Q

must be such that if $cen_{\mathcal{Q}}(\mu_1) = cen_{\mathcal{Q}}(\mu_2)$ and $\Delta_Q(\mu_1) = \mu_3$ and $\Delta_Q(\mu_2) = \mu_4$, then if μ_2 is derived by the applying α to μ_1, then μ_4 is also derived by applying of α to μ_3. Finally, the locality condition is intended to formalize the fact that if $\Delta_Q(\mu_1) = \mu_2$, then μ_2 is in computational proximity to μ_1 in both the sense that the components on which it differs from μ_1 are near μ in the sense of $cen_{\mathcal{R}}(\mu_1)$ and also that the center of μ_2 has not moved too far from that of μ_1. This can be achieved by requiring that if $\Delta_Q(\mu_1) = \mu_2$, then we must have $Near^a_{\mathcal{Q}}(\mu_1, \mu_2)$ and $Near^c_{\mathcal{Q}}(cen_{\mathcal{R}}(\mu_1), cen_{\mathcal{Q}}(\mu_2))$.

If we understand the intention behind the definitions of $cen_{\mathcal{Q}}$, $Near^a_{\mathcal{Q}}$ and $Near^c_{\mathcal{Q}}$ according to the quasi-spatial, quasi-causal interpretations given above, then the role of the boundedness and locality conditions with respect to formalizing the claim that a given model of computation is mechanistic may be explained as follows. First note that the boundedness condition is intended to rule out models whose members are able to make arbitrarily fine discriminations between states or can be aware of differences in a state which are arbitrarily distant from a machine's current computational locus. Similarly, the locality condition is intended to rule out models whose members are capable of transitions which result in "action at a distance" either in the sense of modifying components of their current state which are arbitrarily remote from their current computational locus or moving this locus itself an unbounded distance from its current position. Since both conditions have a clear physical interpretation, one might hope that if we can provide a suitably general way of formalizing the definition of $cen_{\mathcal{Q}}$ and $Near^a_{\mathcal{Q}}$ and $Near^c_{\mathcal{Q}}$, we would then have isolated individually necessary and jointly sufficient conditions which ensure the physical implementability of the model $\mathcal{Q}$ in something like the sense described at the beginning of this section.

The central challenge we face in proceeding in this manner is that our original definition of transition system imposes no constraints on the definition of a computational state. Thus if we take an arbitrary transition system Q, there is no guarantee that the state μSt_Q has the appropriate sort of structure so that $cen_{\mathcal{Q}}$, $Near^a_{\mathcal{Q}}$ and $Near^c_{\mathcal{Q}}$ can be defined in manner which is consistent with the intended physical interpretations of these terms. For note that the availability of these interpretations relies on our ability to view the structure of state μSt_Q as having a spatial structure to which notions like location and nearness of

constituents can be applied. But as things now stand, there is no guarantee that the elements of St_Q even have a compositional structure which we can be decomposed into subcomponents, let alone ones on which a natural metric structure can be imposed.

One possible way to proceed is to attempt to directly define the nearness relations and center function for models for which we already have an intended physical interpretation and then see how they might be generalized. This can readily be accomplished for $\mathcal{T}$ in its transition system presentation given in Chapter 4.3 since have already seen that structure of its states admits to a natural spatial interpretation. In particular, states $\mu \in St_T$ have the form $u_1 a_1 \underline{\mathsf{k}} b_1 v_1$ which directly encodes the geometric properties of T's tape in two senses: 1) uv gives the contents of T's tape as read from leftmost non-blank symbol to the rightmost nonblank symbol meaning that if $uv \equiv a_1 \dots a_n$, then symbols the a_i, a_j for $i, j \leq n$ are naturally interpreted as being at distance $|i - j|$ from one another; 2) if $u \equiv a_1 \dots a_m$ and $v \equiv a_{m+1} \dots a_n$, then T's head is naturally interpreted as being located over the $m+1$st symbol non-blank symbol counting from the left. On this basis, it is straightforward to formulate physically plausible definitions of $center_{\mathcal{T}}$, $Near^a_{\mathcal{T}}$ and $Near^c_{\mathcal{T}}$ as follows:

(4.6) a) $center_{\mathcal{T}}(u_1 a_1 \underline{\mathsf{k}} b_1 v_1) = \langle \underline{\mathsf{k}}, b_1 \rangle$;

b) $Near^a_{\mathcal{T}}(u_1 a_1 \underline{\mathsf{k}}_1 b_1 v_1, u_2 a_2 \underline{\mathsf{k}}_2 b_1 v_1) \Leftrightarrow$ i) $u_1 a_1 = u_2 a_2, v_1 = v_2$ or
ii) $u_2 = u_1 a_1, a_2 = b_1, b_2 = \epsilon, v_2 = v_1$ or
iii) $u_2 = u_1, a_2 = \epsilon, b_2 = a_2, v_2 = b_1 v_1$

c) $Near^c_{\mathcal{T}}(u_1 a_1 \underline{\mathsf{k}}_1 b_1 v_1, u_2 a_2 \underline{\mathsf{k}}_2 b_1 v_1) \Leftrightarrow |len(u_1 a_1) - len(u_2 a_2)| \leq 1$ and
$|len(v_1 b_1) - len(v_2 b_2)| \leq 1$

These definitions respectively record facts about the operation of a Turing machine which can be read off directly from Definition 3.1 – i.e. that if T is in global state $\mu = u_1 a_1 \underline{\mathsf{k}} b_1 v_1$, its next global state will be determined entirely by the local state $\underline{\mathsf{j}}$ and the scanned symbol b, that individual transitions of T can change μ only by changing $\underline{\mathsf{k}}$, overwriting b_1 or by moving the current head position at most one square right or left.

Definitions similar to these can be given for the models mentioned above as being inspire by $\mathcal{T}$. For instance, consider the classes $\mathcal{T}^2$ of two head Turing machines as defined by , e.g., [63]. The state of such a machine can be given by a string of the form $u\underline{\mathsf{k}}^1 bv\underline{\mathsf{k}}^2 dw$

where $u, v, w \in \Sigma^*$, $b, d \in \Sigma$ and the relative positions of $\underline{\mathsf{k}}^1$ and $\underline{\mathsf{k}}^2$ to these strings gives the position of the machines two heads.[13] Over the course of a single transition, a two headed machine $T^2 \in \mathcal{T}^2$ may move only one of its heads or write a single symbol on the square which one of them is scanning. The action undertaken by such a machine is hence local in the sense that it occurs at only one of the two head positions. But note that the decision as to which of these four types of actions (i.e. move the first head, move the second head, write on the square scanned by the first head, write on the square scanned by the second head) is determined on the basis of *both* the symbols b and d which the heads $\underline{\mathsf{k}}^1$ and $\underline{\mathsf{k}}^2$ are scanning.[14] This means that the definition of the center of a state of a machine $T \in \mathcal{T}^2$ thus must include both the local state and these two symbols – i.e. $center_{\mathcal{T}^2}(\mu) = \langle \underline{\mathsf{k}}, b, d \rangle$. And similarly, the definitions of $Near^a_{\mathcal{T}^2}$ and $Near^c_{\mathcal{T}^2}$ must respectively take into account that over the course of a single transition, a two-headed machine can overwrite either of the two symbols scanned by its head or move either of its heads one square to the left or the right.

Although the class $\mathcal{T}^2$ is one of the simplest possible generalizations of $\mathcal{T}$, it is apparent that the definitions of $center_{\mathcal{T}^2}$, $Near^a_{\mathcal{T}}$ and $Near^c_{\mathcal{T}}$ just sketched already represent a substantial abstraction away from the quasi-spatial interpretation of the corresponding definitions given above for $\mathcal{T}$. For note that over the course of a single transition of a machine $T^2 \in \mathcal{T}^2$, a state of the form $\underline{\mathsf{k}}^1 a_1 v_1 \underline{\mathsf{k}}^1 b_1 w_1$ can be transformed into one of the form $\equiv u_1 \underline{\mathsf{k}}^2 a_1 v_1 \underline{\mathsf{k}}^2 b_2 w_1$. In such a case, the value of b_2 may depend on the value of a_1 in the sense that if $a_2 \neq a_1$, we have $\delta_{T^2}(u_1 \underline{\mathsf{k}}^1 a_2 v_1 \underline{\mathsf{k}}^1 b_1 w_1) = u_1 \underline{\mathsf{k}}^2 a_1 v_1 \underline{\mathsf{k}}^2 b_3 w_1$ for some $b_3 \neq b_2$. Note, however, that as with $\mathcal{T}$, there is a natural geometric interpretation of the states of machines in $\mathcal{T}^2$ according to which the proximity of two symbols is given by the absolute

[13] The local state of members of $\mathcal{T}^2$ is defined in the same manner as $\mathcal{T}$ – i.e. as a member of a finite set K. But since a member of $\mathcal{T}^2$ machine has two distinct read/write heads, we must adopt a notational convention for distinguishing between them. Since our original convention for denoting the states of single-head machines already "overloads" the use of $\underline{\mathsf{k}}$ to indicate both the current location state and this head position, two distinct markers of this sort are required for $\mathcal{T}^2$. Relative to the convention just proposed, $\underline{\mathsf{k}}^1$ and $\underline{\mathsf{k}}^2$ should thus seen as distinct markers naming the same member of K.

[14] Formally, this corresponds to the fact that a transition function for a two head machine is defined to be a function $\delta : (K \times \Sigma \times \Sigma) \rightarrow (\Sigma \times \{1, 2\}) \cup \{\blacktriangleright_1, \blacktriangleleft_1, \blacktriangleright_2, \blacktriangleleft_2\}$ where the first two coordinates of its domain correspond to the symbols read by the machine's two heads and the range corresponds to a choice of actions parameterized for $i \in \{1, 2\}$ as follows: i) if $\delta(k, a_1, a_2) = \langle b, i \rangle$, then the symbol b is written on the square currently scanned by the ith head; ii) if $\delta(k, a_1, a_2) = \blacktriangleright_i$, then the ith head is move one square to right; iii) if $\delta(k, a_1, a_2) = \blacktriangleleft_i$, then the ith head is move one square to left.

difference of their position from the left most non-blank symbol. But the possibility of the sort of transition just exhibited demonstrates that if $\delta_{T^2}(\mu_1) = \mu_2$, then the value of a symbol in μ_2 may depend on the value of an arbitrarily distant symbol in μ_1. This means that $cen_{\mathcal{T}^2}$ must be defined to include tape squares which are arbitrarily distant from one another in the geometric sense and similarly that $Near^a_{\mathcal{T}^2}$ must be defined so that it relates states which differ in components which are arbitrarily distant from each other. And finally, since T_2 may also decide to move its second head (i.e. the one whose position is denoted by $\underline{\mathsf{k}}^2$) based on the value of a_1, $Near^c_{\mathcal{T}^2}$ must be defined so as accommodate the fact that one component of the center of the current state may move to a position which is arbitrarily distant from the other.

Without going into more detail about the exigencies of constructing a physical instantiation of a formal model of computation, it is hard to say whether the situation just described represents a substantial departure from the intended quasi-spatial and quasi-causal interpretation which was originally assigned to the boundedness and locality conditions. What is clear, however, is that if we look more broadly at the other models $\mathcal{M}$ of computation which are often described as generalizing $\mathcal{T}$, the respective definitions of $cen_{\mathcal{M}}$, $Near^c_{\mathcal{M}}$ and $Near^a_{\mathcal{M}}$ which must be given strain the limits of physically plausibility. One case in point is that of the Schönhage [121] storage modification machine model $\mathcal{S}$ which is sometimes presented as being a natural generalization of the Turing machine to the domain of graphs. A machine $S \in \mathcal{S}$ is comprised of a read-only input tape, a write-only output, a class of states corresponding to finite, labeled, directed graphs and a control mechanism allowing for the iterated application of various operations on the input and output tapes and the modification of the graph. In somewhat more detail, the states of S (which are known as Δ-*structures* where Δ is a finite set of symbols corresponding to edge labels) have the form $\langle X, a, p\rangle$. Here X denotes a finite set of nodes which comprise a graph whose edges are determined by the family of functions $p = \langle p_\alpha | \alpha \in \Delta\rangle$ which are of type $p_\alpha : X \to X$ where $p_\alpha(x) = y$ means that there is a directed edge from x to y with label α. Finally $a \in X$ is the so-called *center* of S is this node around which operations on X take place. This mode functions something like the head of a Turing machine which can be moved locally during the course of a computation.

One feature of the model $\mathcal{S}$ which complicates the task of formulating plausible boundedness and locality conditions concerns the possible operations in terms of which a machine $S \in \mathcal{S}$ may be defined. In particular, among the possible instructions which may be transacted over the course of a single application of the transition function Δ_S are the following: 1) add a new node to X connected to a via a sequence of labels $\delta \in \Delta^*$; 2) if the node reached via the sequence δ is the same as that reached by sequence δ', then do α (where α can be an operation which like 1) adds a node, reads or writes a symbol from S's input or output tape, halts, etc.). Since the first action can be iterated arbitrarily, it follows not only that the class of states St_S must be taken to be infinite, but also that over the course of the execution of S, an application of the second form of instruction many require surveying arbitrarily many paths originating at a in order to determine what action is to be applied to the current state. This means that while $cen_{\mathcal{S}}$ can be defined so that it will never contain more than finitely many nodes, it may get arbitrarily large. Under a simplistic physical interpretation, this means that unboundedly many distinct physical configurations must be recognized in a single step in any physical instantiation $\hat{S}$ of certain storage modification machines S. And since prima facie it seems implausible that any physical system with this property could exist, it is unclear whether the model $\mathcal{S}$ should be taken to satisfy the spirit of the boundedness condition. Since $\mathcal{S}$ also allows for operations which move the current center to an any node to which it is currently connected and to arbitrarily reassign the destination of a label sequence, similar problem arise with respect to providing physically plausible definitions of $Near^a_{\mathcal{S}}$ and $Near^c_{\mathcal{S}}$.

It is, of course, still possible to define $cen_{\mathcal{S}}$, $Near^a_{\mathcal{S}}$ and $Near^c_{\mathcal{S}}$ so that $\mathcal{S}$ formally satisfies the boundedness and locality conditions. In so doing, however, we are all but abandoning hope that the satisfaction of these conditions serves as a guarantee that they may serve as sufficient conditions to ensure the physical representability of a class of machines. For as we will see below, models like the RAM machine whose states and transitions do not admit to any plausible spatial interpretation may also be defined so that they formally satisfy these conditions. This suggests that with sufficient ingenuity, it may be possible to formulate a definition of an arbitrary model of computation $\mathcal{M}$ so that it is satisfies these conditions. And as such, it seems safest to conclude without additional amendment, the boundedness

and locality conditions are best seen as sharpening certain affinities between historically significant models without truly analyzing what they have in common.

This leaves open the possibility that there is no means of making a precise distinction between models like $\mathfrak{T}$, $\mathfrak{T}^2$ and $\mathfrak{S}$ which are traditionally classified as transition based models from the RAM and flowchart machines I will discuss in the next section. Nonetheless the affinities between models of the former class are sufficiently strong that I believe it is also possible to argue that no model of this sort can serve as a plausible candidate for the class $\mathfrak{Q}$ which the algorithmic realist wants to establish as containing "direct" representatives of as many algorithms in $\mathfrak{Q}$ as possible. The groundwork for this observation was already laid in Chapter 3. There we saw anecdotal evidence to the effect that no single-head, single-tape Turing machine could directly simulate the operation of an informal algorithm like PAL1 in a truly step-by-step sense. The essential problem was that while PAL1 is capable of accessing the elements of its input string $a_1 \ldots a_n$ by index i, allowing it to directly compare the symbols a_i and $a_{n-(i-1)}$ in a single step. However as we saw, it appears that any palindrome deciding Turing machines like S_1 or S_5 must perform this task indirectly over a sequence of operation of length $O(n)$ whereby a_i is stored in the machines states or transported to the right end of the tape for comparison. These informal observations are confirmed by the theorem mentioned above that no single tape, single head machine can decide the language L_{pal} in less than $\Omega(n^2)$ steps. Although I have yet to propose on the realist's behalf a formalization of what it means for an implementation to directly represent the operation of an algorithm, it seems that this result is sufficiently strong to rule out the possibility that $\mathfrak{T}$ contains a direct representative of PAL1 on any plausible analysis.

But in moving from the model $\mathfrak{T}$ to the model $\mathfrak{T}^2$ something substantial is gained. For as mentioned above it is easy to construct a two-head, single-tape Turing machine which decides L_{pal} in time $O(n)$. And with a little more work, it is also possible to construct specific two-head machines S_1^2 and U_1^2 which directly reflect the operation of PAL1 and PAL2 in the sense that a single stage in the operation of these procedures corresponds to a *constant* length sequence of steps for these machines (as compared to $O(n)$, as was the case for the simulations we considered in section 3).[15] This might be taken to suggest that

[15]With more work yet, these observations can be carried over to construct multi-tape Turing machines

$\mathcal{T}^2$ is already a plausible candidate for $\mathcal{Q}$.

In order to see that this is not the case directly would require going through a detailed case study of the sort conducted in Chapter 3 whereby we would attempt to show that there existed informally specified algorithms which operated over strings which could not be directly simulated by any machine in $\mathcal{T}^2$. But in so doing, we would again run into the problem of deciding when a given implementation is sufficiently reflective of the step by step behavior of an informal algorithm to be counted as a direct representation of that algorithm. In particular, even if we are able to exclude simulations in which $O(n)$ time algorithms are implemented by $O(n^2)$ machines, the examples considered in Chapter 3 suggest that failures of direct step-by-step simulatability can arise from other sources as well.

A simple example of this already occurs for the problem of deciding the language $L_{trip} = \{uuu \ : \ u \in \{0,1\}^*\}$. There is clearly an exact time $2\lfloor n/3 \rfloor$ informal algorithm for deciding membership in L_{trip}: on input $a_1 \ldots a_n$, successively compare a_1,$a_{\lfloor n/3 \rfloor+1}$ and $a_{2\lfloor n/3 \rfloor+1}$, a_2,$a_{\lfloor n/3 \rfloor+2}$ and $a_{2\lfloor n/3 \rfloor+2}$, ..., $a_{\lfloor n/3 \rfloor-1}$,$a_{2\lfloor n/3 \rfloor-1}$ and a_{n-1}, ... It is also clear that L_{trip} is decidable in asymptotic time $O(n)$ by a two-head, single-tape machine T^2. For instance T^2 could operate in two phases as follows: 1) during the first phase, T^2 compares $a_1 \ldots a_{\lfloor n/3 \rfloor}$ with $a_{\lfloor n/3 \rfloor+1} \ldots a_{2\lfloor n/3 \rfloor}$ using its two heads; and 2) during the second phase it compares $a_1 \ldots a_{\lfloor n/3 \rfloor}$ with $a_{2\lfloor n/3 \rfloor+1} \ldots a_n$. T^2 can make these comparisons directly in a sense which would be impossible for a one head machine. For note that by positioning its heads so that it is simultaneously scanning the pairs of symbols $\langle a_1, a_{\lfloor n/3 \rfloor+1} \rangle \ldots \langle a_{\lfloor n/3 \rfloor-1}, a_{\lfloor 2/3 \rfloor-1} \rangle$ and $\langle a_1, a_{2\lfloor n/3 \rfloor+1} \rangle, \ldots, \langle a_{\lfloor n/3 \rfloor-1,a_n} \rangle$, it can decide in one step if they are equal. However, it follows by a crossing sequence argument that any k-head, single-tape machine which decides L_{trip} must move its heads a distance $O(n)$ in order to make any of these comparisons.

Nonetheless it is also apparent that there is no k-head, single-tape machine which whose exact running time can come within a scalar multiple of the exact running time of the informal algorithm described above. For note that while a machine like T^2 can directly compare the symbols in the first and second and the first and third blocks of symbols, it

---

with one head per tape which also decide L_{pal} in linear time. This was first shown by Slisenko [133] and later improved by Leong and Seiferas [73].

must also expend at least $O(n)$ steps moving its second head between these blocks, resetting the first head between the second phases and doing bookkeeping to keep track of how the tape should be divided into thirds. The question of whether T^2 or any other two head Turing machine directly implements this algorithm thus inevitably foils our standards of directness. And for reasons with which we are now well acquainted, it is too much to ask that any implementation M of an algorithm A be such that the transitions which comprise M's executions are in one-to-one correspondence with those that comprise executions of A. But at the same time, allowing arbitrarily long (or even linearly bounded) sequences of transitions of M to correspond to single transitions of A is likely to require a liberal standard of simulatability which in turn is likely to allow for unintended simulations of sort considered in Chapter 3. And thus despite the fact that taking $\mathfrak{Q}$ to be $\mathfrak{T}^2$ instead of $\mathfrak{T}$ allows us to construct what appear to be more direct implementations of the particular algorithms Pal1 and Pal2, it is unclear to what extent this advantage generalizes to other algorithms.

This situation is typical of a general concern which arises when we try to carry out the abstractionist strategy by taking $\mathfrak{Q}$ to be any transition-based model $\mathcal{R}$. I have attempted to characterize such a model on the basis of its satisfaction of boundedness and locality conditions. On their intended interpretation, these conditions require that machines $Q \in \mathfrak{Q}$ have a finite locus around which computational activity takes places and which may only be moved bounded distance over the course of a single transition. We have seen that a definition of such a locus can be supplied for a model $\mathfrak{T}^2$, albeit with a loss of intuitive plausibility with respect to the definition which can be given for $\mathfrak{T}$. But at the same time, it is not at all clear that any such bound or even a general notion of a computational locus can be applied to informal specifications of algorithms such as Pal1 and Pal2. And we will see below, this is also true in a stronger sense for many recursive algorithms which also lack not only a clearly defined analog to a "program counter" (which keeps track of the step which they are currently executing) but also a clearly delineated notion of computational state.

In an effort to locate a plausible choice for $\mathfrak{Q}$ on behalf of the algorithmic realist, it thus seems that we must look beyond the class of models which are traditionally classified as

being based on a notion of state transition. But given that the conditions of discreteness, determinacy, boundedness and locality by which we have sought to characterize such models were originally formulated in an attempt to state minimal conditions of a formal model which allow its members to be physically implemented, going beyond the transition-based paradigm also means abandoning the physical interpretability as the potential basis of analysis of what it means for a model of computation to represent processes which occur in time. But as we are about to see, there are at least two other sets of metaphors on which it is also possible to base such an analysis.

4.4 Register Based Models

In the previous section, I attempted to provide a uniform characterization of one historically significant class of models on the basis of the possibility that their mode of operation might be assigned a physical interpretation. Given the foundational emphasis which has often been placed on transition-based models and their relationship to mechanism, it is a minor irony that the first general purpose computing devices which were physically implemented were not based on any of the models which are commonly thought of as belonging to this class. Rather, the design of both early computers such the well known the Zuse Z3, the Colossus and the ENIAC as well as the vast majority of contemporary digital computers are based on a mathematical model of computation which takes a general sort of variable assignment as its basic operation. Although we will see that it makes sense to speak of such assignments as occurring in time, it is much more difficult to assign physically inspired interpretations to these models than it is to typical transition-based models, at least when they are defined in the full generality with which they are studied in computer science.

The new of set of temporal metaphors around the class of models I wish to discuss in this section is based around the notion of what is conventionally known as a *storage register*. The fundamental component common to models in the classes of machines which I will refer to as *register-based* models of computation is that of an abstract location in which a mathematical value may be stored. In this basic sense, storage registers thus serve a purpose analogous to that of individual tape squares of a Turing machine in that they correspond to a sort of notional container which may be initially empty (i.e. not

contain a value) and then repeatedly refilled with different values. There are, however, three fundamental respects in which storage registers are more general than tape cells: 1) they may be used to contain not only individual symbols, but mathematical values drawn from an arbitrary domain D (e.g. natural numbers, rational numbers, strings, etc.); 2) no *a priori* upper bound is generally imposed on the size of a value which can be stored in a register; and 3) although it is typical to refer to (or *address*) registers as if they were elements of a linearly ordered array $r[0], r[1], r[2], \ldots$, no spatial significance is assigned to proximity of an index of a register to that of another.

The action of storing a value $a \in D$ in a register $r[i]$ is known as *assignment*. Another common feature of register-based models is that their mode of operation is generally specified in terms of sequences of assignments which can be expressed as statements of the form "Store a in $r[i]$." Following a common convention, I will use the notation $r[i] \leftarrow a$ to abbreviate such an instruction. The intended interpretation of this statement is an imperative which has as its success criterion (or as it usually called in the literature of programming languages, a *post condition*) that the register $r[i]$ contains the a.

Of course treating the assignment of a value to a register as the result of satisfying a certain form of imperative statement begs the question as to how this assignment is carried out. Although this point is familiar from Chapter 2.2, it is a particularly salient question with respect to the register-based paradigm since unlike transition-based models, instances of register based models are typically specified linguistically as programs over a variety of simple programming languages. Such programs can be taken to have the structure of a sequence of statements of the form $P = l_1 : p_1; \ldots; l_n : p_n$ where the p_i are either assignment statements of the form $r[i] \leftarrow t$ (for t a possibly complex term) other flow control constructs to be considered below. The labels $l_1, \ldots, l_n$ are line numbers which are also required for flow control.

In the paradigmatic case where the instruction p_i in question has the form $r[j] \leftarrow a$, carrying out p_i amounts to storing the value of a in the register indexed by j, potentially replacing a value which is already stored there. Saying what this amounts to in a more concrete sense (i.e. saying what it is about registers which allows them to contain values or how a register is "looked up" according to its index so that the value it contains may

be retrieved or modified) are details which are not specified by the formal development of register-based machines themselves.[16] However if we assume that a basic account of what it is to perform an assignment has been carried out, than a basic temporal interpretation of executing a register-based program P can be provided in terms of its structure. For if to execute a register-based program P is to carry out its constituent instructions *in order*, then we may naturally speak of one instruction being as executed *before* another. Such talk may clearly be extended so as to understand entire executions of the sorts of machines we will consider in terms of linearly ordered sequences of instructions which are carried out over the course of their operation.

With this general understanding of register based models and the way in which they may be treated as legitimate models of computation in the sense of Section 2 in hand, we may turn to a consideration of particular register based models and their suitability to the algorithmic realist's needs in choosing a model $\mathfrak{Q}$. The first thing to note in this regard is that while the Turing machine model $\mathfrak{T}$ is often treated as a sort of canonical instance of the transition based paradigm, for the register-based approach, this role is filled by a family of register-based models known collectively as *random access machines* which share certain central features but differ in substantially in certain detail. I will collectively refer to these models as comprising the RAM family. It is not on models of this sort which I think the algorithmic realist's hopes should ultimately be focused. But since instances of this family compromise the most familiar examples of register based paradigm, it will be useful to start out by reviewing the fundamental properties of RAM machines.[17]

A RAM machine R may be taken to be comprised of three components: 1) an infinite

[16]This sort of abstraction is codified in the sort of formal semantics which is typically supplied for register based programs. As we will see below, the interpretation of an assignment statement in such a semantics is very much like the definition of a variable assignment in first-order logic. For if we think of the value contained in a register as being given by a function v mapping from registers into natural numbers, then the effect of executing the assignment $r[j] \leftarrow a$ will be to update v to the assignment $v' = v[r(j)/a]$ – i.e. the assignment which assigns the same value of $r[i]$ as does v' for $i \neq j$, and assigns $r[j]$ the value a. This affinity between the effects of assignment statements interpreted relative to an imperative semantics and updating a variable assignment function forms the basis for first-order dynamic first-order logic in the style as developed by Harel [56].

[17]The survey which I have provided here is also essentially anachronistic in the sense that the model I will consider here is in fact a simplification of the earliest RAM models. Of these, the significant are of Shepherdson and Sturgis (1963) [128], Hartmanis (1971) [59] and Cook and Reckhow (1973) [23]. All of these models employed some for of indirect indexing. The model I will describe is based on the survey paper of van Emde Boas [149].

sequence of registers $vecr = r[0], r[1], \ldots$ indexed by natural numbers; 2) a program P given over a simple imperative programming language $\mathfrak{L}$; 3) a designated register acc known as the *accumulator*. To a first approximation, the first and second components may be thought of as having the same functions as a Turing machine's tape and finite control – i.e. the registers are a sort of abstract computational medium and, relative to a suitable mathematical interpretation, the program specifies how they are to be acted upon. The accumulator in turn serves as a sort of computational locus akin to the center of a storage modification machine on which mathematical operations are performed and which mediate access to the registers. The language $\mathfrak{L}$ is always assumed to contain four sorts of constructs: i) flow control instructions including instructions to halt and to transfer control to a program step with a certain label, either unconditionally or conditional on the outcome of a simple numerical test, usually applicable only to the accumulator register; ii) instructions for transferring data from the registers to the accumulator and the accumulator to registers; and iii) instructions for performing arithmetic operations on the accumulator.

The variety of different RAM models arise from different ways in which each of these dimensions of $\mathfrak{L}$ may be made precise. For instance, in regard to i), it is traditional to allow only the test $acc = 0?$ which tests if the current value stored in the accumulator is equal to 0. But we may additionally allow tests like $acc \geq 0?$ or $acc = r[i]?$ (where the latter test for equality between the current contents of the accumulator and the contents of register i). In regard to ii), it traditional to allow both direct and indirect addressing of registers with respect to both assignment from the accumulator to registers and retrieval from the registers to the accumulator. These operations are traditionally denoted by $acc \leftarrow r[i]$ (which assigns the accumulator the value currently stored in $r[i]$), $acc \leftarrow r[r[i]]$ (which assigns the accumulator the value which is currently stored in the register indexed by the current contents of $r[i]$) and the store operations $r[i] \leftarrow acc$ and $r[r[i]] \leftarrow acc$ with the corresponding interpretations. However, in simplified RAM models the latter indirect forms of these operations are omitted. Finally, in regard to iii), it is traditional to allow for both addition and subtraction to be applied on the accumulator and a value stored in an arbitrary register with the result being stored in the accumulator. These operations

are respectively expressed as $acc \leftarrow acc + r[i]$ and $acc \leftarrow acc - r[i]$. [18] However, weaker RAM models are also studied in which only the successor and predecessor operations may be applied to the accumulator as well as stronger ones in which division and subtraction may be applied as well as addition and subtraction.

Different combinations of the features i), ii) and iii) lead to a variety of different precise definitions of the language $\mathfrak{L}$ over which a RAM program may be stated. And these different languages lead in turn to different classes of programs which, when interpreted in a manner similar to that which will be discussed below, in turn lead to different models of computation in the RAM family. All combinations of features are known to lead to models which are Turing complete. However the complexity theoretic relationships between different the different models is much more complex.[19] Thus despite the fact that the model with the choices I have flagged as "transitional" (i.e. multiple arithmetic tests on the accumulator, indirect addressing allow, only additional and subtraction) is generally taken as the basic model in complexity theory, it is difficult to provide a completely general or systematic defense of this model over the others.

Luckily, however, the problem of choosing between different variants of the RAM family is not likely bear directly on how the realist is likely to go about choosing $\mathcal{M}$. To why see this is so, it is first useful to take stock how far the family of RAM models departs from the conditions on mechanism discussed in the previous section. As we have seen, it unclear how precisely these conditions can be no defined so as to ensure that they apply uniformly to all models which are traditionally classified as transition-based. Nonetheless, it is still evident that even the most restricted of the RAM models violates the spirit of both the boundedness and locality conditions in at least three different ways. For let $\mathcal{R}$ be the model with just the test $acc = 0?$, no indirect addressing and only predecessor and successor applicable to *acc*. We can represent the states of such a machine $R \in \mathcal{R}$ as an infinite tuple of the form $\langle \mathsf{k}, v(acc), v(r[0]), v(r[1]), \ldots \rangle$ where k represents the current program line

[18] paradox of assignment

[19] Results comparing the different RAM models are usually framed in terms of the existence of "efficient" simulations between different variants. See [149] and [118] for a summary of other known results as well as a discussion of the effects of defining the traditional structural complexity classes (i.e. P, NP, $PSPACE$, etc.) relative to these different models.

being executed $v(acc)$ represent the current accumulator value and $v(r[i])$ represents the current contents of register r. As we will see below, R's program P may be compositionally interpreted so as to determine a transition function $\Delta_R : St_R \rightarrow St_R$. On this basis it should be clear from the intended meanings of the instructions summarized above that the application of Δ_R to a state $\mu \in St_R$ will induce a change at most in k and in one of the $v(r[i])$s.

One initially think on this basis that such a change of state was suitably constrained so that definitions of the nearness and center for $\mathcal{R}$ could be given which were compatible with the intuitive motivations given in previous section and thereby show that the model $\mathcal{R}$ satisfies the bounded and locality conditions. But this turns out not to be the case for a several reasons. One of the most most fundamental of these is that as defined members of $\mathcal{R}$ operates directly on natural numbers. This is in contrast to transition based models, all the example of which we considered operate on a symbols over a finite alphabet. In order to perform one of the RAM operations denoted by the instructions $r[i] \leftarrow acc$, $acc \rightarrow r[i]$ or $acc \leftarrow acc + 1$, it follows that R must *over the course of a single unmediated step* act differentially depending on infinitely many possible values of $r[i]$ or acc. For instance, if the next instruction to executed in state μ is of the form $acc \leftarrow r[i]$, the operation which mediates between μ and $\mu' = \Delta_\mu$ (i.e. that of storing the value of acc in the register $r[i]$) will be different depending one of infinitely many potential values of $v(r[i])$. And from this it follows that R cannot possibly satisfy the boundedness condition which states that if R in state μ, the operation performed by Δ_R in transforming μ into $\mu' = \Delta_R(\mu)$ must be completely determined by finitely many "local" component of its state.[20]

It thus appears that all common variants of the RAM model lack the kind of quasi-physical motivation which can be provided for transition-based models. But in this case

[20]It somewhat less clear whether we ought to accept the basic RAM model $\mathcal{R}$ as satisfying the locality condition. For note that while a fixed R may operate on an arbitrary large number of during its computation, no only is this number bounded, but it can be determined in advance by simply examining the finite set $S \subset \mathbb{N}$ of indices which occur in its program. Thus although there is no natural way of speaking of the proximity of one register to another, it is possible to define the relation $Near^a_R$ so that all pairs of states which differ only on the values stored in registers with indices in S are all treated as being in proximity to one another. But of course since there are machines $R \in \mathcal{R}$ such that $|R| > n$ for any n, this definition may be at odds with the intent of the locality constraints to rule out "action at a distance." Note also that this technique will not work for the RAM variants allowing indirect indexing, for in this case a single machine R may index arbitrarily many different registers during the course of a single computation.

we have also seen that it is still possible to provide a principled basis for treating these formalisms as models of computation for which a temporal interpretation can be given. And thus while the register based paradigm clearly represents a step away from a physically grounded explanation of why certain mathematical formalisms can play the role of procedural implementations, it is exactly this sort of abstraction away from considerations rooted in the foundational analysis of effectiveness which allows this framework to encompass implementations which more directly reflect the properties of algorithms such as Pal1 and Pal2.

One way to argue for this claim is to show that there exist informal algorithms whose transition-based implementations must be "indirect" (in the sense of S_1 relative to Pal1), but which may be implemented as RAM machines in a manner which more directly reflects their informal specification. But in attempting to do this, we immediately run into the problem that the exemplars of transition based models which we have thus far considered operate on discrete combinatorial structures (i.e. strings or graphs) whereas all of the models of the RAM family operate of natural numbers. And thus in order to make direct comparisons of implementations between, say, the classes $\mathcal{T}$ and $\mathcal{R}$, some form of encoding of inputs and states would be required which either allowed the members of $\mathcal{T}$ to be interpreted as operating on natural numbers or the members of $\mathcal{R}$ to be interpreted as operating on strings.

Since such encodings present another dimension along which it is possible to construct "unintended" simulations of the sort considered in Section 3, it will be useful to consider a form of register-based model which may be interpreted as operating over types other than natural numbers. One model with this property is known as that of the *flowchart machines* which were first considered by Luckham et al. [78]. This model is slightly simpler than most forms of RAM machine as each flowchart machine possesses a finite number of registers which must be addressed directly and allows an n-ary mathematical operation to be applied directly to the values stored in n registers without the need to first move them to a designated accumulator register.

In more detail, flowchart machines are the mathematical objects which result from assigning an interpretation to a simple class of uninterpreted imperative programs known

as *program schemes* which are similar in form to RAM programs. Formally, a program scheme is defined as follows:

Definition 4.4.1. A program scheme P is determined relative to a first-order language $\mathcal{L}_P$ consisting of a finite class $\mathcal{P}_P = \{P_1, \ldots, P_{n_p}\}$ of predicate letters, a finite class $\mathcal{F} = \{f_1, \ldots, f_{n_f}\}$ of function symbols (both of arbitrary arities), and a finite class $\mathcal{V} = \{v_1, \ldots, v_{n_v}\}$ of variable symbols. P itself is then defined to be as length-$n \in \mathbb{N}$ sequence of statements labeled with natural numbers $\mathsf{k} \leq n$ of one of the following three forms:

i) k: $y \leftarrow f(x_1, \ldots, x_n)$ for $f \in \mathcal{F}$ and $y, x_1, \ldots, x_n \in \mathcal{V}$

ii) k: if $P(x_1, \ldots, x_n)$ then goto l_1 else goto l_2 for $P \in \mathcal{P}$, $x_1, \ldots, x_n \in \mathcal{V}$, $\mathsf{j}, \mathsf{k} < n$

iii) k: Halt

Program schemes thus employ the same basic imperative syntax as RAM programs. And since this syntax is very similar to that of informal pseudocode specifications, it is often easy to understand the intended procedural interpretation of a program informally. Consider, for instance, the following scheme:

P_{fib}

1: $x_2 \leftarrow 1$
2: $x_3 \leftarrow 1$
3: if $x_1 < 2$ then goto 8 else goto 3
4: $x_2 \leftarrow x_2 + x_3$
5: $x_3 \leftarrow x_2$
6: $x_1 \leftarrow x_1 - 1$
7: if $1 = 1$ then goto 2 else goto 2
8: Halt

(4.7)

Suppose we assign the normal meanings to the predicates $<$ and $=$ and the function symbol $+$ and treat 1 as a shorthand for the constant 1 function. If we additionally assume that this scheme receives its input in register x_1 and produces its output in register x_2, then under this interpretation, P_{fib} ought to determine the standard Fibonacci function $fib : \mathbb{N} \to \mathbb{N}$

Although it is generally possible to informally understand the interpretation of a program scheme in this manner, a linguistic entity of this sort should be taken to correspond

to an implementation in the technical sense until we also supply a precise semantics which show how it gives rise to a mathematical object with minimally possesses the properties discussed in section 4.1. This achieved for a scheme P by choosing an interpretation I of $\mathcal{L}_P$ which assigns a denotation to the sets of symbols $\mathcal{P}$ and $\mathcal{F}$ relative to a domain D in the standard sense of first-order logic – i.e. $P^I \subseteq D^n$ for all n-ary $P \in \mathcal{P}$ and $f^I \in D^{D^m}$ for all m-ary $f \in \mathcal{F}$. We interpret the variable x_i appearing in P via a vector of values $\vec{a} \in D^{v_n}$. Finally, we assume that if a scheme P contains n_v variable and we wish to interpret as inducing a function $f : D^n \to D$ for $n \leq n_v$, then the initial values of $x_1, \ldots, x_n$ are taken to correspond values $a_1, \ldots, a_n \in D$ which are treated as the arguments to f.

A flowchart machine may now be defined as a pair $\boldsymbol{P} = \langle P, I \rangle$. If for $\boldsymbol{P} \in \mathcal{P}$ we take $X_P = Y_P = \mathcal{D}$, all that remains to adapt this formalism to our original definition of model of computation is to show how a function $App : \mathcal{P} \times \mathcal{D}^n \to \mathcal{D}$ may be defined which gives the value which would be returned by a machine $\boldsymbol{P} \in \mathcal{P}$ when started in an initial state determined by a vector $\vec{a} \in \mathcal{D}^n$. This may be achieved by providing a simple operational semantics for program schemes. In order to do so, first define a *state* of a flowchart machine to be a sequence $\langle b_1, \ldots, b_{n_v}, \mathsf{k} \rangle$ such that b_j (for $j \leq v_n$) gives the current contents of register x_j and $\mathsf{k} \leq v_n$ gives the line number of the instruction currently being executed. Let St_P denote the set of such sequences. The *execution* of $\boldsymbol{P}$ determined by initial values $\vec{a} \equiv \langle a_1, \ldots, a_{v_n} \rangle$ is then defined to be a finite or infinite sequence of such states $\mu_0, \mu_1, \mu_2, \ldots$ which is determined in the following manner.

First, let $val(x_j, k)$ denote the value stored in register x_j at the kth state in such a sequence and $line(k)$ denote the line number associated with the kth state. The state μ_{k+1} is defined in terms of μ_k as follows:

(4.8) 1) If $k = 0$, then $\mu_k = \langle a_1, \ldots, a_n, 1 \rangle$ – i.e. μ_k is the state with register i assigned the initial value a_i and current location assigned as line 1.

2) If $k > 0$ and $line(k)$ of P is $x_j \to f(x_{q_1}, \ldots, x_{q_n})$, $val(x_{q_i}, k) = b_i$, $f^I(b_1, \ldots, b_n) = c$, then $\mu_{k+1} = \langle d_1, \ldots, d_{n_v}, line(k) + 1 \rangle$ where $d_i = val(x_{q_i}, k)$ if $i \neq j$ and $d_i = c$ otherwise.

3) If $k > 0$ and $line(k)$ of P is $x_j \to x_l$, $val(k, x_l) = c$ then

$\mu_k + 1 = \langle d_1, \ldots, d_{n_v}, line(k) + 1 \rangle$ where $d_i = val(x_i, k)$ if $i \neq k$ and $d_i = c$ otherwise.

4) If $k > 0$ and $line(k)$ of P is $\mathsf{If}\ P(x_1, \ldots, x_n)\ \mathsf{then\ goto\ l}_1\ \mathsf{else\ goto\ l}_2$, $val(x_i, k) = b_i$ and $P^I(b_i, \ldots, b_n)$ is true, then $\mu_{k+1} = \langle val(x_1, k), \ldots, val(x_{v_n}, k), \mathsf{l}_1 \rangle$. Otherwise $\mu_{k+1} = \langle val(x_1, k), \ldots, val(x_{v_n}, k), \mathsf{l}_2 \rangle$.

5) If $k > 0$ and $line(k)$ is Halt, then $\mu_{k+1} = \langle val(x_1, k), \ldots, val(x_n, k), \mathsf{h} \rangle$ where h is a special marker denoting a halting state.

Suppose that $\boldsymbol{P}$ is a flowchart machine and $n < n_v$. In order to view $\boldsymbol{P}$ as inducing a function of type $D^n \to D$, we may finally define the function $App(\boldsymbol{P}, x_1, \ldots, x_n)$ as follows. For input values $a_1, \ldots, a_n \in D$, and arbitrary $d_1, \ldots, d_{v_n - n} \in D$ we define $App(\boldsymbol{P}, b_1, \ldots, b_n)$ to be $val(k, n + 1)$ where k is the index of the least component of the execution of $\boldsymbol{P}$ starting with initial configuration $\langle b_1, \ldots, b_n, d_1, \ldots, d_{v_n - n}, 1 \rangle$ such that $line(k) = \mathsf{h}$, and is undefined if there no such k exists.[21]

We may now return to the question of whether a register-based model $\mathcal{P}$ represents an improvement over transition-based models with respect to the realist's central desire to locate more a flexible model of computation for representing the algorithms in $\mathcal{A}$. The first thing to note in this regard is that since it was essentially locality-based considerations which prevented direct Turing machine implementations of PAL1 and PAL2, it should not be surprising that there are much more natural representations of these algorithms as flowchart machines. In order to see this, we must be somewhat careful about how we formalize the string-based operations which appear in the informal specifications of these procedures. For note that there are two ways in which we can construct a flowchart machine which takes a finite string $w \equiv a_1 \ldots a_n$ over $\{0, 1\}^*$ as input: 1) we may initial store each of the symbols a_i in a separate register x_i and then proceed to compare the values stored in

[21] The choice of the name *flowchart machine* to denote members of $\boldsymbol{P}$ may be justified on the basis of this definition. For note that we may think of the interpretation of each program scheme P as giving rise to a directed graph whose nodes correspond to individual instructions and whose edges correspond to how the control is transfered from instruction to instruction during the course of its execution as follows: 1) if a node corresponding to a statement of the form k: $y \leftarrow f(x_1, \ldots, x_n)$ is connected to the node corresponding to the instruction with label $\mathsf{k{+}1}$; 2) a node corresponding to a statement of the form *k: if* $P(x_1, \ldots, x_n)$ *then goto* l_1 *else goto* l_2 will be connected to the nodes with labels l_1 and l_2; and 3) a labeled node corresponding to a statement of the form k: Halt will have no outgoing edge.

different registers; 2) we may initially store w in a single register and then using appropriate string-based operations defined on single registers to access and manipulate its constituent symbols.

A number of both general and technical factors militate in favor of choosing option 2) over option 1). First, note that since any given flowchart machine P will have only a fixed number of registers n_v, no individual flowchart machine will contain sufficiently many registers to allow it to store strings of arbitrary length. It thus follows that no flowchart machine could compute the characteristic function of L_{pal} unless it employed some variant of 2). While employing this strategy may seem like unnecessary detour since other members of other register based models (such as RAM machines) already contain enough registers to store strings of arbitrary length in a bitwise fashion, we will shortly see that adopting 2) also enforces a useful segregation between the data structures which are intrinsically part of a given model from those are more properly associated with the particular class of mathematical objects on which we employ it to operate.

Having made this decision, it is now straightforward to construct register machines $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ which hae the appearance of implementing PAL1 and PAL2 more directly than any of the Turing machines considered in Section 3. The first step in doing so is to adopt some means of distinguishing between registers which contain string values, those which contain symbol values (i.e. 0 and 1 in the case under consideration) and those which contain integer values under the intended interpretations of $\mathfrak{L}_{P_1}$ and $\mathfrak{L}_{P_2}$. This can be carried out in any of a number of ways, but the most straightforward is to simply name registers using separate sets of variables $w_1, w_2, \ldots$, $a_1, a_2, \ldots$ and $x_1, x_2, \ldots$ and adopt the convention that the first may only be used to store strings, the second to store symbol values and third to store numerical values. If we now examine our original pseudocode specifications of PAL1 and PAL2 in light of these conventions, it should be clear that if we wish to naturally express these procedure as program schemes, the language $\mathfrak{L}_{pal}$ over which $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ are stated should contain the function symbols $\mathcal{F}_{pal} = \{length(w), getbit(w,i), setbit(i,a), x + y, x - y, \lceil x/2 \rceil\}$ and the predicate symbols $\mathcal{P}_{pal} = \{a_i = a_j, x \leq y\}$.

We now can now proceed to render PAL1 and PAL2 as program schemas. One set of possible outcomes is indicated in Figure 4.1. Now let I be an interpretation with domain

P_{pal1}
1: $x_1 \leftarrow$ no
2: $x_2 \leftarrow 1$
3: $x_3 \leftarrow length(w_1)$
4: $x_4 \leftarrow \lceil x_3/2 \rceil$
5: If $x_2 \leq x_4$ then goto I_6 else goto I_{12}
6: $a_1 \leftarrow getbit(w_1, x_2)$
7: $x_5 \leftarrow x_2 - 1$
8: $x_6 \leftarrow x_3 - x_5$
9: $a_2 \leftarrow getbit(w, x_6)$
10: If $a_1 = a_2$ then goto I_{10} else goto I_{13}
11: $x_2 \leftarrow x_2 + 1$
12: If $1 = 1$ then goto I_5 else goto I_5
13: $x_1 \leftarrow$ yes
14: Halt

P_{pal2}
1: $x_1 \leftarrow$ no
2: $x_2 \leftarrow 1$
3: $x_3 \leftarrow length(w_1)$
4: If $x_2 \leq w_3$ then goto I_5 else goto I_9
5: $a_1 = getbit(w_1, x_2)$
6: $x_4 \leftarrow x_2 - 1$
7: $x_5 \leftarrow x_3 - x_4$
8: $w_2 = setbit(x_5, a_1)$
9: $x_2 \leftarrow x_2 + 1$
10: If $1 = 1$ then goto I_4 else goto I_4
11: $x_2 \leftarrow 1$
12: If $x_2 \leq x_3$ then goto I_{11} else goto I_{16}
13: $a_1 \leftarrow getbit(w_1, x_2)$
14: $a_2 \leftarrow getbit(w_2, x_2)$
15: If $a_1 = a_2$ then goto I_{14} else goto I_{17}
16: $x_2 \leftarrow x_2 + 1$
17: If $1 = 1$ then goto I_{10} else goto I_{10}
18: $x_1 \leftarrow$ yes
19: Halt

Figure 4.1: PAL1 and PAL2 as program schemes.

$\{0,1\}^* \cup \{0,1\} \cup \mathbb{N}$ of $\mathfrak{L}_{pal}$ which assigns the intended meaning to each of its symbols. In other words I assigns the normal arithmetic interpretations to $x + y$, $x - y$, $\lceil x/2 \rceil$ and is such $length(w)^I = |w|$, $getbit(a_1 \dots a_i \dots a_n, i)^I = a_i$, $setbit(i, b) = a_1 \dots a_{i-1} b a_{i+1} \dots a_n$ (where $a_1 \dots a_n$ is the string stored in the register to be updated) and $a_1 =^I a_2$ if and only if the symbols associated with a_1 are a_2 are identical. It should then be clear that under the under the definition of *App* given above, the flowchart machines $\boldsymbol{P}_1 = \langle P_{pal1}, I \rangle$ and $\boldsymbol{P}_2 = \langle P_{pal2}, I \rangle$ determine the characteristic function of L_{pal} under the input/output mappings which assigns input w to register w_1 and reads output from register x_1 (interpreting yes as the name for the constant 1 function and no as a name for the constant 0 function).

It should also be clear that individual executions of these machines as formalized by (4.8) correlate closely with those of PAL1 and PAL2 on the same inputs as described in Section 3 in terms of informally delimited stages. Of course, at this point we are also well acquainted with the exigencies involved with making such claims precise. For note that as was the

case in Section 3, the necessity of regimenting a method originally described as pseudocode in the exact format of a program scheme requires that various instructions which were originally as single pseudocode statements must be expressed as multiple program scheme instructions. For instance the iterative Step 2 of PAL1 is expressed across lines 5-10 of P_{pal1}. In this case, the necessity of expressing a pseudocode statement indirectly arises for two reasons: 1) there is no native for-do loop construction in the basic language of program schemes and thus such loops must be expressed indirectly by using a counter register (in this case x_3) together with a conditional goto state (in this case, line 4); 2) since the program scheme language also does not support the use of complex functional terms, the index value which we would informally express as $n - (i - 1)$ has be built by storing the values of its subexpressions in individual registers (in this case, over the course of lines 3, 6 and 7.

One might initially think that the necessity of resorting to such computational detours in expressing PAL1 and PAL2 suggest that little has been gained in moving from the model $\mathcal{T}$ to the model $\mathcal{P}$. But upon examining the executions of P_{pal1} and P_{pal2} in more detail it is also evident that the necessity of expressing certain the pseudocode step just mentioned indirectly leads only to the necessity of associating *constant length* sequences of transition of these machine with single stages in the execution of PAL1 and PAL2. In other words, even though the loop expressed as line 2 of PAL2 must be expressed over lines 5-10 of P_{pal1}, each stage of an execution of the former will correspond to exact five transitions in an execution of the latter. Thus although there will not be a one to one correspondence between the informally delimited stages of executions of PAL1 and PAL2 and transitions in the executions of P_{pal1} and P_{pal2}, there will be a fixed constant k, independent of $|w|$, such that one stage can correspond to no more than k transitions. And while there similarly cannot be a one to one correspondence between stages and Turing transitions, we have seen above that in this case, certain stages in the execution of PAL1 and PAL2 must be correlated with sequences of transition of length $O(|w|)$. On this basis of this observation, the realist might attempt to claim that P_{pal1} and P_{pal2} come closer to directly implementing PAL1 and PAL2 than do any of the Turing machines considered in section 3.

But for equally familiar reasons, there is no obvious basis on which P_{pal1} and P_{pal2} can be claimed as the *unique* or flowchart representatives of PAL1 and PAL2. For note that

a number of minor modifications can be made to either of these schemas which produce machines with equal title to being counted as implementations of PAL1 and PAL2. For instance, the ordering of sequential assignment instructions with disjoint sets of variables such as lines 1-4 of P_{pal1} of lines 13-14 of P_{pal2} can be permuted. And similarly, the guards used on the conditional used to implement the for-do of PAL1 and PAL2 can be modified in a variety of different ways which will lead to schemes with related but non-identical executions.

These considerations suggest that if the realist wishes to employ $\mathcal{P}$ as the class of implementations $\mathcal{M}$ with respect to which he wishes to pursue the abstractionist programme, he still have some work to do in proposing and justify a definition of $\underline{\leftrightarrow}$. In doing so, he will have to confront some of the same issues which arose in section 3 – e.g. the need to formulate a representation requirement over structurally distinct classes of states, the possibility of one step in the operation of one flowchart machine corresponds to more than one steps in the operation of another machine and vice versa, etc..

To see in a precise sense that progress has indeed been made would require us proceed as we did in section 3 – i.e. by examining both the executions of $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ and those of other flowchart machines which can plausibly be taken to implement PAL1 and PAL2. But based on the structure of the argument adduced against the abstractionist programme in section 3, we can also see in advance that the outcome of such a study is likely to turn on whether there exists machines $\boldsymbol{Q}_{pal1}$ and $\boldsymbol{Q}_{pal2}$ which stand in analogous relations to $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ as the Turing machines S_5 and U_2 do to S_1 and U_2. Such a pair of machines would respectively implement PAL1 and PAL2 with the same degree of intuitive plausibility as do $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$. But at the same time, they would have to operate so differently from $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ in a local step-by-step sense as to confound any proposed definition of $\underline{\leftrightarrow}$ which did not also assimilate $\boldsymbol{P}_{pal1}$ and $\boldsymbol{P}_{pal2}$ themselves.

While it initially seem implausible that the existence of such machines can be ruled out a priori, in choosing $\mathcal{M}$ to be $\mathcal{P}$ he now at least has a plausible means of reply. For note that the flowchart model makes available two constructs which were not present in $\mathcal{T}$: 1) a means of directly accessing and manipulating the elements of strings by index; 2) a flow control mechanism which allows for the direct implementation of iteration over indices. As

we have seen, the presence of the first feature allows for comparison of the bits a_i and $a_{n-(i-1)}$ to be performed without the use of an auxiliary method for transporting the value of a symbol from the left end of the string to the right end. And the presence of the second feature allows for expression of loops whose guard conditions resemble those of Pal1 and Pal2 in the sense that a counter variable (e.g. x_2 in the case of P_{pal1} and P_{pal2}) may be directly compared to the length of $a_1 \dots a_n$ instead of relying on an indirect method such as the overwriting symbols at the end of the tape in order to keep track of progress.

The realist might hope to use these observations as the basis for formulating a precise definition of what it means for a flowchart to "directly" implement the loops expressed in Pal1 and Pal2. For instance we have seen that there will thus only be a need to recognize constant (i.e. $O(1)$) length sequences of their transitions as corresponding to single stages in the execution of Pal1 and Pal2 as opposed to linear (i.e $O(n)$) length sequences in the case of $\mathcal{T}$. These observations in turn suggest that a definition of $\underline{\leftrightarrow}$ may be formulated which restricts the quantification over decomposition functions (c.f. Definition ???) much more tightly than is possible for Turing machines. And on the basis of such an observation, it is at least plausible to hope that a definition of $\underline{\leftrightarrow}$ can be formulated over $\mathcal{P}$ which avoids many of the problems of grain encountered in Section 3.

There are, of course, sufficiently many what-ifs built into the scenario just described to shift the rhetorical onus back on to the algorithmic realist, who has the responsibility of producing an explicit definition of $\underline{\leftrightarrow}$. But at this point it should also be clear that the added sophistication of $\mathcal{P}$ over $\mathcal{T}$ can only serve as an advantage in this regard. In particular, it appears that in moving from one to the other, the classes of intuitively acceptable implementations of algorithms like Pal1 and Pal2 can only become more computationally homogeneous and thus (as the realist may hope) easier to define implicitly via an appropriate definition of bisimulation.[22]

[22]But these appearances may be deceiving. One genuine area for concern arises from the presence of the arithmetic symbols $+$, $-$ and $\lceil \cdot/2 \rceil$ in $\mathfrak{L}_{pal}$ which we are obliged to assume are interpreted according to their standard means. For note that once such operations are admitted to the basic set of operations in terms of which a register machine is defined, the model of computation which results from considering only the interpretation of these symbols will not only be Turing complete but also will require only a small number of auxiliary registers (in fact 2 – c.f. [87]) to compute any recursive function. This leads to the possibility that seemingly-innocuous variables can be used to formulate guard conditions of arbitrary computational complexity. As such, there will be instances of programs schemas which appear structurally similar to

Even if an adequate definition of bisimulation cannot be given for $\mathcal{P}$ as things stand, the forgoing observations also suggest an additional strategy which is open to the realist. For just as the putative gain in directness between, say, S_1 and P_{pal1} qua implementation of PAL1 can be traced to the presence in the flowchart model of mathematical constructions which already appear in the pseudocode specification of this procedure, it also seems reasonable to suspect that adjoining additional operations and procedural constructs present in the informal specifications of the algorithms in $\mathcal{A}$ can only lead to programs schemes capable of representing these algorithms more directly. For instance, if we modified $\mathcal{L}_{pal}$ so that it contained complex functional expressions (i.e. expressions of the form $f(g_1, \ldots, g_n)$) which were then interpreted in the standard first-order manner, we could then formulate program schemes which eliminated the minor detours which required the use of extra registers in P_{pal1} and P_{pal2} to build up the desired index values. There are additionally many other readily foreseeable generalizations in which the model $\mathcal{P}$ could be generalized – e.g. by introducing additional looping constructs, reintroducing indirect indexing of registers, allowing for subroutines, etc.. And on this basis, the realist might propose that if we simply keep expanding the set of available operations and constructs available as primitive components then we will ultimately converge to a model which contains machines which so accurately reflect the intended interpretations of our informal specifications of algorithms that the problem of defining $\underline{\leftrightarrow}$ will ultimate become manageable, if not entirely trivial.

This strategy raises several questions about the general problems involved with providing a definition of implementation which we are now in a position to begin to discuss in detail. One of these concerns the relationship between the class $\mathcal{P}$ of flowchart machines and our original definition of transition systems. We observed above that register based such as those of the RAM family can not reasonably be taken to satisfy the boundedness condition on mechanism and as such are not naturally classified as transition based model. It should be clear on the basis of (4.8) that the same if true of the flowchart machine model. For although the which can occur to a state over the course of a single transition are still

---

P_{pal1} and P_{pal2} but whose equivalence even with respect to the function they compute will be formally undecidable. This suggests that unless the realist can propose a general means of ruling out pathological machines which disguise complex computations inside simpler ones, an extensionally adequate definition of bisimulation for flowchart machines may actually be more complex than one for Turing machines.

"local" in the sense of effective only two of its components (i.e. the value contained in a single register and the current line number), the ability to compute on arbitrary data types as determined by the symbols in $\mathcal{L}_P$ and the interpretation I means that there will be members of $\mathcal{P}$ which do not satisfy the intent of the boundedness condition for essentially the same reason as the simple RAM model $\mathcal{R}$ discussed above. It is equally clear, however, that the members of $\mathcal{P}$ may be uniformly represented as transition systems in the sense of Definition 3.4 of Chapter 3. In particular given a flowchart machine $\boldsymbol{P} = \langle P, I\rangle$ it is straightforward to define a transition system M_P with the same states and transition function based on (4.8).[23]

Thus despite the greater flexibility which a register based model like $\mathcal{P}$ offers over a transition base model like $\mathcal{T}$, both models are subsumed under the general definition of transition system. And it should be equally clear that many of the ways alluded to above in which we might think to produce a more general register will still result in models which can also be straightforwardly presented as transition systems. The question thus arises whether we are indeed justified in treating this definition as a general analysis of the notion of implementation which the realist hopes to construct so as to define the class $\mathcal{M}$ in a principled manner. I will ultimately argue that this question should be answered positively, but that when the relationship between extent models of computation and transition systems is understood in full detail that this conclusion will offer support to the realists hope that by taking a very inclusive class of implementation as $\mathcal{M}$ he may substantially reduce the complexity of defining $\underline{\leftrightarrow}$.

To see why this is so requires that we examine the final class of the three traditionally recognized classes of models of computation – i.e. those based on recursion. But in order to set the stage for this, it will useful to first introduce a distinction which distinguishes between the class $\mathcal{P}$ and all of the models we have considered previously. Note in particular that $\mathcal{P}$ differs from the models in the RAM family (and similarly from transition based

[23]In fact the definition of a flowchart machine is already so to similar to that of a transition system so as to make this definition trivial. To define $M_P = \langle X_P, Y_P, St_P, \Delta_P, H_P, in_P, out_P\rangle$ we may take $X_P = D^n$ (for $n < n_v$), $Y_P = D$, St to be as defined above, Δ_P to defined so that $\Delta_P(\mu) = \mu'$ just in case $\mu = \mu_i$ and $\mu' = \mu_{i+1}$ in some execution of $\boldsymbol{P}$, H_P to be the set of states of the form $\langle a_1, \ldots, a_n, h\rangle$ for $a_i \in D$, $in_P(a_1, \ldots, a_n) = \langle a_1, \ldots, a_n, d_{n+1}, \ldots, d_{n_v}\rangle$ (for $d_{n+1}, \ldots, d_{n_v}$ some arbitrarily chose values of D) and $out_P(a_1, \ldots, a_{n_v}) = a_{n+1}$.

models like $\mathcal{T}$ or $\mathcal{S}$) in that individual flowchart machines may be defined so that the basic operations and predicates they employ on registers correspond to arbitrary mathematical functions and predicates. For instance, the machines $\boldsymbol{P}_{pa1}$ and $\boldsymbol{P}_{pal2}$ employ operations and predicates which operate on natural numbers, strings, and symbols whereas this would not be possible for either a Turing machine or RAM machine. Moreover, other flowchart machines could be defined which operate directly on matrices, graphs, groups, etc.

I will call a model of computation whose members may include objects and operations drawn from a general class which is independent of the definition of the model itself an *open* model. Since the algorithms we are taking to constitute the class $\mathcal{A}$ are themselves usually specified in terms of arbitrary (effective) operations and predicates, openness seems like desirable property of any model of computation which, per the realist's desire, contains members which serve as suitably direct members of $\mathcal{A}$. However, openness in this sense must be distinguished from another important dimension which may add to (or constrain) the ease with which a given model may be adapted so as to directly express an algorithm. This other access concerns the manner in which we allow a model to access, store, and manipulate the mathematical objects on which it operates. For instance, we have a seen that a model like $\mathcal{T}$ is very conservative in this regard in that in order to access a symbol value, its head must literally be moved to the square that contains it. Register-based models like $\mathcal{P}$ are more flexible in this regard since they allow value to be accessed by index. But as we are about to see, recursion-based models can be thought of as more flexible yet since there is a sense in which they can access and manipulate multiple components of their state at the same time in a manner which properly extends the abilities of flowchart machines. And since we will also see that there is an in-principle reason why certain variants of these models cannot be represented as transition systems, there is also a prima facie reason to doubt that such systems provide a sufficiently broad framework for the realist's purposes.

4.5 Recursion-based models

In traditional taxonomies of models such as those given in [113] or [63], it is customary to contrast both transition- and register-based models with models based on various forms of recursion. This distinction turns on a fundamental difference in the sense that informally

described computational processes may be viewed as occurring in time. As we have seen, the notion of execution which is typically defined for both transition- and register-based models $\mathfrak{I}$ is grounded in the mathematical notion of *iteration*. Such models may typically be presented as transition systems in a manner which allows us to view $App_{\mathfrak{I}}$ as defined in terms of the successive application of a transition function Δ_I to an initial state μ_0 to yield a sequence of states $\mu_0, \Delta_B(\mu), \Delta_B(\Delta_B(\mu_0)), \ldots$. In such cases, μ_{i+1} is derived by applying a mathematical operation to some structurally delimited component of μ_i. And for this reason, we can consequently view the executions of I as a model of processes which occur in time by likening transitions from μ_i to μ_{i+1} to either a change in the local physical state or an updating of a value stored in a distinct location.

In order to understand the conceptual basis of recursion-based models of computation, we must first recall that the term "recursive" is most fundamentally employed to describe certain informal modes of defining mathematical functions. A simple recursive definition of a function $f : D \to D$ will typically have the form $f(x) =_{df} t(x)$ where $t(x)$ is a complex functional term which contains one or more subterms containing the symbol f itself. Such definitions are circular in the sense that the value of the term $f(x)$ is defined in a manner which involves reference to whatever function (if any) such a definition succeeds in defining. But this sort of circularity need not be vicious in the sense of either leading to paradox or even to the necessity of recognizing that the function determined may not be defined for all $x \in D$. For $t(x_1, \ldots, x_n)$ will typically have a form resembling $f(r_1(x)) \ldots f(r_n(x))$ where $r_1, \ldots, r_n$ are explicitly defined functions which may contain f. But at least in most of the simple cases with which we will be concerned, f will be defined on D^n and we will know in advance that for all $x \in D$, $r_i(x) <_D x$ where $<_D$ is some explicitly given well partial order on the domain D.[24] Recursive definitions of this sort succeed in determining total functions because even though the value of $f(x_1, \ldots, x_n)$ is defined in terms of other values of the same function, these values will alway be smaller in the sense of $<_D$. And thus by expanding the definition of f, we will eventually reach values of $x_1, \ldots, x_n$ for which no further expansion is possible.

[24] I.e. $<_D$ is transitive, antisymmetric, and such that any linearly ordered chain $x_0 >_D x_1 >_D x_2 >_D \ldots$ is of finite length and contains no infinite antichains.

This informal description of recursive definition leads naturally to a temporal interpretation of what it means to apply a recursively defined function to an argument in its domain. This is most readily illustrated by considering a simple example. To this end, consider the set $S = \{|\}^*$ consisting of all finite strings over the alphabet $\{|\}$ (including the empty string denoted by ϵ). Let $\cdot$ denote the concatenation function on D and $tail : S \to S$ be defined by $tail(| \cdot \sigma) = \sigma$. Now consider the following definition of a function $double : S \to S$.

$$(4.9)\quad double(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ || \cdot double(tail(\sigma)) & \text{otherwise} \end{cases}$$

It should be clear that this definition determines a function of type $S \to S$ and that for all $\sigma \in S$, $double(\sigma) = \sigma \cdot \sigma$. Also note that the ordering on S, with respect to which $tail(\sigma)$ (which corresponds to the term $r_1(x)$ in the previous paragraph) is less than σ, is the ordering $\tau <_S \sigma$ iff τ is a proper prefix of σ. $<_S$ is clearly a well-partial ordering (in fact a well-ordering), and from this it follows by a standard set theoretic argument that (4.9) determines a unique total function on S.

Definitions like (4.9) thus serve a dual purpose in our mathematical discourse. On the one hand, they serve as acceptable means of introducing new function symbols. And on the other hand they serve as a means of specifying a sort of algorithm by which their values may be explicitly computed. Given that we are treating algorithms as the sort of entities whose execution leads to a temporally extended sequence of intermediate stages, assigning this status to (4.9) requires that we interpret its right-hand side in a certain manner. In particular, we must regard the occurrence of the functional expression $tail(\sigma)$ as corresponding to an operation which serves to decompose σ according to its structure. Viewed in this light, (4.9) specifies a method whereby the value of $double(\sigma)$ can be obtained by first decomposing σ as $| \cdot tail(\sigma)$ and then concatenating the string $||$ to the left of the result of applying *double* to the second component of this decomposition.

On this basis we may, for instance, use (4.9) to calculate $double(|||) = || \cdot double(||) = || \cdot (|| \cdot double(|)) = || \cdot (|| \cdot (|| \cdot double(\epsilon))) = || \cdot (|| \cdot (||)) = || \cdot (||||) = ||||||$. Such a chain of equalities may be seen as representing the result of a process whereby σ is "broken down" into its structural constituents until the base case $\sigma = \epsilon$ is reached. Such a process of

decomposition is most naturally understood as occurring in time. For note that in order to employ (4.9) so to derive these equalities we must, for instance, think of the input string ||| as being decomposed as $| \cdot tail(|||)$ and the result of $double(tail(|||))$ being calculated *before* the operation $\lambda\tau.|| \cdot \tau$ is applied to the value of the latter expression. And this in turn requires us to think of $||(= tail(|||))$ as being decomposed as $| \cdot tail(||)$ and the result of $double(tail(||))$ being calculated *before* the operation $\lambda\tau.|| \cdot \tau$ can be applied to the value of the latter expression.

But while the sort of process just described is naturally understood as occurring in time, there are a variety of factors which stand in the way of representing the expansion of a recursive definition as a computational process which may be described by a register-based model or even a transition system. Understanding why this is so will require us to examine more sophisticated forms of recursive definition as well as their formal representation as members of models of computation which treat recursion as a mathematical primitive.

One of the simplest and most familiar recursion-based models is the class of primitive recursive functions first introduced by Dedekind [27] and Skolem [132] as a model of computation on the natural numbers. The definition of a primitive recursive function can be taken as simultaneously identifying an extensional subclass of the set of functions $\mathcal{PR}^N \subseteq \mathbb{N}^n \rightarrow \mathbb{N}$ as being computable in a particularly elementary manner and also a set PR of functional terms whose members denote functions in $\mathcal{PR}^N$. It is well known that $\mathcal{PR}^N$ turns out to be too narrow to serve as a conceptually adequate definition of effective computability on the natural numbers. But for heuristic purposes, the definition of PR taken together with the manner in which these terms are assigned denotations in $\mathcal{PR}^N$ remains the canonical example of a recursive model.

The set of terms PR is defined as the class of terms over a formal language $\mathcal{L}_{pr}$ according to the following definition.

Definition 4.5.1. $\mathcal{L}_{pr}$ contains the *basis functional symbols* z,s (unary) together with the class of n-ary symbols π^n_i for all $i < n \in \mathbb{N}$. PR is then defined as the smallest class of terms containing these symbols and closed under the application of the following *functional combinators*:

i) Composition: If PR contains the m-ary symbol g and the n-ary functional symbols

$h_0, \ldots, h_m$, then it also contains the n-ary functional symbol $Comp[g, h_1, \ldots, h_m]$.

ii) Primitive Recursion: If PR contains the n-ary functional symbol g and the $n+2$-ary functional symbol h, then it also contains the $n+1$-ary functional symbol $PrimRec[g, h]$.

I will refer to the members of PR as *primitive recursive definitions*. Such definitions play a role analogous to program schemes in the flowchart model $\mathcal{P}$ in the sense that they function as linguistic descriptions of mathematical objects which may be interpreted as denoting functions via the construction of an appropriate definition of application.

The typical means of doing this is to assign each symbol f of $\mathfrak{L}_{pr}$ an arithmetic interpretation f^N as follows:

Definition 4.5.2. Let $f \in PR$. f^N is defined according to f's compositional structure as follows:

i) If $f = z$, then $f^N(x) = 0$ for all $x \in \mathbb{N}$.

ii) If $f = s$, then $f^N(x) = x + 1$ for all $x \in \mathbb{N}$.

iii) If $f = \pi^i_n$, then $(\pi^i_n)^N(x_1, \ldots, x_i, \ldots, x_n) = x_i$.

iv) If $f = Comp[g, h_1, \ldots, h_m]$, for $g, h_1, \ldots, h_m \in PR$, g^N m-ary and $h^N_1, \ldots, h^N_m$ n-ary, then $f^N(x_1, \ldots, x_n) = g^N(h^N_1(x_1, \ldots, x_n), \ldots, h^N_m(x_1, \ldots, x_n))$.

v) If $f = PrimRec[g, h]$, for $g, h \in PR$, g^N n-ary and h^N $n+2$-ary, then f^N is the unique function of type $\mathbb{N}^{n+1} \to \mathbb{N}$ such that $f^N(x_1, \ldots, x_n, 0) = g^N(x_1, \ldots, x_n)$ and $f^N(x_1, \ldots, x_n, y) = h^N(x_1, \ldots, x_n, f^N(x_1, \ldots, x_n, n), y - 1)$.

Employing this means of interpreting primitive recursive definitions, we may now define the model of computation $\mathcal{PR}$ to consist of the set terms PR, the input set $X_{pr} = \cup_m \mathbb{N}^m$ and the output set $Y_{pr} = \mathbb{N}$. In order to define application for this model, first note that the interpretation f^N of a term $f \in PR$ is a function f^N of type $N^n \to \mathbb{N}$. We may thus define $App_{\mathcal{PR}}(f^N, \langle m_1, \ldots, m_n \rangle)$ for $m_1, \ldots, m_n \in \mathbb{N}$ simply as the value of this function applied to these arguments – i.e., as $f^N(m_1, \ldots, m_n)$.

Given these components, $\mathcal{PR}$ clearly forms a model of computation in the sense discussed in Section 2. This is true despite the fact that App_{pr} is defined in terms of *extensional* functional application. Such a definition thus provides little insight into why we are justified in taking primitive recursive definitions as formalizing the sorts of computational processes

which I suggested are naturally determined by a definition like (4.9). Given any two functions g^N and h^N meeting the hypotheses of (4.5.2v), an elementary theorem of set theory states that there will exist a unique function f^N which can serve as denotation of the term f defined therein. But as I will discuss presently, the method by which this theorem is proved provides little direct insight into why function definitions of this sort ought to be treated as models of computational processes.

In order to understand why primitive recursive definitions fail to have this property, it is easiest to consider an example. Take the PR term

(4.10) $plus(x, y) = PrimRec[\pi^1_1, Comp[s, \pi^3_2]](x, y)$

which we would normally write using informal definition by cases notation as follows:

(4.11) $$plus(x, y) = \begin{cases} x & \text{if } y = 0 \\ s(plus(x, y - 1)) & \text{otherwise} \end{cases}.$$

In order to see why we are justified in using (4.11) to compute the value of $plus^N(n, m)$ for fixed n, m, it suffices to observe that although this function is defined on $\mathbb{N} \times \mathbb{N}$, the expression on the right-hand side of (4.11) (and in fact all primitive recursive definition in general) "goes down" on its second coordinate – i.e. the value of $plus(x, y)$ is defined in terms of an operation defined on $plus(x, y-1)$. We may think of the function $pred(x) = x-1$ as corresponding to a decomposition operator which plays a similar role to that of $tail(\sigma)$ as it appears in (4.9). In particular, $pred(x)$ may be thought of "breaking down" a natural number n into the subcomponents 1 and $n - 1$.[25] Since the domain $\mathbb{N}$ is obviously well-ordered by $<$ and for all x, $pred(x) < x$, it follows that we may view (4.11) as specifying a method for computing $plus^N(n, m)$ from the "top down." For instance, to compute $plus^N(2, 2)$, we may use (4.11) to calculate $plus(2, 2) = s(plus(2, 1)) = s(s(plus(2, 0))) = s(s(2)) = s(3) = 4$.

[25] If we think of natural numbers as freestanding mathematical objects, talk of numerical decomposition must obviously be treated metaphorically. If, however, we think of natural numbers as corresponding to finite (e.g. von Neumann) ordinals, then the function $pred(x)$ literally does structurally decompose x into y and $\{y\}$ where y is not only the ordinal predecessor of x but also $y \in x$. A similar interpretation is possible if we think of natural numbers as being given as unary numerals. For in case n is represented as $\sigma_n = |\ldots|$ ($n + 1$ times), the operation $pred(x)$ corresponds directly to the operation $tail(\sigma)$ defined over the set of strings S.

It is straightforward to formalize derivations of this sort using an equational calculus T_{pr} of uninterpreted PR terms. Such a calculus would include as axioms mimicking the different clauses of (4.5.2) as well as the first-order identity axioms. And on this basis, we could redefine $App_{\mathcal{PR}}$ such that $App_{\mathcal{PR}}(f^N, \langle m_1, \ldots, m_n \rangle) = q$ just in case the equality $f(\underline{m_1}, \ldots, \underline{m_n}) = \underline{q}$ was derivable in T_{pr}. The equations appearing in a derivation of this latter statement T_{pr} may thus be viewed as intermediate steps in a calculation such as that illustrated in the previous paragraph. Such a definition of $App_{\mathcal{PR}}$ is obviously more in line with our efforts to interpret primitive recursive definition as specifying procedures. But we may already see that certain problems arise in giving such a definition for the simple reason that there will be infinitely many distinct T_{pr} derivations with final step $f(\underline{m_1}, \ldots, \underline{m_n}) = \underline{q}$. And thus, taken by itself, such a calculus cannot be taken as completely specifying a method of computing with primitive recursive definitions.

This, however, is only one of the problems which occurs when we attempt to view $\mathcal{PR}$ as a model of computation on a descriptive par with a register-based models such as $\mathcal{P}$. For recall that our ultimate goal in this chapter is to assess the adequacy of various models of computation with respect to their suitability for serving as the class $\mathcal{Q}$ with respect to which the algorithmic realist wishes to reduce the members of $\mathcal{A}$ via an appropriate definition of $\underline{\leftrightarrow}$. I have already mentioned one feature which suggests that $\mathcal{PR}$ is not suitable for this task – i.e. the fact that this model is not Turing-complete and as such does not contain members which determine every intuitively effective function on natural numbers. Note, however, that the best-known examples of intuitively computable but not primitive recursive functions (e.g., the Ackermann function) may be shown to grow so fast that they are unlikely to correspond to the sort of practical algorithms whose status the realist is most concerned with accounting for.

There are two other straightforward but less often cited properties of $\mathcal{PR}$ which suggests that it is a poor candidate to choose for $\mathcal{Q}$. The first of these is that $\mathcal{PR}$ is not an open model in the sense of the previous section – i.e. it does not contain members which operate on structures other than natural numbers or employ as basic operations anything other than the class of basis functions enumerated above. This limitation would, for instance, prevent us from employing $\mathcal{PR}$ to directly model an algorithm such as PAL1 or PAL2 which

operated on strings rather than natural numbers.[26]

Another somewhat less obvious problem with attempting to employ $\mathcal{PR}$ as a general model of computation relative to which we can attempt to find implementation of everyday algorithms is that not all informal recursive definitions are of a form which can be directly represented as terms over PR. Consider, for instance, the customary recursive definition of the Fibonacci function $fib(x)$:

$$(4.12)\quad fib(x) = \begin{cases} 1 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \\ fib(x-1) + fib(x-2) & \text{otherwise} \end{cases} .$$

By the conventional standards discussed above, (4.12) serves not only as a means of introducing the functional symbol $fib(x)$, but also determines a procedure for computing its values. For instance, we may use (4.12) to compute the value $fib^N(3)$ as follows: $fib^N(0) = 1$, $fib^N(1) = 1$, $fib^N(2) = fib^N(1) + fib^N(0)$, $fib^N(2) = 1 + 1$, $fib^N(2) = 2$, $fib^N(3) = fib^N(2) + fib^N(1)$, $fib^N(3) = 2 + 1$, $fib^N(3) = 3$.

But now note that even if we elect to define $App_{\mathcal{PR}}$ relative to a equational calculus in the manner described above, there can be no PR term whose execution directly corresponds to the statements occurring in the derivation just described. This is *not* because the function

[26]Of course this does not mean that the definition of $\mathcal{PR}$ is not naturally extended so as to characterize the *language* L_{pal} as being primitive recursive. In order to do so, we would first have to take the standard step of saying that a set $X \subset \mathbb{N}$ is primitive recursive just in case its characteristic function is. We can next define an effective coding c of strings $w \in \{0,1\}^*$ into natural numbers. In this setting, it is easy to construct $p \in \mathcal{L}_{pr}$ so that $p^N(c(w)) = 1$ if and only $w \in L_{pal}$. Moreover it is also possible to do this so that the derivations of $p^N(c(w)) = i$ (for $i \in \{0,1\}$) mirror the stages in the execution of, e.g., PAL1 on w. For instance, *pal* could be defined so that its formal structure mirrored that of the informal recursive definition

$$pal(c(w)) = \begin{cases} 1 & \text{if } c(w) = 0 \\ 0 & \text{if } firstbit(c(w)) \neq lastbit(c(w)) \\ p(t(c(w))) & \text{else} \end{cases}$$

where c is defined so that $c(w) = 0$ just in case $w = \epsilon$ and if $w = a_1 \ldots a_n$, then $t(c(w)) = c(a_2 \ldots a_{n-1})$. But a familiar problem arises in making such claims precise: even if p is defined so that its compositional structure over $\mathcal{L}_{pr}$ mirrors that of *pal* as closely as possible, in order to informally calculate with this definition in the manner described above, the subrecursions corresponding to the computations of the auxiliary functions c, $firstbit$, $lastbit$, and t must all be carried out individually. This means that the computation of the value of $p^N(c(w))$ will be much longer than the corresponding computation of PAL1. And if he were to proceed in such a manner, the realist will again be responsible for specifying how sequences of steps in the computation of the former correspond to single stages in the computation of the latter. But this is likely to embroil him in exactly the same sorts of issues about constraining possible bisimulation relations which arose in Chapter 3.

fib^N itself fails to be primitive recursive (i.e. is not a member of the set of functions $\mathcal{PR}^N = \{f^N : f \in \mathcal{L}_{pr}\}$). In fact, it is a fairly straightforward exercise to show that $fib^N \in \mathcal{PR}^N$. The problem concerns the fact that since (4.12) makes two calls to the implicitly defined function $fib(x)$ on its right-hand side instead of one, there is no way of representing (4.12) as a PR term so that the course of its execution will mirror the calculation just described.

A more general way of describing this problem is to note that (4.12) employs a simple instance of what is known as *course of values recursion*. In the sort of course of values recursion with which I will be concerned, the value of the defined function $f(x_1, \ldots, x_n, y)$ is determined as a function of two or more of the finitely many values $f(x_1, \ldots, x_n, 0), \ldots, f(x_1, \ldots, x_n, y-1)$. The general form of the relevant sort of definition is thus given informally as

$$(4.13)\quad f(x_1, \ldots, x_n, y) = \begin{cases} g(x_1, \ldots, x_n) & \text{if } y = 0 \\ h(x_1, \ldots, x_n, f(r_i(y)), \ldots, f(r_m(y))) & \text{otherwise} \end{cases}$$

where $m \leq n$ and $r_i(y) = y \dot{-} n_i$ for $1 \leq i \leq m$. There is again a set theoretic theorem which ensures the well-definedness of a function satisfying such definitions. But definitions of this form cannot be converted directly to primitive recursive definitions since the primitive recursion combinator $PrimRec$ is defined so that the value of $PrimRec[g, h](x_1, \ldots, x_n, y)$ depends only on the single value of $f(x_1, \ldots, x_n, y-1)$ and not on $f(x_1, \ldots, x_n, y-1)$ and $f(x_1, \ldots, x_n, y-2)$ as employed in our informal definition of $fib(x)$. [27]

It may also be shown that even a simple case of course of values recursion like (4.12) can only be reformulated as a primitive recursive at the cost of making equational derivations of the values of $fib^N(x)$ substantially more complex than the sample carried out above. The exact extent of such speedup varies according to how we formulate the proof theoretic definition of execution for $\mathcal{PR}$. But the basic result is that equational derivations in the system T_{pr} will be exponentially longer than the sort of informal calculations of the value of

[27] It can, however, be shown that not only definitions in the form of (4.13), but also a slightly more general class where the fixed sequence of arguments $f(r_i(y)), \ldots, f(r_m(y))$ to h is replaced by $f \upharpoonright y$ do not lead outside of the class $\mathcal{PR}^N$. This is typically proven by showing that there a primitive recursive coding function by which the finite sequence $f \upharpoonright y = \langle f(0), \ldots, f(y-1)\rangle$ may be represented as a single number whose components may then be "decoded" in the definition of f. For details, c.f., e.g., [108].

$fib(x)$ illustrated above.[28] But now suppose an algorithmic realist simultaneously wishes to take (4.12) as directly expressing an informal algorithm FIB for computing the values of fib^N and also wishes to employ $\mathcal{PR}$ (with $App_{\mathcal{PR}}$ defined relative to T_{pr}) as a general model of recursive algorithms. In this case, he must be prepared to allow that individual steps in the execution of FIB on input n will be mirrored by exponentially long sequences of steps in any T_{pr} derivation of $fib(n) = m$.

Recall that one of the morals of Chapter 3 was that the realist ought to attempt to define the class of implementations he ultimately puts forth as $\mathfrak{Q}$ so as to avoid exactly this sort of situation. The present example thus suggests that there is a built-in motivation to seek out a model of computation which encompasses forms of recursion which are not directly representable as $\mathcal{PR}$-terms. A large variety of such models exists, most of them tracing their lineage to the class of so-called *general recursive definitions*. This model was originally proposed by Gödel [44] and Kleene [65] in an effort to formalize a notion of recursive function which subsumed recursive definitions (such as that of the Ackermann function) which determine intuitively effective functions which are not primitive recursive. What is more relevant given our current interests, however, is that when formulated correctly, the class of general recursive definitions also includes direct analogs of definitions like (4.12). However, in order to best facilitate viewing general recursive definitions in this manner, it will be useful to consider a version of the definition which differs somewhat from that of Kleene. The particular specific formalism I will employ was introduced by Paterson and Hewitt [106] as a recursive counterpart to the class of program schemes considered above.

Unlike many other models which have been proposed as generalizations of Gödel/Kleene general recursive definitions (e.g. [85], [47], [92]), these schemes correspond to *uninterpreted*

[28] In order to formulate this matter precisely, it is most convenient to assume that the addition is available as a basis function for primitive recursive definitions and that we do not count calls to $fib^N(0)$ or $fib^N(1)$ in calculating the length of a derivation. In this case, it can be shown that there are derivations of $fib^N(n) = k$ of length $2k - 1$ and thus that the running time of the algorithm FIB is $O(\phi^n)$ where ϕ denotes the golden ratio $(1 + \sqrt{5})/2$. But by proof theoretic means, such as those developed in [104] it may also be shown that the function $fib^N(n)$ is the Grzegorcyk class $\boldsymbol{E}^3 - \boldsymbol{E}^2$. This means that even if we include addition as a basis function in a primitive recursive definition, it follows that there can be no derivation of $fib^N(n) = k$ from a standard primitive recursive definition of fewer than 2^k steps. For future reference, these running time complexities should be compared with the $O(n)$ complexity of the simple flowchart machine based on P_{fib} for computing $fib^N(n)$.

functional definitions and may thus potentially be used to define an open model of computation which employs arbitrary (effective) operations on arbitrary (effectively presentable) structures. A recursion scheme $\hat{E}$ is thus defined over a first-order language $\mathcal{L}_E$, this time consisting of four finite classes of symbols: a finite class of predicates $\mathcal{P}_{\hat{E}} = \{P_1, \ldots, P_{n_p}\}$, a finite class of *basic function symbols* $\mathcal{F}^0_{\hat{E}} = \{f_1, \ldots, f_{n_f}\}$, a finite set of *defined function symbols* $\mathcal{F}^1_{\hat{E}} = \{F_0, F_1, \ldots, F_{n_F}\}$ (all or arbitrary arities) and a finite set of variables $\mathcal{V}_{\hat{E}} = \{x_1, \ldots, x_{n_v}\}$. The constituents of $\hat{E}$ itself are given by a defining equations E over $\mathcal{L}_E$ satisfying the following definition:

Definition 4.5.3. A *recursive equation* E is a statement of the form

$$F(y_1, \ldots, y_n) = \text{ if } P(z_1, \ldots, z_m) \text{ then } T(u_1, \ldots, u_{m'}) \text{ else } T'(v_1, \ldots, v_{m''})$$

satisfying the following conditions:

i) $y_1, \ldots, y_m \subseteq \mathcal{V}_E$ are distinct variables and

ii) $F_k \in \mathcal{F}^0_E$ is am m-ary defined function symbol

iii) $P \in \mathcal{P}_E$ is an n-ary predicate symbol

iv) T and T' are $\mathcal{L}_E$ terms (i.e. expressions built up inductively from the variables, basic function symbols and defined function symbols of $\mathcal{L}_E$)

v) $\{z_1, \ldots, z_m\} \subseteq \{y_1, \ldots, y_m\}$, $\{u_1, \ldots, u_{m'}\} \subseteq \{y_1, \ldots, y_m\}$, $\{v_1, \ldots, v_{m''}\} \subseteq \{y_1, \ldots, y_m\}$

The recursion equation E is said to *define* F_k.

A complete recursion scheme may be defined as follows:

Definition 4.5.4. A *recursion scheme* $\hat{E}$ is a finite sequence of recursive equations $\langle E_0, \ldots, E_{n-1}\rangle$ over a language $\mathcal{L}_E$ together with a designated *initial* defined function symbol $F_0 \in \mathcal{F}^0_E$ such that the following hold:

i) The defined function symbol on the left-hand side of E_0 is F_0.

ii) No defined function symbol appears on the left-hand side of more than one E_i.

iii) Every defined function symbol appearing in a term on the right-hand side of any E_i appears exactly once on the left-hand side of some equation E_j.

The recursion scheme $\hat{E}$ as a whole is said to define the function defined by E_0.

Like program schemes, recursion schemes need to be interpreted before they can be assimilated to our general definition of a model of computation. This is accomplished by providing an interpretation in the normal first-order sense for the predicate letters $\mathcal{P}_{\hat{E}}$ and basic functions symbols $\mathcal{F}^0_{\hat{E}}$ – i.e. by providing a pair $\langle D, I\rangle$ where D is the domain of the interpreted and I maps each n-ary predicate P_i into a subset D^n and each n-ary function into a function from D^n onto D. The intention of providing such an interpretation is that once the meanings of the terms in $\mathcal{P}_{\hat{E}}$ and $\mathcal{F}_{\hat{E}}$ are fixed, the meaning of the defined functions symbols in $\mathcal{F}^1_{\hat{E}}$ will become implicitly defined given the definition of $App_{\mathcal{E}}$ which will be stated below. Once this definition has been given, we will once again have defined a model of computation $\mathcal{E}$ in the sense of Section 2.

Since the definition of $App_{\mathcal{E}}$ turns out to be somewhat complex, it will be useful to first consider an example of a recursion scheme which illustrates how such schemes circumvent the limitations imposed by primitive recursive definitions. Consider, for example, the arithmetic languages $\mathcal{L}_{E_1} = \{<, pred, plus, 0, 1, 2\}$ and $\mathcal{L}_{E_2} = \{<, pred, suc, 0, 1, 2\}$ and the two schemas

$$\begin{aligned}
(4.14)\ \hat{E}_1\textbf{:}\quad & F_0(x) = \ \mathsf{if}\ x < 2\ \mathsf{then}\ 1\ \mathsf{else}\ plus(F_0(pred(x)), F_0(pred(pred(x))))\\
\hat{E}_2\textbf{:}\quad & F_0(x) \quad = \ \mathsf{if}\ x < 2\ \mathsf{then}\ 1\ \mathsf{else}\ F_1(F_0(pred(x)), F_0(pred(pred(x))))\\
& F_1(x, y) = \ \mathsf{if}\ y < 1\ \mathsf{then}\ x\ \mathsf{else}\ suc(F_1(x, pred(y))).
\end{aligned}$$

If we interpret $\mathcal{L}_{E_1}$ and $\mathcal{L}_{E_2}$ over $\mathbb{N}$ and assign their symbols their normal arithmetic meanings, it should be clear that the E_1 directly formalizes our original definition of the Fibonacci function via (4.12). E_2 may also be seen to define fib^N, where this time the addition function is not taken as a primitive function but rather defined simultaneously by the auxiliary equation or F_1.

We may now start to explore some of the difficulties involved precisely with the application function $App_{\mathcal{E}}$ for interpreted recursion schemes. Given an interpreted recursion scheme $\boldsymbol{E} = \langle \hat{E}, I\rangle$, it is natural to view the interpretation of $\hat{E}$ relative to I as giving rise to an equational calculus such that if $E \in \hat{E}$ has the form given in (4.5.3) and we expand I so that $y^I_i = a_i \in D$, $z^I_i = b_i \in D$, $u^I_i = c_i \in D$ and $v^I_i = d_i \in D$, then the calculus contains axioms of the form $F_k(y^I_1, \ldots, y^I_n) = T(u^I_1, \ldots, u^I_{m'})$ if $P^I(z^I_1, \ldots, z^I_{m'})$ is

true and $F_k(y_1^I, \ldots, y_n^I) = T'(v_1^I, \ldots, v_{m'}^I)$ otherwise. Such an equational calculus is non-deterministic in the sense that if multiple recursions or compositions are involved, there is no predetermined order in which the defining equations must be expanded. But as already noted by Kleene [65], two new possibilities arise: 1) it is possible that two different derivations will eventuate in different irreducible equations (i.e. statements of the form $F_0(t_1, \ldots, t_n) = t'$ for $t_1, \ldots, t_n, t'$ basic terms over $\mathcal{F}^0_{\hat{E}}$) 2) certain derivations will never terminate in statements of the form $F_0(t_1, \ldots, t_n) = t'$.

The second problem is closely related to the fact that $\mathcal{E}$, unlike $\mathcal{PR}$, is a Turing-complete model and thus contains instances which correspond to every partial recursive function (including those which are genuinely partial). However, the first problem is a feature of the equational calculus which may be resolved by adopting a stipulation about the order in which we go about expanding the terms appearing a recursion scheme. One such stipulation is to always expand terms on the right-hand side of a defining equation from the inside out – i.e. in an equation of the form $F(x_1, \ldots, x_n) = T(\ldots, T'(t_1, \ldots, t_m), \ldots)$ – we always make substitutions so as to evaluate T' before substituting T into another equation.[29] This is facilitated by thinking of $\hat{E}$ as a term calculus such that if T_1, T_2, T_3, T_4 are terms over $\mathcal{F}^0_{\hat{E}} \cup \mathcal{F}^1_{\hat{E}}$, then T' is derivable from T just in case $T \equiv T_2(\ldots, T_3, \ldots)$, $T' \equiv T_2(\ldots, T_4, \ldots)$ and the equations $T_1 = T_2(\ldots, T_3, \ldots)$ and $T_3 = T_4$ are both derivable. It then follows that by a result of Rosen [114] that the corresponding system of term reductions has the Church-Rosser property – i.e. if T' and T'' are distinct terms derivable from T by sequences of inside-out substitutions, then there is a term T''' which may be derived from both T' and T''.

Following a suggestion of Greibach, [47] we may now define application in the model $\mathcal{E}$ as follows. With a given interpreted recursion scheme $\boldsymbol{E} = \langle \hat{E}, I \rangle$, we define an associated context-free grammar $G_{\boldsymbol{E}}$. To do so, we first expand the $\mathcal{L}_{\hat{E}}$ with a set C of constant

[29] This convention has the result of making derivations within the term calculus deterministic by supplying a rule for choosing which substitution is to be carried out next in a derivation. But it turns out that there are many distinct conventions (which are generally called *reduction strategies*) which have the same effect. In Chapter 5, I will argue that not only is the adoption of some such strategy required in order to assimilate most recursive formalisms to our general definition of a model of computation, but also that the adoption of one strategy over another is highly non-trivial as executions relative to different strategies may be procedurally distinct.

symbols for each for member of D. The terminal symbols of $G_{\boldsymbol{E}}$ will be functional terms over $\mathcal{F}^0_{\hat{E}} \cup C$ and the non-terminal symbols of $G_{\boldsymbol{E}}$ are functional terms over $\mathcal{F}^1_{\hat{E}} \cup C$. We will associate two classes of production rules with each $E_i \in \hat{E}$ of the form given in (4.5.3), which for convenience we will assume are of the form

$$F(x_1, \ldots, x_n) = \text{ if } P(x_1, \ldots, x_n) \text{ then } T(x_1, \ldots, x_n) \text{ else } T'(x_1, \ldots, x_n)$$

where the dependence of P, T, and T' on some of the x_i may be vacuous. Let $a_1, \ldots, a_n$ be constants which will be assigned to the x_i, and a_i^I be their associated denotations in D. If $P^I(a_i^I, \ldots, a_i^I)$ is true, then $G_{\hat{E}}$ will contain the production $F(a_1, \ldots, a_n) \rightarrowtail T(a_1, \ldots, a_n)$, otherwise it will contain the production $F(a_1, \ldots, a_n) \rightarrowtail T'(a_1, \ldots, a_n)$. Finally, for all functional terms S_1, S_2 over $\mathcal{F}^0_{\hat{E}} \cup \mathcal{F}^1_{\hat{E}} \cup C$, G will also contain the productions $S_1U_1S_2 \rightarrowtail S_1U_2S_2$ where U_1 and U_2 are any two terms such that $U_1 \rightarrowtail U_2$ as defined previously.

Now define $\overset{*}{\rightarrowtail}$ to be the reflexive transitive closure of $\rightarrowtail$. For any interpreted recursive scheme $\boldsymbol{E}$, we may define the result of applying $\boldsymbol{E}$ to a sequence of values $d_1, \ldots, d_n \in D$ by looking at the result of derivations in $G_{\hat{E}}$ with initial term $F(a_1, \ldots, a_n)$ where $a_i^I = d_i$. In particular, relative to the interpretation I, $App_{\mathcal{E}}(\boldsymbol{E}, a_1, \ldots, a_n)$ will be defined as the (unique) terminal term T such that $E_0(a_1, \ldots, a_n) \overset{*}{\rightarrowtail} T$ if such a term exists and to be undefined otherwise. So for instance, the result of applying the scheme $\boldsymbol{E}_1$ defined in (4.14) under the standard arithmetic interpretation to that value 2 will be given by the derivation $F_0(2) \rightarrowtail plus(F_0(pred(2), F_0(pred(pred(2))))) \rightarrowtail plus(1, F_0(pred(pred(2)))) \rightarrowtail plus(1, 1)$. And thus since $F_0(2) \overset{*}{\rightarrowtail} plus(1, 1)$ and $plus(1, 1)^I = 2$, we have $App_{\mathcal{E}}(\boldsymbol{E}_1, 2) = 2$.

Having given a definition of application for interpreted recursion schemes, we are now justified in treating $\mathcal{E}$ as a model of computation on a par with the set of flowchart machines $\mathcal{P}$. And we may now also turn to the general question of comparing these two models with respect to their fitness for use in the abstractionist programme favored by the algorithmic realist. I will take it for granted that recursion schemes allow for a reasonably direct formalization of all intuitively effective forms of recursive definition.[30] But what is less

[30]This assumption is not entirely warranted since this model does not incorporate a typing mechanism such as that provided in a programming language based on either classical type theory (e.g. Haskell) or constructive type theory (e.g. MetaPRL). However, this deficiency does not impede the expression of any of the algorithms I will consider in this chapter.

clear at the moment is the relationship between this class of algorithms and those like PAL1 and PAL2 which appear to be most naturally represented using program schemes.

It would thus be useful to have a precise means of formalizing the expressive capacities of $\mathcal{P}$ and $\mathcal{E}$ so that they could be compared. As I have commented above, both models are already Turing-complete with respect to an arithmetic language containing only symbols for successor, predecessor, and zero. And thus for any recursion scheme $\boldsymbol{E}$ computing a given function of type $\mathbb{N}^m \to \mathbb{N}$, there will be a program scheme $\boldsymbol{P}$ equivalent in the sense of computing the same function according to the respective definitions of $App_{\mathcal{E}}$ and $App_{\mathcal{P}}$ and conversely. What a result of this sort does not demonstrate, however, is that these models are equivalent in the finer-grained procedural sense in which the realist is interested. For note that it is at least possible that there are certain functions $f : \mathbb{N}^m \to \mathbb{N}$ which can easily be computed by members of one model, but only computed by members of the other model by virtue of various roundabout means involving, say, arithmetic encodings such as those alluded to in Note 26. And since the use of such coding will almost certainly introduce computational detours of the sort considered in Section 3, it might still be that one of $\mathcal{P}$ or $\mathcal{E}$ contains more direct representatives than the other at least with respect to certain forms of algorithms.

As it turns out, not only does an asymmetry of this sort exist between $\mathcal{P}$ and $\mathcal{E}$, but that there are well-known algorithms which can be directly represented as recursion schemes but for which there exists no natural program scheme representation. In order to demonstrate this, it will be useful to employ a technique for comparing the expressive powers of schematic models first introduced by Paterson and Hewitt [106]. To this end, consider the following definition of equivalence between program and recursion schemes which strengthens the idea of extensional equivalence by quantifying over interpretations.

Definition 4.5.5. Let P be a program scheme and $\hat{E}$ a recursion scheme based on the same language $\mathfrak{L}$. Given an $\mathfrak{L}$ interpretation I, we say that the resulting machines $\mathcal{P}$ and $\mathcal{E}$ are *extensionally equivalent* just in case for all $d_1, \ldots, d_n \in D$, $App_{\mathcal{P}}(\boldsymbol{P}, \langle d_1, \ldots, d_n \rangle) = App_{\mathcal{E}}(\boldsymbol{E}, \langle d_1, \ldots, d_n \rangle)$ or both values are undefined. The schemes P and $\hat{E}$ are said to be *strongly equivalent* if they are extensionally equivalent for every $\mathfrak{L}$ interpretation.

The idea behind these definitions is as follows. If the schemes P and $\hat{E}$ are extensionally equivalent interpreted relative to a fixed interpretation I, this simply means that they compute the same function in extension. In and of itself, this means little with respect to their ability to represent the various kinds of algorithms since it may be achieved as a consequence of the fact that one of them (say P) contains predicate and function symbols which allow it to use a roundabout arithmetic encoding to achieve what the other (say $\hat{E}$) achieves directly. For instance, it might turn out that certain functions on strings are computable by flowchart machines only if we allow them to use an arithmetic encoding function (denoted by e) on strings which lets them emulate recursion on strings by applying arithmetic functions (denoted by $f_1, \ldots, f_n$) to codes of strings. Note, however, that the ability of a given scheme P to make use of these operations depends on the meanings assigned by interpretation I in the sense that only the "intended" arithmetic interpretation of these symbols will allow P to compute the same function as $\hat{E}$. By requiring as part of the definition of strong equivalence that P and $\hat{E}$ determine the same function under *all* interpretations, we thus rule out the possibility that P exploits an encoding which allows it to compute the function determined by $\hat{E}$ by operating indirectly on objects in some other domain.

On the basis of these definitions, the following two results can be established:

(4.15) a) For every program scheme P, there exists a strongly equivalent recursion scheme $\mathcal{E}_P$.

b) There is a recursion scheme for which there exists no strongly equivalent program scheme.

Understood informally, (4.15a) shows that up to the notion of strong equivalence, recursion schemes can do everything that program schemes can, but not conversely. But since our interest lies in the relative expressive capacities of $\mathcal{P}$ and E rather than in the general theory of schema, I will illustrate their significance through the use of examples rather than indicating how they are proven.[31]

[31] (4.15a,b) were first demonstrated by Paterson and Hewitt [106] in the paper which launched the field known as *comparative schematology*. More extensive treatments may be found in [47] and [32].

(4.15a) is demonstrated by showing how any program scheme P can be uniformly transformed into a recursion schemes $\hat{E}_P$ such that for all interpretations I of P, the sequences of derivations in the grammar induced by $\boldsymbol{E}$ will mirror the sequences of transitions in the execution of $\boldsymbol{P}$ as given by the operational semantics in (4.8). This is achieved by letting $\hat{E}_P$ contain a defined function for each of the k lines in P, each of which have as arguments variables $x_1, \ldots, x_n$ where n is the number of registers of P. These functions $F_i(x_1, \ldots, x_n)$ $(i \leq k)$ are defined so that the operation they induced on their arguments corresponds to the update on the values stored in $x_1, \ldots, x_n$ by the ith line of P.

The details of how $\hat{E}_P$ is constructed from P is illustrated by considering the following scheme corresponding the program scheme P_{fib} considered above:

$$
\begin{aligned}
&\hat{E}_{fib}\\
&\quad F_0(x_1,x_2,x_3) = F_1(x_1,x_2,x_3)\\
&\quad F_1(x_1,x_2,x_3) = F_2(x_1,1,x_3)\\
&\quad F_2(x_1,x_2,x_3) = F_3(x_1,x_2,1)\\
&\quad F_3(x_1,x_2,x_3) = \text{ if } x_1 < 2 \text{ then } F_8(x_1,x_2,x_3) \text{ else } F_3(x_1,x_2,x_3)\\
&\quad F_4(x_1,x_2,x_3) = F_5(x_1,x_2+x_3,x_3)\\
&\quad F_5(x_1,x_2,x_3) = F_6(x_1,x_2,x_2)\\
&\quad F_6(x_1,x_2,x_3) = F_7(x_1-1,x_2,x_3)\\
&\quad F_7(x_1,x_2,x_3) = F_2(x_1,x_2,x_3)\\
&\quad F_8(x_1,x_2,x_3) = x_2.
\end{aligned}
\tag{4.16}
$$

[32]

Consider the operation of the flowchart machine $\boldsymbol{P}_{fib}$ and the interpreted recursion scheme $\boldsymbol{E}_{fib}$ determined by P_{fib} and $\hat{E}_{fib}$ and the interpretation I_A which assigns the standard arithmetic meanings to the non-logical symbols in these schemes. On the understanding that the input is provided to $\hat{E}$ as the value of x_1 and that x_2 and x_3 are initially assigned the value 0, it is easy to see that there will be a one-to-one correlation between the derivation of terms in the grammar $G_{\hat{E}_{fib}}$ and the steps in the execution of $\boldsymbol{P}_{fib}$ relative to the

[32] I have here adopted the standard expedient of omitting the conditional on the right-hand side of a recursive equation with the understanding that the given equality is intended to hold regardless of the properties of the values of the variables. This does not involve a proper extension of the definition of a scheme since $F(x_1, \ldots, x_n) = T(x_1, \ldots, x_n)$ can be taken to abbreviate $F(x_1, \ldots, x_n) =$ if $\top$ then $T(x_1, \ldots, x_)$ else $T(x_1, \ldots, x_n)$ where $\top$ is some $\mathcal{L}_{\hat{E}}$ predicate which is true under all interpretations.

operational semantics supplied in (4.8).[33]

This observation suggests that not only do $\boldsymbol{P}_{fib}$ and $\hat{E}_{fib}$ both determine the same function relative to the $App_{\mathcal{P}}$ and $App_{\mathcal{E}}$ but that it may reasonably be said that they operate in the same manner in a step-by-step sense. Since it is clear that these relationships do not depend on the interpretation relative to which they are evaluated, we can see that the schemes P_{fib} and $\hat{E}_{fib}$ are also strongly equivalent. It thus seems that a strong case can be made that a register-based model like $\mathcal{P}$ may be assimilated to a sufficiently sophisticated recursion-based model like $\mathcal{E}$ in a manner that preserves the step-by-step behavior of its members. And it thus seems reasonable to conclude that a model like $\mathcal{E}$ will contain *at least* as many direct representatives of the members of $\mathcal{A}$ as does a model like $\mathcal{P}$. And per (4.15b), it may also appear that $\mathcal{E}$ is *strictly* more flexible in this regard. I will ultimately argue in Chapter 5 that this appearance is deceptive. But in order to see why this is so, it will be useful to first observe the significance of (4.15b) with respect to everyday algorithms.

The canonical example of a recursion scheme which is not strongly equivalent to any program scheme (again due to [106]) is given as follows:

(4.17) $\hat{E}_0 : F_0(x) =$ $\mathsf{if}\ P(x)\ \mathsf{then}\ x\ \mathsf{else}\ f(F_0(g(x)), F_0(h(x)))$.

The fact that there is no program equivalent to $\hat{E}_0$ may be seen by considering the operation of the class of interpreted recursion schemes $\boldsymbol{E}_0^n = \langle \hat{E}_0, I_n \rangle$ where the interpretation I_n over the language $\mathcal{L}_0 = \{f, g, h\}$ is defined as follows: i) I_D is the set $Term_{\mathcal{L}_0}$ of functional terms over the language $\mathcal{L}_0$; ii) $P_D(t)$ is true just in case $t \in Term_{\mathcal{L}_0}$ contains at least n distinct occurrences of f and g; iii) $f^{I_n} = f$, $g^{I_n} = g$, $h^{I_n} = h$. As the reader is invited to confirm, the value of $App_{\mathcal{E}}(\boldsymbol{E}_0^n, x)$ under I^n will correspond to the $\mathcal{L}_0$ term t whose compositional structure can be visualized as a binary tree whose internal nodes correspond to applications of f and whose leaves correspond to each of the 2^n correctly parenthesized terms which may be composed by iterating occurrences of g and h n times

[33] In particular, note that a transition between states $\langle a_1, a_2, a_3, k \rangle$ and $\langle a_1', a_2', a_3', k' \rangle$ in $\boldsymbol{P}_{fib}$ will be mirrored by a reduction of the form $F_k(a_1, a_2, a_3) \rightarrowtail F_{k'}(a_1', a_2', a_3')$ in $G_{\hat{E}_{fib}}$. In this way, transitions between states localized by line numbers k and k' in $\boldsymbol{P}_{fib}$ will correspond to calls by the function F_k calling to the function $F_{k'}$ in $G_{\hat{E}_{fib}}$. And on this basis we may easily see that not only that is there a one-to-one correspondence between steps in the execution of $\boldsymbol{P}_{fib}$ from initial state $\langle n, 0, 0, 1 \rangle$ and steps in the (unique) derivation in $G_{\hat{E}_{fib}}$ with initial term $F_0(n, 0, 0)$ but also that these states and terms will closely reflect each other's structure.

on the variable x. For instance, we have $App_{\mathcal{E}}(\boldsymbol{E}_0^1, x) = f(g(x), h(x))$, $App_{\mathcal{E}}(\boldsymbol{E}_0^2, x) = f(f(g(g(x)), g(h(x))), f(g(h(x)), h(h(x))))$, etc. It may now be shown by a combinatorial argument that any flowchart machine which computes this value as output must contain at least $n+1$ registers. And from this it follows that there can be no *single* program scheme which computes the appropriate output for all interpretations of the form I_n.

This example is based on a class of so-called *free* or *Herbrand interpretations* – i.e. interpretations where the value of term is given by itself. And since there seems to be no prior significance attached to the function $\lambda x.App_{\mathcal{E}}(\boldsymbol{E}_0^n, x)$, the significance of the fact that there is no flowchart machine which computes the same function under all interpretations may also be unclear. Note, however, that the form of the recursion described by the scheme $\hat{E}_0$ directly matches that of one of the paradigmatic members of the class $\mathcal{A}$ – i.e. MERGESORT.

We originally specified the algorithm MERGESORT in Chapter 2 using informal definition by cases notation as follows:

(4.18) $mergesort(x) =$

$$\begin{cases} \ell & \text{if } |x| \leq 1 \\ merge(mergesort(firsthalf(x)), mergesort(secondhalf(x))) & \text{otherwise} \end{cases}.$$

On its intended interpretation, (4.18) is to be understood so that x varies over finite lists composed of elements from an arbitrary domain D ordered by $\prec$, $firsthalf(x)$ denotes the function which returns the first half of a list (inclusive of its middle element if it is of odd length), $secondhalf(x)$ denotes the function which returns the second half of a list, and $merge(x_1, x_2)$ denotes the list merging functions described in Chapter 2.[34]

[34] $merge(x_1, x_2)$ may itself be specified recursively. Its definition may be given informally as

$$merge(x_1, x_2) = \begin{cases} x_1 & \text{if } |x_2| = 0 \\ x_2 & \text{if } |x_1| = 0 \\ head(x_1) \cdot merge(tail(x_1), x_2) & \text{if } head(\ell_1) \preceq head(x_2) \\ head(x_2) \cdot merge(x_1, tail(x_2)) & \text{otherwise} \end{cases}$$

where $tail(d \cdot \ell) = \ell$ and $head(d \cdot \ell) = a$ and $\prec$ is the order defined on D. This definition can readily be expressed as a recursion scheme M_{ms} which can then be taken together with the definition E_{ms} to give a system of mutually recursive equations like (4.14) above. But since the ability to further decompose the merging function will not be significant here, the corresponding function symbol $m(x)$ in the recursion scheme corresponding to (4.18) will be treated as a basic function symbol.

We are now in a position to observe that the preceding definition of Mergesort may also be formally expressed as a recursion scheme over the language $\mathfrak{L}_{ms}$ consisting of a single predicate letter Q, a single defined function S, and basic functions m, h_1, h_2 symbols. This scheme would have the form

(4.19) $\hat{E}_{ms} : S(x) =$ if $Q(x)$ then x else $m(s(h_1(x)), s(x(h_2(x))))$.

Suppose we interpret E_{ms} relative to the interpretation I_{ms} with domain D^* consisting of finite lists of elements of the set D ordered by $\prec$. And suppose we additionally define $Q^{I_{ms}}$ so as to hold of just those members of D^* of length less than or equal to 1, $h_1^{I_{ms}}(x) = firsthalf(x)$, $h_2^{I_{ms}}(x) = secondhalf(x)$ and $m^{I_{ms}}(x) = merge(x)$. Then under this interpretation, (4.19) will not only compute the function $sort : D^* \to D^*$ which takes unordered lists into ordered ones, but will also do so in the same manner that we think (4.18) determines this function. In other words, relative to operational semantics by which $App_{\mathcal{E}}$ has been defined, the value of $App_{\mathcal{E}}(\boldsymbol{E}_{ms}, x)$ under the interpretation I_{ms} will be derived by a sequence of terms which mirror those by which we would derive the value $mergesort(x)$ by employing (4.18) as an informal recursive procedure.

But now note that modulo the choice of names for basic functions and predicates, the schema (4.19) has exactly the same structure as (4.17). It thus follows that we may apply the above reasoning to conclude that there is no program scheme which is strongly equivalent to (4.19). The significance of this observation may not be immediately apparent. For it is only relative to the interpretation I_{ms} that we take the scheme (4.19) to express the same algorithm as (4.18) – i.e. MERGESORT. Thus the non-existence of a program scheme which is extensionally equivalent to $\hat{E}_{ms}$ under *all* interpretations is not necessarily of interest. In particular, by itself it does not allow us to conclude that no program scheme computes the same function as $\hat{E}_{ms}$ under the intended interpretation I_{ms}.

But it turns out this stronger result may also be shown to be true. To see why, it suffices to note that in the argument sketched above no program scheme that is strongly equivalent to $\hat{E}_0$ goes through under any interpretation in which the values of the terms labeling the leaves of the tree associated with $App_{\mathcal{E}}(\hat{E}_0, x)$ are distinct. In the current case, these labels will correspond to terms of the form $h_{i_1}(h_{i_2} \ldots (h_{i_{\lceil \log_2 x \rceil}}(x)))$. Under the intended

interpretation, $h_1^{I_{ms}}(x) = firsthalf(x)$ and $h_2^{I_{ms}}(x) = secondhalf(x)$, these values will all be pairwise distinct any time the constituents of x are themselves distinct. From this it follows that as long as D is infinite (as will generally be the case in natural instances such as $D = \mathbb{N}$), it follows that no single program scheme will compute the same function as $\mathcal{E}_{ms}$ relative to the intended interpretation I_{ms}. But since this function is exactly the function $sort : D^* \to D^*$ which takes arbitrary lists over D into sorted lists, this mean that no single interpreted program scheme (i.e. flowchart machine) can function as a sorting algorithm on an infinite domain without resorting to some sort of arithmetic encoding of sequences.

Since sorting procedures comprise an important and well-studied class of informal algorithms, this observation appears to provide substantial evidence that the class $\mathcal{P}$ of flowchart machines does not contain direct representatives of certain algorithms which are directly represented by members of $\mathcal{E}$. And since we have seen that the converse is *not* true (i.e. there is a strong sense in which program schemes *can* be operationally assimilated to recursion schemes), it follows that the realist is equipped with a natural reason to prefer $\mathcal{E}$ over $\mathcal{P}$ as a choice for $\mathcal{Q}$. But this conclusion should not be mistaken for the much stronger claim that $\mathcal{E}$ meets the realist's more general explanatory needs in selecting the class $\mathcal{Q}$. For note that thus far I have offered no argument intended to show that $\mathcal{E}$ is sufficiently broad to contain a member which directly represents *every* algorithm in $\mathcal{A}$. And perhaps more significantly, I have also not broached the topic of whether $\mathcal{E}$ will contain a unique or identifiably canonical representative corresponding to the algorithms for which it does contain plausibly direct mathematical models.

Our prior experiences suggest that either of these questions may pose a substantial challenge to the viability of the thesis that $\mathcal{E}$ could be taken by the realist as an adequate candidate for $\mathcal{Q}$. For note that even though the foregoing considerations suggest that $\mathcal{E}$ is an improvement over $\mathcal{P}$ with respect to direct representability, there is no reason to think that the gain in expressivity achieved by moving from one class to the other is enough to ensure that the former class will be able directly mirror modes of procedural specification which we may not yet have considered. And even if it could, the realist might still be left to work out the technical details of defining $\underline{\leftrightarrow}$ to account for the relationship which different recursion schemes bear to one another if they are to be taken as equally good representatives of the

same algorithm. Were I interested in arguing for the positive thesis that $\mathfrak{Q}$ can be taken to be $\mathcal{E}$, a more thorough study of the modes would have to be undertaken.

But as I will explore in detail in Chapter 5, there are substantial reasons to be dubious about the adequacy of taking $\mathfrak{Q}$ to be $\mathcal{E}$. In order to get an idea of why this is so in advance, however, we may begin by reflecting further on how the adoption of open computational models was taken above to lighten the theoretical burden of the algorithmic realist. This generalization allows the realist to construct mathematical models whose primitive functions and predicates mirror those appearing in informal specifications of algorithms arbitrary, thus allowing computational operations to be performed over the course of a single step which might otherwise require further mediation in the case of a closed model like $\mathcal{PR}$ or $\mathcal{T}$. The other step we have been considering is that of adopting a model whose basic mode of operation is made on recursion rather than than as iteration (in the case of transition- and register-based models). Based on the foregoing, it may appear that this generalization can be given a similar justification in the sense that we have just observed that as per (4.15) it appears that there are informal specifications of algorithms which can be directly represented by recursions schemes but not by flowchart machines or any other similar model of computation.

In Chapter 5, I will argue that while the first of these steps is a legitimate means of attempting to construct direct models of informally specified algorithms, the second step trades on a misunderstanding about the theoretical status of recursion-based models. In particular, I will argue that although it is legitimate to view an informal recursive definition like (4.11) as a sort of template for a method of calculating the values of $fib^N(x)$, certain definitions of this form – in particular those like (4.12) involving limited course of values recursion – cannot be seen as expressing fully explicit procedures. As such, models of computation like $\mathcal{E}$ which are designed to directly reflect the informal modes of computation deriving from these definitions must not be taken as allowable analyses of the general notion of implementation. In order to be treated in this manner, I will argue that the members of such models must be converted into deterministic transition systems. For all the models we have considered other than $\mathcal{E}$, this sort of conversion has been straightforward. But it turns out that converting certain complex recursion schemes into this sort of transition

system reintroduces complexities of the sort encountered in Section 3. And for this reason, it turns out that recursion-based models like $\mathcal{E}$ do not offer the realist a genuine means of escaping the same basic problems about the definition of $\underline{\leftrightarrow}$ which arose there.

Chapter 5

What algorithms could not be

5.1 Introduction

I have thus far spoken little of theorists who would describe themselves as explicit proponents of algorithmic realism. Since one of the announced aims of this work is to refute this view, this is likely to seem peculiar. For as I have noted in Chapter 2, the discourse of computer scientists is overwhelming realistic in how it speaks about algorithms. And for this reason, one might naturally think of computer scientists as natural-born algorithmic realists. However, if most workers in this field harbor these tendencies, they largely keep their views to themselves. For as I touched on briefly in Chapter 1, questions about the foundational status of algorithms have tended to fall into a sort of intellectual no man's land between mathematics, computer science and formal ontology. For this reason, what little attention questions about the status of algorithms have provoked has arisen in narrow theoretical contexts (e.g. the discussion of levels of explanation in cognitive science in the sense of [83]) wherein the sort of general ontological questions which are relevant to the evaluation of algorithmic realism have been largely ignored.

To illustrate one reason why this situation has arisen, consider again the status of a statement φ which is accepted by the mathematical community at large but which, at the current time, can only be practically derived by the application of an algorithm A. I argued in Chapter 1.4 that such statements abound in contemporary mathematics. For concreteness, however, we may take φ to be a simple statement about the value of a function $f : \mathbb{N} \to \mathbb{N}$ at a fixed value n, say $\varphi \equiv f(n) = m$. I also argued in Chapter 1.4 that if our only evidence for φ is grounded in the application of A to n, then A should be proven correct with respect to a (fixed definition of) f before our belief in $f(n) = m$ derived in this manner ought to be considered justified. This fact is generally acknowledged within

the mathematical community.

Such a proof of correctness typically involves the construction of a formal model M_A of A. I argued previously that M_A must itself be a mathematical object in order for the correctness proof to have the significance of demonstrating that our belief in φ can be justified on the basis of having carried out A. However, both the construction of M_A and the task of finding a correctness proof will often be carried out by computer scientists using the specialized methods of algorithmic analysis and formal verification. Thus although M_A will have been constructed precisely to model the operation of A, the question of whether this procedure is *identical* to M_A will be of little concern to mathematicians whose primary concern lies not with the algorithm A itself, but only with knowing whether they are justified in using it to determine the values of f. But it is widely believed that the use of A for this purpose can be justified by reasoning about M_A alone. And for this reason, no practical or theoretical necessity has forced computer science to adopt a fixed position about the relationship between A and M_A.

I suggested in Chapter 1 that this way of looking at matters is somewhat misleading. For without an additional demonstration that M_A faithfully represents A's mode of operation, we have no reason to accept a correctness proof for M_A as justifying the application of A (as opposed to some other algorithm) to compute the values of f. The necessity of providing such an argument would naturally focus attention directly on the status of A itself. And although this fact appears to have been overlooked in computer science proper, its clear epistemic significance would suggest that it could also have been taken profitably within logic or philosophy of mathematics. The fact that it has not can be attributed to a variety of factors which I have touched on briefly in previous chapters.[1] With the partial exception of two programmes I will discuss in this chapter, it thus is difficult to find mention of the thesis that algorithms ought to be identified with mathematical objects, let alone careful argumentation for or against such a view.

---

[1]Among these are i) confusion about the significance of so-called "computer proofs" in the justification of mathematical beliefs derived through computation (Chapter 1.1), ii) the predominance of possible world-based semantics (a la Carnap [17] and Montague [90], [89]) over verification-based semantics (a la Dummett [30] and Prawitz ([109]) in intensional logic (Chapter 1.2), and iii) the confusion about the relationship between Church's Thesis and the intuitive conception of algorithm (Chapter 2.2.3.2).

Evidence that something like a view resembling algorithmic realism does indeed underly the received view within computer science is provided by the rare instances in which exegetical factors force questions about the foundational status of algorithms to the fore. A paradigmatic example of this phenomenon may be found in Donald Knuth's ground breaking survey of algorithmic analysis *The art of computer programming, Volumes 1-3* [72].[2] Midway through the first chapter of the first volume of this work, Knuth makes the following observation:

> So far our discussion of algorithms as been rather imprecise, and a mathematically oriented reader is justified in thinking that the preceding commentary makes a very shaky foundation on which to erect any theory about algorithms. (p. 7)

Here Knuth here explicitly acknowledges the need for a formal mathematical model on which to base the theory of algorithmic analysis that he then goes on to develop. He follows through in this by providing a definition of a class of models in the form of what he refers to as *computational methods* which he explicitly proposes to *identify* with algorithms.[3]

Such a method is defined in essentially the same way as a transition system – i.e. as a quadruple (Q, I, Ω, f) consiting of a class of states (Q), an input function (I), a class of terminal states (Ω) and a transition function (f). Knuth then goes on to show how Euclid's algorithm can be represented by such a model and then formally proven to be correct in essentially the manner considered in Chapter 1. On this basis, it is tempting to view Knuth as having committed himself not only to algorithmic realism, but to the explicit proposal that algorithms are identical to computational methods. But there is also reason to doubt that he truly intends to to adopt this view as after the section in which the definition of computational methods is presented, Knuth never refers to the definition of a computational model again in any of the three volumes of [72] which have thus far been published. In particular, he makes no attempt to show how computational properties like

---

[2]The first edition of the first volume of this work was published in 1969 and was the first textbook-style presentation of a many topics and techniques in the analysis of algorithms which have now become standard in computer science curricula. In particular, not only did Knuth [67], [66] offer the first systematic argument for the use of asymptotic notation to compare the efficiency of practical computing procedures, but he was also the first to apply this method to many of the well known procedures which I have mentioned in prior chapters. For this reason, Knuth is not only regarded as the doyen of algorithmic analysis, but his books remain a standard reference in the field. Knuth states that his survey is ultimately to contain eight volumes; the fourth and fifth have recently been circulated in draft form.

[3]In particular, he appears to take the following remark as a definition: "[A]n *algorithm* is a computational method which terminates in finitely many steps for all x in I" ([72], p. 8, his italics)

running time complexity can be analyzed in terms of those of computational methods or to show how standard informal arguments involving, say, the use of the Master Theorem or decision tree methods (cf. [24]) can be formalized using these models.

This is particularly significant because the main exposition of [72] is unique among mainstream texts on the analysis of algorithms in that most of the procedures it treats are specified both as pseudocode and also as programs in a specially designed formal programming language known as `MIX`. This language extends that of the RAM machines studied in Chapter 4.4 with a variety of "high level" features like conditional and unconditional jump instructions. It is intended, however, to be interpreted so that every `MIX` program π uniquely determine a machine M_π which is a member of a class of sophisticated register machines $\mathcal{MIX}$. Knuth motivates this model of computation both on the basis of its fidelity to actual computer hardware and also because it allows for the relatively direct expression of complex operations and data structures which appear in his pseudocode specifications of algorithms. Unlike his original definition of computational methods, `MIX` programs *are* consistently employed throughout [72] to make precise procedures which have been stated in pseudocode, and also to formally prove correctness and complexity results. In fact, in this work, results pertaining to an algorithm A are standardly presented both for pseudocode specifications of A using asymptotic notation and also for using exact notation for the $\mathcal{MIX}$ counterpart of such a specification.

Giving proofs of this sort in full detail is often complex and laborious. And for this reason, it is reasonable to assume that Knuth assigns more than just instrumental importance to the `MIX` model. However, he never explains what he takes the relationship between algorithms and $\mathcal{MIX}$ machines to be. In particular, although he consistently speaks as if `MIX` programs can be used to express informal algorithms precisely. For this reason, much of [72] can viewed as an attempt to work out the details of algorithmic realism according to the view that algorithms may be identified with the $\mathcal{MIX}$ machines which are determined by the `MIX` programs which are used to express them. However Knuth never states that he views this as a foundational thesis in the same manner as his proposal about computational methods described above. And for this reason, while it seems reasonable to think of Knuth as attempting to work out an explicit form of algorithmic realism, he does not appear to

treat this as a proposal which he regards as requiring explicit argument.

Modern textbook treatments of the analysis of algorithms such as [24], [46] and [125] do not go to the trouble of specifying the algorithms relative to a specific machine model or programming language. But as we saw in Chapter 2, this does not mean that they do not share Knuth's generally realistic tone about their status as mathematical objects. This is particularly apparent from the justifications which are standardly provided for the use of asymptotic complexity hierarchies to measure the relative efficiencies of different algorithms. For recall that the practice of ascribing asymptotic as opposed to exact running time to individual algorithms is standardly justified by noting that while machines which we would intuitively taken to be implementations of the same algorithm may differ in exact running time, they will always fall within the same asymptotic class. And since an independent argument can be offered why asymptotic running time is better gauge of practical efficiency than exact running time, this reason is offered as part of a justification for failing to enter into the detailed implementation-specific issues which Knuth must confront. However rather than being taken as desiderata pertaining to the ontological relationship between algorithms and implementations, these considerations are merely presented as practical justifications for avoiding the need to consider specific implementations in the course of constructing subsequent proofs establishing the complexity of individual algorithms.

Knuth's proposal notwithstanding, there have thus been very few sustained attempts within computer science to show that any particular formal model can be employed to uniformly reconstruct a substantial portion of the informal reasoning about algorithms in the manner discussed in Chapter 3. The two notable exceptions to this generalization correspond to well worked-out proposals put forth by Yiannis Moschovakis and Yuri Gurevich for explicitly identifying algorithms with the members of well defined mathematical classes. Both of these programmes are mature and systematic in the sense that they have been developed over the course of a number of papers which address both technical and foundational issues. And although neither Moschovakis nor Gurevich explicitly acknowledge that algorithmic realism is a view which might be false in a sense which would require an explicit defense, their proposals can both be reasonably be interpreted as providing such a positive argument for such a view.

It is in this sense that the programmes of Moschovakis and Gurevich differ from Knuth's original proposal to identify algorithms with computational methods. For in addition to defining classes of models, each theorist goes on to offer a detailed defense of why *his own* model (as opposed to some other) is the appropriate one in which to locate algorithms. And to varying degrees each also attempts to justify this claim by providing a principled mathematical analysis of how we may justifiably ascribe complexity theoretic properties directly to algorithms. Unlike other authors whose work touches on the foundational significance of algorithms, Moschovakis and Gurevich are thus sensitive to the requirements of developing algorithmic realism as a philosophical thesis which needs to be precisely framed and defended.

But having said this, however, it will also be clear that the programmes which these theorists propose can be easily classified as corresponding to specific instances of the strategies for defending algorithmic realism outlined in Chapter 3. In particular, Gurevich's view can be characterized as a straightforward form of reductionism whereby it is claimed that algorithms may be directly identified with instances of a class of very general models of computation known as *abstract state machines* [ASMs]. And similarly, Moschovakis's programme may be characterized as a form of abstractionism whereby it is claimed that algorithms may be identified with models of computation known as *recursors* under an equivalence relation known as *recursor isomorphism*.

But not only are the reductive strategies favored by Moschovakis and Gurevich already familiar, but so are the ASM and recursor models relative to which they seek to carry them out. For as we will see below, an abstract state machine is essentially nothing more than a particular form of transition system. And similarly, a recursor is nothing more than a form of semantically interpreted recursion scheme of the sort considered in Chapter 4.. It thus follows that the programmes of Moschovakis and Gurevich fit neatly into the framework of computional models developed in Chapter 4. And for this reason, it is also easy to predict the sorts of rhetorical exigencies with which the proposals must contend. For instance, in proposing to identify algorithms with equivalence classes of recursors, Moschovakis must show that recursor isomorphism is both intensionally and extensionally adequate in the sense discussed in Chapters 3. And in proposing to identity algorithms

with ASMs, Gurevich must not only present some means of selecting a unique ASM M with which to associate every informally specified algorithm A, but he must also argue that the computational properties of M exactly coincide with those of A (i.e. that M does not possess any "artefactual" properties not shared by A).

In Sections 2 and 3 I will attempt to show that the specific proposals put forth by Moschovakis and Gurevich run afoul of these predictable difficulties. And I will argue that for these reasons, their programmes both *fail* as defenses of algorithmic realism. But what turns out to be significant about these proposals is not so much the details of the recursor and ASM models themselves, but rather the way in which Moschovakis and Gurevich argue that their chosen model represents the best framework for mathematically analyzing general discourse about algorithms. Putting aside the details of the ASM and recursor models themselves, the two proposals stand in stark contrast: Moschovakis argues that this task is best achieved relative to a recursion-based model and Gurevich argues that it is best achieved relative to a register-based model. Thus not only are the mathematical features of the formalism which they employ fundamentally different, each theorist also marshals different considerations in favor of adopting his computational paradigm in order to analyze the general concept of algorithm.

In the case of Moschovakis, this amounts to two claims: 1) recursors are a more general model of computation than transition- and register-based models in the sense that the latter may be assimilated to the former in the sense discussed in Chapter 5; 2) recursors more accurately reflect the degree of abstraction from implementation-specific details at which informal complexity theoretic analyses are standardly given. Gurevich, on the other hand, offers both positive and negative consideration on behalf of his proposal. On the one hand, he argues that ASMs most accurately reflect the degree of abstraction at which algorithms are generally specified in computational practice in the sense that when formulated with maximum generality, the ASM model contains implementations which may employ arbitrary (effective) operations over arbitrary (effectively presented) data types. And on the other hand, he argues contra Moschovakis that recursion-based models are not an appropriate formalism for analyzing the informal conception of algorithm because such models may fail to completely specify the sequence of steps which must be transacted to execute their

instances on fixed inputs. For this reason, he argues that the concept of algorithm ought to be analyzed in terms of a transition- or register-based model whose instances completely determine the ordering of states in their executions.

This debate between Moschovakis and Gurevich brings to light several fundamental issues on which I will argue that the fate of algorithmic realism ultimately depends. Among these are not only Gurevich's worries that recursive models may not serve to adequately analyze the background notion of implementation, but also more refined questions about whether recursive implementations can be uniformly transformed into transition-based ones in a manner which preserves not only the computational properties of the algorithms which they are claimed to implement, but also their identity conditions. Although the debate between Moschovakis and Gurevich over these issues is largely confined to a few parentethical remarks, this discussion still represents one of the most informed exchanges about technical issues surrounding algorithmic realism of which I am aware. And it is for this reason that it is worth considering the proposals of Moschovakis and Gurevich in detail.

My plan for the rest of this chapter will thus be as follows. In Sections 2, I will present Moschovakis's programme together with recursor model. In Section 3, I will present Gurevich's programme together with the ASM and his critiques of Moschovakis. Along the way, I will attempt to highlight why these proposal fail as defenses of algorithmic realism while also highlighting issues about the relationship between recursion- and iteration-based implementations of the sort just mentioned.

5.2 Moschovakis and recursors

5.2.1 The recursor model

Using the terminology of Chapters 2 and 3, Moschovakis' proposal can be described as a form of abstractionism which seeks to identify algorithms with equivalence classes of implementations known as *recursors* $\mathcal{R}$ relative to the relation of recursor isomorphism $\underline{\leftrightarrow}_{\mathcal{R}}$. As I have already indicated, I claim that this proposal fails for the now familiar reason that the relation $\underline{\leftrightarrow}_{\mathcal{R}}$ turns out to be too fined grained. This means in particular that it may be shown that it fails to hold between particular pairs of implementations M_1 and M_2 which

we have good reason to regard as implementing the same algorithm A. In the terminology of Chapter 3, this means that $\underline{\leftrightarrow}_{\mathcal{R}}$ is not an *extensionally adequate* bisimulation relation. And as a consequence of this, it can be shown there are $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant properties which do not correspond to intrinsic properties of individual algorithms. As we will see, this stands in conflict both with the intensional adequacy condition on abstractionism and also with Moschovakis' stated aims in presenting a mathematical foundation for a theory of algorithms.

As a defense of algorithmic realism, I thus believe that the case against Moschovakis' programme is over-determined. But at the same time, his proposal also corresponds to the best worked-out and most mathematically sophisticated attempt to vindicate algorithmic realism which has yet been proposed in the literature of computer science. This is true in three respects. First, Moschovakis is the only theorist to date who explicitly acknowledges that algorithms must be defined by abstraction with respect to another category of mathematical objects (in his case recursors) and thus the only theorist who has even begun to take account of the particular technical burdens that come with this as discussed in Chapter 4. Second, he proposes a means by which certain forms of recursors can be taken as canonical representatives of $\underline{\leftrightarrow}_{\mathcal{R}}$ equivalence classes and thereby serve as concrete representations of certain algorithms. And in so doing, he also proposes a substantial analysis of the relationship which must be borne between an implementation M and an algorithm A so that the former may be counted as an implementation of the latter. And third, using such abstract representatives of individual algorithms he proposes a uniform definition of computational complexity which serves to provide a mathematical foundation for the ascription of complexity properties to individual algorithms.

Moschovakis' proposal about algorithms is worked out over the course of a series of papers beginning with [99]. This paper outlines a broad programme for how various topics in classical recursion theory can be linked to more finitary concerns about individual procedures by using Moschovakis' [98] earlier theory of induction on abstract structures. This survey was followed by a series of more technical papers [99], [91] and [93], [94]. In [92], Moschovakis describes a formal term-based calculus of recursive definition known as *the formal language of recursion* [FLR] and provides an extensional form of operation (or

"intensional") semantics. Moschovakis has subsequently proposed that FLR may be used to provide a formal medium for reasoning about algorithms which would be informally specified using simultaneous recursion. As such, this language has technical affinities to both the theory of general recursive equations of Gödel [44] and Kleene [65], simple type theory in the sense of Church [20] and also the theory of recursion schemes discussed the previous chapter.[4]

Moschovakis has consistently motivated the recursor model as an attempt to provide a precise mathematical semantics for informal recursive definitions such as those considered Chapter 4. It is thus central to his proposal to view these definitions as being *interpreted* statements. In other words, unlike the recursion scheme model $\mathcal{E}$ discussed in Chapter 4.5 which seeks to abstract away from the mathematical interpretation of the terms appearing in such definitions, Moschovakis' proposal attempts to make precise the sense in which such definitions serve as implicit definitions of mathematical functions (in extension) like $plus^N(x, y)$, $fib^N(x)$ or $sort^A(x)$. The manner in which he seeks to accomplish this is best illustrated by examining an example.

To this end, consider again the primitive recursive definition (5.1). As I mentioned in passing in Chapter 4.5, the fact that there is a unique function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which may interpret the symbol $plus(x, y)$ in this definition follows from a theorem of elementary set theory which asserts that all relations defined by recursive definitions of the sort exemplified by (5.1) correspond to well defined (total) functions. In order to understand how such a definition determines a function, it is useful to think of (5.1) as

[4]In the sequel, I will for the most part adopt the style of Moschovakis' more recent papers and suppress detailed exposition of this system. It should be kept in mind, however, that not only can recursive definitions of the sort considered in Section 4.3 be formalized in FLR, but also many correctness and termination properties can be formally proven therein. Since Moschovakis' recursors correspond to the denotations of certain FLR expressions, this language thus provides a formal medium for simultaneously talking about executions, implementation and algorithms (via Moschovakis' definition in terms of recursors). It follows that if we accept this definition as an adequate analysis of the notion of algorithm, a theory formulated over FLR could form the basis of the kind of unified formal theory of procedures and computational reasoning T_p which was described in Chapter 3. I have previously argued that if such a theory be constructed, it ought to be the preferred vehicle for carrying out the uniform reinterpretation of procedural discourse which would ultimately be required to vindicate the algorithmic realist's claim that the validity of computational discourse can be accounted for in pure mathematical terms. Although Moschovakis does not himself recommend proceeding in this manner, one way of understanding the relationship between his programme and that of the algorithmic realist is to take FLR as a an explicitly worked out proposal for the form this theory should take.

implicitly defining a functional $Plus$ which operates on partial approximations p to $plus^N$. This functional has type $Plus : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N} \to \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$[5]and may be seen to take the following form

$$(5.1)\ \ Plus(p) = \{\langle n, m, q\rangle \ : \ (n = 0 \wedge n = q) \vee \exists u \exists v[p(n, u) = v \wedge m = Su \wedge q = Sv]\}$$

for $p \in \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$. This functional may be seen to have a (unique) least fixed point – i.e. a function $\boldsymbol{p} \in \mathbb{N} \rightharpoonup \mathbb{N} \to$ such that $Plus(\boldsymbol{p}) = \boldsymbol{p}$ and such that for all $\boldsymbol{q} \in \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$, if $Plus(\boldsymbol{q}) = \boldsymbol{q}$, then $\boldsymbol{p} \subseteq \boldsymbol{q}$. This is a special case of a general result known as the *Knaster-Tarski fixed point theorem* which states that every monotonic function on a complete partial order has a least fixed point.[6]An easy induction yields that not only is this $\boldsymbol{p}$ total, but that $\boldsymbol{p}(n, m) = plus^N(n, m)$ for all $n, m \in \mathbb{N}$.

A recursor is essentially a functional (like $Plus$) defined on an arbitrary partially ordered set (like $\langle \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}, \sqsubseteq\rangle$) together with a means of regarding certain co-ordinates of its domain and range as inputs and outputs. In full detail, Moschovakis's definition is as follows:

Definition 5.2.1. A *recursor* $\alpha : X \rightsquigarrow W$ on a poset X to a poset W is a triple $\langle D, \tau, value\rangle$ such that:

i) D is an inductive poset;

ii) τ is a monotonic functional on type $D \to D$ (the *transition mapping* of α);[7]

iii) $value : X \times D \to W$ is a monotonic partial functional (the *output mapping* of α).

By the Knaster-Tarksi theorem, there will always exist a least $q \in X \times D$ such that $\tau(q) = q$ – call this element $\mu q.\tau$. We may then define a notion of application for the recursor model by defining for each $x \in X$, $App_{rec}(\alpha, x) = value(x, \mu q.\tau)$. Taking the definition of recursor together with this definition of application, we have a model of computation $\mathfrak{R}$ in the sense

---

[5]I will use the notation $X \rightharpoonup Y$ to denote the set of all partial functions from domain X into domain Y. The functional $Plus$ is thus a map from the set of partial functions on pairs of natural numbers into natural numbers.

[6]In this case the relevant partial order order is the structure $N_0 = \langle \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}, \sqsubseteq\rangle$ where $p_1 \sqsubseteq p_2$ if and only if $p_1 \subseteq p_2$. A partial order (or *poset*) is *inductive* just in case every linearly ordered sequence of elements has a least upper bound. This is satisfied in the case of N_0 since if $p_0 \sqsubseteq p_1 \sqsubseteq \ldots$, then $supp_i = \bigcup_i p_i \in \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$. Finally, a functional $f : D \to D$ is *monotonic* just in case if for all $p, q \in D$, if $p \sqsubseteq q$, then $f(p) \sqsubseteq f(q)$.

of Chapter 5. As a concrete example, consider the recursor $\pi : \mathbb{N} \times \mathbb{N} \rightsquigarrow \mathbb{N}$ associated with the informal definition (5.1). π is given by $\langle \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}, Plus, \lambda p.\lambda x.\lambda y.p(x,y) \rangle$ from which it follows that $App_{rec}(\pi, \langle n, m \rangle) = \mu p.Plus(p)(n,m) = plus^N(n,m)$.

5.2.2 The Recursor Thesis

Recall that I originally characterized Moschovakis' view about the nature of algorithms as the claim that individual algorithms may be identified with equivalence classes of recursors. However, this is an oversimplification of Moschovakis' stated position in two respects. The first of these owes to a complication we have already encountered with respect to regarding a recursive model of computation such as $\mathcal{R}$. For note that the application function App_{rec} of this model is very much like that given for the model $\mathcal{PR}$ in Chapter 5.3. As is the case for $\mathcal{PR}$, the result of applying a recursor to a value is obtained by applying an (extensional) function which itself is obtained outside the model itself. For instance, in the case of π, the function $\boldsymbol{p} = \mu p.Plus(p)$ is defined in the Knaster-Tarksi theorem only as the infinite intersection $\boldsymbol{p} = \bigcap\{q : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N} \ : \ Plus(q) \subseteq q\}$. Such a definition is not only impredicative in that it quantifies over all partial functions on pairs of natural numbers, but it also non-constructive in the sense that provides no information on how to compute the values $\boldsymbol{p}(n,m)$ for particular $n, m \in \mathbb{N}$. On this basis, one might thus look on $\mathcal{R}$ as a model of computation only in the degenerate sense discussed in Chapter 5.5 in that the definition of App_{rec} does not specify how the application of a recursor to an argument corresponds to a calculation leading to its value. And thus as we saw was also the case with the model $\mathcal{PR}$, it may initially appear that we cannot hope to define the complexity theoretic properties of algorithms in terms of recursors.

But it is also possible to specify a sort of operational (or, to use Moschovakis' term *intensional*) semantics for recursors which makes explicit the computational features which are latent in certain fixed point definitions. For note that under certain conditions, a function determined as the fixed point of a functional can be approximated constructively "from

[7]This is a slight simplification of Moschovakis' definition of transition mapping. The original definition requires that τ be a map of type $X \times D \to D$ which allows parameterization relative to an input. Since this added degree of flexibility plays little role in the mathematical development of the theory of recursors, I have suppressed it here.

below" – i.e. as the limit of a constructively generated sequence of partial approximations . For instance, if we let $p_0 = \emptyset$, and $p^{i+1} = Plus(p_i)$, then the function $\boldsymbol{p} = plus^N$ will be approximated by the sequence of partial functions of type $\mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$

$$
\begin{aligned}
(5.2)\ \ & p^0 := \emptyset = plus^N \upharpoonright [0] \\
& p^1 := Plus(p^0) = \{\langle n, 0, n\rangle \ : \ n \in \mathbb{N}\} = plus^N \upharpoonright [1] \\
& p^2 := Plus(p^1)^{\mathbb{N}} = plus_1 \cup \{\langle n, 1, n+1\rangle \ : \ n \in \mathbb{N}\} = plus^N \upharpoonright [2] \\
& p^3 := Plus(p_2)^{\mathbb{N}} = p^2 \cup \{\langle n, 1, n+2\rangle \ : \ n \in \mathbb{N}\} = plus^N \upharpoonright [3] \\
& \quad \vdots
\end{aligned}
$$

where $plus^N \upharpoonright [i] = plus^N \cap \{\langle n, m, q\rangle \ : \ m < i\}$. It follows that every pair of natural numbers $\langle n, m\rangle$ will be in the domain of p^{m+1}. And from this it follows that we may calculate the value of $\boldsymbol{p}(n, m)$ as follows: 1) construct p^{m+1} by iterating $Plus$ on $\emptyset$ $m + 1$ times; and 2) search for and return the unique q such that $\langle n, m, q\rangle \in p^{m+1}$.

Inasmuch as we are also justified in looking at a functional like (5.1) as specifying a method for constructing the p^{i+1} from p^i, a definition like (5.1) may be viewed as giving rise to a quasi-constructive method for computing the values of its fixed point.[8] On the basis of this observation, Moschovakis introduces the following definitions pertaining to recursors:

Definition 5.2.2. Let $\alpha = \langle D, \tau, value\rangle : X \rightsquigarrow W$ be a recursor and $\boldsymbol{d} = \mu p.\tau(p)$. We associate with each ordinal ξ the partial function $d^\xi_\alpha = \tau(\sup\{d^\eta_\alpha \ : \ \eta < \xi\})$ which is known as the *ξth stage of $\boldsymbol{d}$*.[9] We additionally define the *closure ordinal of* α (denoted $|\alpha|$) to be the least ξ such that $\forall x \in X[d^\xi_\alpha(x) = \sup\{d^\eta_\alpha \ : \ \eta < \xi\}]$. And finally with each $x \in X$, we associate a *stage assignment* $stage_\alpha(x)$ defined as the least η such that $value(d^\eta_\alpha(x)) \in W$.

Moschovakis argues that genuine computational significance attaches to each of these notions. Note first that the stages d^ξ_α correspond to approximations to $\boldsymbol{d}$ in the sense

[8]The "quasi-" qualification is necessary for two reasons. First note that while the described procedure allows us to effectively determine the values of $p^i(n, m)$ pointwise, the mapping $p^i \mapsto p_{i+1}$ is itself infinitary in the sense that for $i > 0$, the p^i consists of an infinite set. Also note that while in the case under consideration, the "approximations" $p^0 \sqsubseteq p^1 \sqsubseteq \ldots$ do converge to $\boldsymbol{p}$ in the sense that $\bigcup_{i\in\omega} p^i = \boldsymbol{p}$, it is not alway the case that an arbitrary monotonic functional F is such that $\mu p.F(p) = \bigcup_{i\in\omega} F^i(\emptyset)$. In order for this to be true, we must additionally required that be F *continuous* – i.e. $F(p)(\vec{x}) = y$ then there exists $p' \sqsubseteq p$ such that $dom(p')$ is finite and $F(p')(\vec{x}) = y$.

[9]In this definition, suprema are to be taken with respect to the ordering $\sqsubseteq$ defined on D, subject to the convention that $\sup(\emptyset) = \bot$, the everywhere undefined partial function.

considered above. On this basis Moschovakis suggests that they can be likened to steps in the computation of $\boldsymbol{d}$. The closure ordinal of a recursor corresponds to the number of times τ must be iterated on $\perp$ in order to obtain a fixed point. As recursors which are such that $|\alpha| \leq \omega$ will correspond to finitary algorithms in the sense that for every input $x \in X$, either $d_\alpha^\xi(x)$ will be defined for some finite ξ or $\boldsymbol{d}(x)$ will be undefined.[10] And finally Moschovakis also suggests that for each $x \in X$, the ordinal corresponding to $stage_\alpha(x)$ can be taken as a measure of how many steps the procedure described above must be iterated until the approximation d_α^ξ becomes defined on x. And as we will see below, this will be the basis for his proposed analysis of complexity theoretic properties of algorithms in general.

Despite the fact that the official definition App_{rec} of application on $\mathcal{R}$ is defined extensionally, the definitions just considered show how a computational interpretation may be given of what it means to "execute" a recursor. The availability of the foregoing definition thus lends credece to the proposal that the general notion of algorithm in which we are interested may be analyzed in terms of recursors. But as I noted above, Moschovakis's views on how this should be accomplished are somewhat complex. For on the one hand he does not directly claim that algorithms are identical to recursors, but rather that the former are, as he puts it, "faithfully modeled" by the latter. And on the other, rather than claiming that every algorithm is modeled by a *unique* recursor, his most careful statements of his position contain only the claim that there is *some* recursor with this property.

His motivation for adopting the latter position is partially explained by the fact that there is a natural notion of computational equivalence for recursors which is given as follows:

Definition 5.2.3. Let $\alpha_1 = \langle D_1, \tau_1, value_1 \rangle$ and $\alpha_2 = \langle D_2, \tau_2, value_2 \rangle$ be recursors of type $X \rightsquigarrow W$. α_1 and α_2 are *isomorphic* is there exists an order-preserving bijection $\rho : D_1 \to D_2$ such that for all $d \in D$ and $x \in X$ the following hold:

i) $\rho(\tau_1(d)) = \tau_2(\rho(d))$;

[10] There are well known examples of mathematical procedures which lack this property. For instance, as Moschovakis points out, it is possible to formalize many cut elimination procedures in proof theory as recursors. On this basis one may, for instance, show that the recursor corresponding to the traditional Gentzen cut elimination procedure for first-order logic has closure ordinal ω^ω. However, the recursor corresponding to the Gentzen cut eliminiation for first-order arithmetic has closure ordinal ϵ_0. Part of the motivation for Moschovakis' programme is to provide a setting which allow us to view both of these constructions as genuine procedures despite the infinitary nature of the latter.

ii) $value_1(x, d) = value_2(x, \rho(d))$

I will write $\alpha_1 \underline{\leftrightarrow}_\rho \alpha_2$ to denote the fact that α_1 and α_2 are isomorphic via ρ and $\alpha_1 \underline{\leftrightarrow}_{\mathcal{R}} \alpha_2$ to denote the fact that such a mapping exists.

Very roughly, the recursors α_1 and α_2 are isomorphic just in case every application of τ_1 to D_1 makes a structurally analogous contribution to the computation of $\mu q.\tau_1(q)$ (in the sense of the ordering on X) as does the application of τ_2 to D to the computation of $\mu q.\tau_2(q)$.

With this definition in place, I can finally present Moschovakis' most careful statement of his view about the relationship between algorithms and recursors:

> **Proposal: Algorithms are recursors.** The mathematical structure of every algorithm on a poset X to a set W is modeled faithfully by some recursor $\alpha : X \rightsquigarrow W$; and two recursors model the same algorithm if they are isomorphic. ([94], p. 18)

This and other similar passages in [97] suggest that Moschovakis's position is that algorithms correspond to equivalence classes of recursors under recursor isomorphism. I will subsequently refer to this claim as the *Recursor Thesis* [RT].

In the framework of chapter 3, it may seem appropriate to characterize RT as equivalent to a form of abstractionism about algorithms wherein the class of implementations $\mathcal{M}$ is given by the class of recursors $\mathcal{R}$ and the notion of computational equivalence $\underline{\leftrightarrow}$ is given by $\underline{\leftrightarrow}_{\mathcal{R}}$. But there are two obstacles standing in the way of Moschovakis's view quite straightforwardly, both of which are already evident from the manner in which he appears to equivocate in stating his RT. The first concerns the distinction which Moschovakis appears to draw between the "mathematical structure" of an algorithm and the algorithm itself. The only use he makes of this terminology, however, is to highlight the familiar point that he is concerned with describing structural features which are, in a sense I will consider in a moment, intrinsic to algorithms and not to their implementations. And for this reason it is reasonable to understand Moschovakis as equating what he refers to as "the mathematical structure of an algorithm" with the notion of an abstract, machine and language independent procedure which I have taken to characterize the notion of algorithm.

The other respect in which Moschovakis may appear to be be hedging his bets in RT concerns the nature of the relationship which he is claiming to exist between algorithms

and recursors. For rather than claiming that algorithms are *identical* to recursors, he rather claims that the former are "faithfully modeled" by the latter. But the force of this apparent equivocation is again diminished by what Moschovakis has to say about what it means for one sort of mathematical structure to faithfully model another. His paradigm example of this relationship is that borne by an arbitrary Peano system $\mathfrak{M} = \langle M, 0^M, s^M \rangle$ – i.e. a structure satisfying the second-order Peano axioms $PA2$ – to the actual natural number structure $\mathfrak{N} = \langle \mathbb{N}, 0^{\mathbb{N}}, s^{\mathbb{N}} \rangle$. Moschovakis describes this situation as one in which structures of the former sort "faithfully models 'the natural numbers' ... up to first-order isomorphism."

Note, however, that $\mathfrak{M}$ shares this property with a proper class of formally distinct structures. And for this reason, the view about the ontological relationship between $\mathfrak{F}$ and $\mathfrak{N}$ which Moschovakis ultimately adopts is similar to the so-called *structuralist* philosophy of mathematics anticipated by Benacerraf [7] and favored by (among others) Parsons [105] and Shapiro [127]. According to this view, the elements of $\mathfrak{N}$ are not held to be identical to the members of any particular set theoretic structure but are rather to be equated with "positions" within any structure $\mathfrak{M}$ satisfying $PA2$. Note for instance that the structures $\mathfrak{M}_1 = \langle \omega_1, 0_1, s_1 \rangle$ (where ω_1 is the finite von Neumann ordinals, 0_1 its least element and s_1 its ordinal successor operation) and $\mathfrak{M}_2 = \langle \omega_2, 0_2, s_2 \rangle$ (where ω_2 is the finite Zermelo ordinals, 0_2 its least element and s_2 its ordinal successor operation) both satisfy the Peano axioms. On this basis we may define so-called *reduction maps* $r_1 : \mathbb{N} \to \omega_1$ by $r_1(0) = 0_1$, $r_1(s(n)) = s_1(r_1(n))$ and $r_2 : \mathbb{N} \to \omega_2$ by $r_1(0) = 0_2$, $r_2(s(n)) = s_2(r_2(n))$. But the structuralists maintain that if we were to take the view that say r_1 gave rise to genuine *identities* between numbers and sets, then we would be hard put to explain by this should not also be the case for r_2. And famously it cannot be the case *both* that $2 = r_1(2) = \{\emptyset, \{\emptyset\}\}$ and $2 = r_2(2) = \{\{\emptyset\}\}$.

From this Moschovakis concludes that the goal of providing a set theoretic definition of a mathematical concept C (his two examples are that of *natural number* and *order type*) is not "to tell us in ... metaphysical terms term *what* the [objects falling under C] *are*, but to identify and delineate their fundamental mathematical properties" ([94], p. 6, his

italics). He then states that this may be accomplished by providing

> [A] class of sets M_C together with a equivalence relations $\sim_C$ on M_C, with the intention that each $\alpha \in M_C$ *faithfully represents* (codes) some C-object α_C, and that two members $\alpha, \beta \in M_C$ code the same C-object exactly when $\alpha \sim_C \beta$. A modeling of this type is successful if the $\sim_C$-invariant properties of the members of M_C capture exactly the fundamental properties of the C-objects – which implies that every fundamental property of a C-object can be "read off" any of its codes. [94], p. 6

Moschovakis does not go into detail about how this analysis might be carried out with respect to any concept natural number. But on the basis of his prior comments, it is easy to envision how this might be handled. In particular, we may take M_{natnum} to be the disjoint union of the domains $M_1, M_2, \ldots$ of an indexed family of Peano systems $\mathfrak{M}_1, \mathfrak{M}_2, \ldots$. And we may additionally define $\sim_{natnum}$ to be the relation $O(x, y)$ which holds between $x \in M_i$ and $y \in M_j$ such in case there exists an isomorphism $g : \mathfrak{M}_i \to \mathfrak{M}_j$ and $g(x) = y$.[11]

Since Moschovakis explicitly states that he intends to use the foregoing analysis as a foundational framework for analyzing the notion of algorithm, it is in this way that we ought to understand the use of the expression "faithfully models" in RT. This means that in order to view this proposal as successful, the class of $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant properties of recursors must *exactly correspond* to the fundamental properties of algorithms. For as we will see in a moment, Moschovakis is primarily interested in using RT to justify the application of complexity theoretic properties to individual algorithms (as opposed to their implementations). But at the same time, Moschovakis does not start out by presenting an axiomatic description of what he takes the fundamental properties of algorithms to be of the sort which would be provided by theory T_p discussed in Chapter 2. And thus since there is no clear candidate for such a theory, there also appears to be no way in which we

[11] This analysis extends Moschovakis' comment that Peano systems faithfully model the natural numbers "up to first-order isomorphism." Note that since the ordering relation $x < y$ on natural numbers is definable in the language of $PA2$ (i.e. as $(x < y) \iff \exists S[S(x) \wedge S(y) \wedge \neg\exists z[S(z) \wedge s(z) = x] \wedge \forall z[S(z) \to S(s(z))]]$), it may readily be seen that this analysis corresponds to treating natural numbers as finite order types (since we will $O(x, y)$ if and only if the relations $(<\restriction x)^{M_i}$ and $(<\restriction y)^{M_i}$ are order isomorphic). This analysis is again equivalent with the typical structuralist credo that natural numbers are "positions" in ω-sequences. However Moschovakis also comments that the set $\{\emptyset, \{\emptyset\}\}$ "faithfully models 2 up to *equinumerosity*". On this basis, one could also use Moschovakis' framework to develop an analysis of natural number similar to that of the neo-Fregeans by taking M_{natnum} to a structure like $R(\omega)$ (the hereditarily finite sets) and $\sim_{natnum}$ to be the relation $\equiv$ of equinumerosity. Since Moschovakis is already committed to an ontology of sets (which neo-Fregeans like Wright [154] presumably are not), he could simply take the natural number n to correspond to the equivalence class $[n] = \{y \in R(\omega) \ : \ y \sim \alpha_n\} \in R(\omega + 1)$ where α_n is some canonical n-member set such as the n^{th} von Neumann ordinal.

can currently hope to *prove* that RT is correct in the same way that, say, the theory $PA2$ allows us to prove a theorem stating that the properties of equivalence classes of finite sets under equinumerosity are exactly those expressible in second-order arithmetic.

But although our current grasp on the properties of individual algorithms may not allow us to demonstrate RT definitively, it is proportionally easier to show how it might fail. For note that according to the passage cited above, the claim that the pair $\langle \mathcal{R}, \underline{\leftrightarrow}_{\mathcal{R}} \rangle$ succeeds in providing a faithful modeling of the concept *algorithm* is equivalent to the claim that the set Γ of fundamental properties of algorithms is coextensive with the set Δ of $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant properties of $\mathcal{R}$.[12]Of course, given we lack an axiomatic theory of algorithms, it may be that Γ is not determinately defined. But at the same time, it might also be that our current practices are sufficiently well developed that we can determinately say of *particular* computational properties Φ, Ψ that they either belong to Γ or fail to belong to Γ. And if this is the case, to *disprove* RT it suffices to demonstrate either of the following claims:

(5.3) i) there is a "fundamental" property $\Phi \in \Gamma$ which is *not* invariant under $\underline{\leftrightarrow}_{\mathcal{R}}$ (i.e. $\Phi \notin \Delta$);

ii) there is a property $\Psi \in \Delta$ which *is* invariant under $\underline{\leftrightarrow}_{\mathcal{R}}$ but is not "fundamental" (i.e. $\Psi \notin \Gamma$).

---

[11]In fact, Moschovakis explicitly forswears the possibility of developing such a theory essentially for the same sort of reasons I discussed in connection with the development of T_p in chapter 3: "The trouble [is that such a theory would be] too complex: There are too many notions competing for primitive status (algorithms, implementations and computations, at the least) and the relations between them do not appear to easily expressible in first-order terms. I doubt that this project can be carried through, and, in any case, there are no proposal on the table suggesting how we might get started." [94], p. 9

[12]One might think that the requirement that $\Gamma = \Delta$ is stronger than needed to demonstrate the sort of ontological conclusion which is apparently announced in Moschovakis's Proposal. In fact the weaker requirement that $\Delta \subseteq \Gamma$ would seem to be a sensible way of disambiguating the relevant notion of "faithful representation" in play (in this regard, cf. work in formal measurement theory such as [140]) were it not for the explicit discussion of this notion cited above. The fact that coextensiveness really is what is required from the standpoint of algorithmic realism follows from the fact that the opposite inclusion $\Gamma \subseteq \Delta$ is needed to show that the analysis accounts for the status of artifactual properties which may be possessed by certain implementations M of an algorithm A but which we would not generally say are possessed by A itself. Note that the analogous neo-Fregean treatment of natural numbers arguably achieves this result. For instance, the set $2_1 = \{\emptyset, \{\emptyset\}\}$ which represents the natural number 2 under the reduction mapping r_1 discussed above possesses the property of being a transitive set. But note that we would typically say that the natural number 2 itself lacked this property (in fact on the informal understanding of natural number, it makes no sense to even say that 2 has members). However, transitivity is clearly not preserved under the equivalence relation $\equiv$ of equinumerosity – for instance $2_1 \equiv \{\emptyset, \{\{\emptyset\}\}\}$, but the latter set is clearly not transitive. From this it follows that transitivity cannot be "read off" as a fundamental property of natural numbers. And it

I believe Moschovakis's proposal fails on both counts. The fact that i) and ii) hold (and thus that both the inclusions $\Gamma \subseteq \Delta$ and $\Delta \subseteq \Gamma$ fail) may ultimately be seen to follow from a familiar problem pertaining to RT – i.e. that the equivalence $\underline{\leftrightarrow}_{\mathcal{R}}$ turns out to be too fine to adequately track our intuitions about algorithmic identity. This has two consequences. First, it means that it is possible that there exist properties Ψ which *are* invariant under $\underline{\leftrightarrow}_{\mathcal{R}}$ but which are not possessed by all recursors which we would intuitively accept as representing the same algorithm. And second, it means that there may exist pairs of recursors α_1 and α_2 which we would intuitively accept as representing the same algorithm but which are not linked by $\underline{\leftrightarrow}_{\mathcal{R}}$.

In order to demonstrate that the first of these claims (which corresponds to a proof of (5.3ii)), we will need to further investigate Moschovakis' analysis of complexity theoretic properties. I will do this over the course of the remainder of this section. In order to demonstrate the second claim (which corresponds to a proof of (5.3i)), we will need to further investigate Moschovakis' analysis of the relation which holds between register-based and recursion-based implementations of the same algorithm. This topic is most naturally understood in light of an example I will consider with respect to (5.3ii). And for this reason, I will postpone a discussion of (5.3i) until Section 2.4.

5.2.3 Recursors and computational complexity

As mentioned, one of Moschovakis' primary motivations in framing the RT is to provide a principled means of making informal proofs about correctness and running time complexity mathematically rigorous. In order to so, he proposes that talk of informal computational notions like "execution", "computational step" and "numbers of comparisons" be replaced with talk about formal properties of recursors. Under such a systematic reinterpretation of computational discourse, informal arguments given in algorithmic analysis of the sort discussed in Chapter 2 could be replaced with standard mathematical proofs about recursors and their properties. And thus under the assumption that recursors do in fact "faithfully model" informal algorithms, this would enable us to conclude not only that these results

---

is for this reason that although 2_1 may not be identified with 2, the class ω_1 of which this set is a member may be used to faithfully model the natural numbers with respect to $\equiv$ in Moschovakis' sense.

are valid, but also that they report on the properties of genuine mathematical objects.

The fact that Moschovakis takes this to be worth doing is indicative of the fact that he must accept that at least certain complexity theoretic properties are contained in the set Γ of "fundamental" computational properties which can be attributed directly to algorithms. As we have seen, such properties are standardly taken to be those expressed by predicates of the form "algorithm A has running time complexity $time_A(n)$" defined relative to an appropriate size metric $|\cdot| : X \to \mathbb{N}$ on the domain X of A. And for this reason, it seems that Moschovakis must accept that at least these properties are contained in Γ.

In order to see how this observation bears on the status of (5.3i), we must next inquire into how Moschovakis intends to formalize properties of this sort so that it makes sense to apply them to recursors. For note that while relative to the practice of the analysis of algorithms it makes perfect sense to say something like "MERGESORT has running time $O(n \log_2(n))$", it is not immediately clear what it means to predicate such a property of a recursor. In order to do so, a mathematical analysis of running time complexity properties must be presented so that the informal complexity theoretic property $\Phi \in \Gamma$ is correlated with a structural property Φ^* of recursors in the manner discussed in Chapter 4.1.

And as I mentioned in the introduction to this chapter, Moschovakis is unique among (the admittedly small) class of theorists who explicitly endorse algorithmic realism in that not only does he acknowledge that it is incumbent upon a realist to provide such an analysis, but actually seeks to present one himself. This analysis is notable in that it attempts to identify the complexity features with structural properties of recursors which do not correspond in any obvious or direct sense with temporally interpretable features of other models of computation we have thus far examined. However, this analysis rests on a variety of non-trivial assumptions about how algorithms may be specified as recursors. Among these are the following:

(5.4) i) An algorithm A may only specified directly as a recursor α_A. It may, of course, also be possible to refer to A by using some other sort of implementation M such as a flowchart machine or Turing machine. However, the fact that M implements A is to be explained in terms of the fact that α_A may be *reduced* to M in the technical sense of recursor reduction which will be discussed in Section

2.4. Thus although, per RT recursors must themselves be counted as implementations, Moschovakis holds that they are more abstract than other forms of transition- or register-based implementations.

ii) In order to analyze the complexity of an algorithm A, we must assume not only that it has been presented as a recursor $\alpha_A : X \rightsquigarrow W$ where $\alpha = (D_A, \tau_A, value_A)$ but also that α_A is itself specified as the denotation of an expression of the form $\rho_A(\vec{x}) = T_0(x_1, \ldots, x_n)$ where $\{T_1(x_1, \ldots, x_n) = F_1(x_1, \ldots, x_n), \ldots, T_n(x_1, \ldots, x_n) = F_n(x_1, \ldots, x_n)\}$. In [94], [96] and [97] such expressions are taken to correspond to terms over the Formal Language of Recursion. In this setting, τ_A will correspond to the so-called *intension* associated with $\rho_A(\vec{x})$ relative to the provided operational semantics provided in [92]. For present purposes, however, $\rho_A(x)$ may be understood as systems of mutually recursive equations which differ only notationally from the recursion schemes introduced in section 5.5. (In particular, each such term ρ will correspond to a scheme $\hat{E}_\rho$ with head F_0 and body $F_1, \ldots, F_n$.)

iii) The running time complexity of an algorithm A on an input $x \in X$ is defined only relative to prior decisions about how to measure the size of x and also about which of the mathematical operations which figure in its specification ought to be counted as contributing to its complexity. The former parameter may be formalized in the usual manner by giving a size metric $|\cdot| : X \to \mathbb{N}$. The latter parameter can be formalized by one of the terms F_i $(1 \leq i \leq n)$ which appears in the term ρ_A and which I will refer to as a *complexity index* for ρ_A. A definition of *recursor-based running time complexity* for α_A relative to $|\cdot|$ and F_i will then be given by a function $tr^{f_i}(|x|)_{\alpha_A} : X \to \mathbb{N}$ which maps $|x|$ to the number of times f_i (the denotation of F_i) must be applied in order to determine the value $\alpha_A(x)$.

Although Moschovakis offers only cursory arguments in favor of these assumptions, they form the basis of his definition of what it means for a recursor $\alpha_A(x)$ to have complexity $tr^{f_i}_{\alpha_A}(|x|)$. Stating this definition in full generality would, however, require a substantial digression into the semantics of the FLR. But it turns out that it is assumptions i), ii) and

iii) themselves rather than the exact definition of $tr^{f_i}_{\alpha_A}(|x|)$ which bear more directly on the status of RT. Rather than provide a complete statement of the definition of this function, I will instead illustrate its application together with the role of these assumptions via an example.

To this end, consider again the algorithm MERGESORT. We have seen that this algorithm is paradigmatically specified by an informal recursive definition. In particular, suppose we are interested in sorting lists of items over a set L linearly ordered by $\preceq$ which, for reasons which will become apparent in a moment, we will take to be represented by its characteristic function $c = \chi_{\preceq}$ which will be treated as a explicit argument to the algorithm. Then if $u \in L^*$, MERGESORT may be expressed by the following pair of mutually recursive equations

(5.5) i) $$sort(c)(u) = \begin{cases} u & \text{if } |u| \leq 1 \\ merge(sort(c)(firsthalf(u)), sort(c)(secondhalf(u))) & \text{else} \end{cases}$$

ii) $$merge(c)(v,w) = \begin{cases} w & \text{if } v = \epsilon \\ v & \text{else if } y = \epsilon \\ head(v) \cdot merge(c)(tail(v), w)) & \text{else if } c(head(v), head(w)) = \top \\ head(w) \cdot merge(c)(v, tail(w)) & \text{else if } c(head(v), head(w)) = \bot \end{cases}$$

Here $firsthalf, secondhalf, head,\ tail,$ and $\cdot$ have their standard meaning as defined in Chapter 4.5.

If we now let h_1, h_2, h, t and a be formal functional symbols corresponding to these functions, then an FLR term $\rho_{\text{MERGESORT}}(u)$ representing this algorithm may be given as follows:

$$\begin{aligned} \rho_{\text{MERGESORT}}(u) \quad = \quad & s_0(u, s, m, c) \text{ where } \{ \\ & s(u) = \text{ if } |u| \leq 1 \text{ then } u \text{ else } m(s(h_1(u)), s(h_2(u))), \\ & m(v,w) = \text{ if } v = \epsilon \text{ then } w \\ & \qquad\qquad \text{if } w = \epsilon \text{ then } v \\ & \qquad\qquad \text{if } c(h(v), h(w)) \text{ then } a(h(v), m(t(v), w)) \\ & \qquad\qquad\qquad\qquad \text{else } a(h(w), m(v, t(w)))\}^{13} \end{aligned} \tag{5.6}$$

Relative to the so-called *intensional* semantics for FLR given in [92], (5.7) determines as its extension a recursor $\alpha_{\text{MERGESORT}} : L^* \to L^*$ of type $(D, \tau, value)$ where

(5.7) i) $D = (L^* \rightharpoonup L^*) \times (L^* \times L^* \rightharpoonup L^*) \times (L^* \times L^* \rightharpoonup \{\top, \bot\})$

ii) $\tau : X \times D \to D$

iii) $value(u, s, m, c) = s(u)$

In this case τ takes as input a list $u \in L^*$ and *partial* functions $\boldsymbol{s}', \boldsymbol{m}', \boldsymbol{c}'$ which respectively correspond to initial segments of the functions $\boldsymbol{s}$ (the denotation of the $sort(u)$ in (5.5)), $\boldsymbol{m}$ (the denotation of $merge(v, w)$ in (5.5)) and $\boldsymbol{c}$ (the denotation of the characteristic function of $\preceq$ in (5.5)) which are determined as the mutual fixedpoints in (5.7).

Suppose we now make the assumptions that the size of a list $u \in L^*$ is to be measured by its length $|n|$ and that the appropriate complexity index for MERGESORT is the comparison function $\boldsymbol{c}$. Note that these formalize the conventional choices for sorting algorithms to measure the size of their input in terms of the length of the list to be sorted and the amount of computational "labor" to return an output in terms of the number of element-to-element comparisons required. On the basis of these assumptions, the complexity of MERGESORT would be analyzed by Moschovakis in terms of $\alpha_{\text{MERGESORT}}$ as a function between $|u|$ and the size of a domain of the smallest partial function $\boldsymbol{c}' \subseteq \boldsymbol{c}$ which is required so that the fixed points $\boldsymbol{m}' \subseteq \boldsymbol{m}$ and $\boldsymbol{s}' \subseteq \boldsymbol{s}$ determined as the solution to the equation $\tau(u, \boldsymbol{s}', \boldsymbol{m}', \boldsymbol{c}') = \langle \boldsymbol{s}', \boldsymbol{m}', \boldsymbol{c}' \rangle$ are such that $\boldsymbol{s}'(u)$ is defined and correct. In particular, the following result may be obtained by reasoning directly about the recursor $\alpha_{\text{MERGESORT}}$:

Theorem 2. Let $\alpha_{\text{MERGESORT}}$, $\boldsymbol{s}, \boldsymbol{m}$ and $\boldsymbol{c}$ be as above, $u \in L^*$, $|u| = n$ and $v \in L^*$ be such that $value(u) = \boldsymbol{s}(u) = v$ (i.e. v is the sorted version of u). Then there exists $\boldsymbol{c}' \subseteq \boldsymbol{c}$, $\boldsymbol{s}' \subseteq \boldsymbol{s}$ and $\boldsymbol{m}' \subseteq \boldsymbol{m}$ such that $\tau(u, \boldsymbol{s}', \boldsymbol{m}', \boldsymbol{c}') = \langle \boldsymbol{s}', \boldsymbol{m}', \boldsymbol{c}' \rangle$, $|dom(\boldsymbol{c}')| \leq n\lceil \log_2(n) \rceil$ and $\boldsymbol{s}'(u) = v$.[14]

[13] It should be noted that there is generally *not* a one-one relationship between informal recursive definitions like (5.5) and FLR terms which might be taken to represent them. In [92], however, a reduction calculus on such terms is defined and a normal form theorem is proven whereby a term like $\rho_{\text{MERGESORT}}$ (which is not in normal form in virtue of the fact that it contains functions expressions with more than one level of nesting) may be reduced to a canonical form terms in a denotation-preserving manner. It is such a normal form $\rho'_{\text{MERGESORT}}$ which Moschovakis would presumably take to be the "official" FLR representation of MERGESORT.

This statement describes the relationship between the length $|u|$ of an input u to $\alpha_{\text{MERGESORT}}$ and the size $tr(|u|)$ of the smallest approximation to the comparison relation $\preceq$ (as represented by its characteristic function c) which needs to be passed as an argument to the system of fixed point equations determined by (5.7) so that an approximation $\boldsymbol{s}'$ to the sorting function thereby determined is guaranteed to be correctly defined on u. In particular, it is immediate from the theorem that we should define the recursor-based complexity $\alpha_{\text{MERGESORT}}$ relative to c as $tr^c(|u|) = |u|\lceil\log_2(|u|)\rceil$.

As noted above, Moschovakis's primary motivation for developing the analysis on which this theorem is based is to provide a means of reframing complexity theoretic analyses derived using the informal methods from the algorithmic analysis as purely mathematical results about recursors. Relative to the assumptions that recursors may be used to reason about algorithms in the manner suggested by RT, such results may be taken as demonstrating the use of informal procedural reasoning (in particular, reasoning that employs the informal notion of a "computational stage" or what it means for one stage to occur before or after another in the execution of an algorithm) may be eliminated in favor of purely structural reasoning about recursors. It is thus reasonable to think of Theorem 1 as having the same significance as the corresponding informal complexity analysis of MERGESORT given in Chapter 2.

These considerations suggest that Moschovakis has indeed succeeded in giving a mathematical analysis of complexity theoretic properties which comports well with our informal understanding of what it means to assign a running time to an individual algorithm. But although we have seen that this is one of the responsibilities incumbent on an algorithmic realist, having provided such an analysis, we can now ask which complexity theoretic properties are invariant under $\underline{\leftrightarrow}_{\mathcal{R}}$ and thus, per RT, correspond to fundamental properties of individual algorithms. And in this regard, it may be noted that the following observation follows directly from the definitions of recursor isomorphism and recursor-based complexity:

(5.8) **Corollary** Let $\rho_1(x)$ and $\rho_2(x)$ be FLR terms both containing the term F_i and specifying isomorphic recursors $\alpha_1 : X \rightsquigarrow W$ and $\alpha_2 : X \rightsquigarrow W$. Then if $|\cdot| : X \to \mathbb{N}$ is a size metric and f_i a complexity index corresponding to the interpretation of F_i, then $tr^{f_i}_{\alpha_1}(|x|) = tr^{f_i}_{\alpha_2}(|x|)$.

According to this observation, recursor-based complexity is a $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant property of recursors. Given that an isomorphism π relating α_1 and α_2 relates their transition functions in a "step-by-step" manner (i.e. such that $\pi(\tau_1(x,d)) = \tau_2(x,\pi(d))$ for all $x \in X$, $d \in D$), this result is not surprising. But what is less clear is whether it should be regarded as a positive or a negative result with respect to the status of RT.

Two important themes which I have repeatedly argued are central to algorithmic realism in previous chapters are as follows: i) complexity theoretic properties *are* fundamental properties of individual algorithms; and ii) these properties must be individuated finely enough to distinguish between pairs of algorithms between whose efficiency we distinguish in practice (cf. the discussion of NaiveGCD versus Euclid in chapter 1 and Insertionsort and Mergesort in chapter 2). There are, however, a variety of problems which complicate how complexity properties can legitimately be attributed to individual algorithms in a mathematically robust manner. Not the least of these is the dependence of complexity classifications on the choice of which mathematical operations are to be counted as contributing to a procedure's complexity. However we have seen that Moschovakis's account has a means of accounting for this relativity in that the definition of recursor-based complexity is defined relative to a choice of complexity index.

There is, however, another subtlety involved with attributing complexity theoretic properties to individual algorithms. Suppose, for instance, that we are considering a fixed algorithm A and a set of operations $f_1, \ldots, f_n$ in terms of which the complexity of A will be measured. But suppose that have also constructed two different implementations M_1 and M_2 for which we are equally willing to accept $imp(M_1) = A$ and $imp(M_2) = A$. If we wish to calculate the complexity of A in a mathematically rigorous manner in the way both algorithmic realism and Moschovakis demand, then it seems that we have no choice but to reason about M_1 or M_2 directly. But this raises the possibility that the complexities which we calculate as $time_{M_1}(|x|)$ and $time_{M_2}(|x|)$ will not differ even if these functions are defined so as to only count the number of applications of $f_1, \ldots, f_n$. And we thus must face the question of how a running complexity can be assigned to A given that there need not be agreement among the complexities associated with its implementations.

It does not take much exploration to convince oneself that this situation is in fact that

the rule rather than the exception. Some evidence for this is provided by considering again the example of Chapter 3, for instance, where I argued that we have $imp(S_1) = \text{PAL1}$ and $imp(S_5) = \text{PAL1}$ and $time_{S_1}(|w|) = (n^2 + 5n + 6)/2 - 2$ and $time_{S_5}(|w|) = (3n^2 + 6n)/2 + 1$. Of course, per Chapter 4, the data that S_1 and S_5 are indeed implementations of PAL1 can be challenged at least in part on the basis of their quadratic complexity. However, it is easy to construct other examples with demonstrate the same point – e.g. two implementations M_1, M_2 of INSERTIONSORT as RAM machines which respectively have running times $time_{M_1}(|u|) = 3n^2 + 3$ and $time_{M_2}(|u|) = n^2 + 4n + 4$.

The existence of such disparities in the running time of different implementations of the same algorithm is a well known phenomena in the analysis of algorithms. And as I have intimated several times previously, it is in fact the basis for a widely accepted informal thesis about how running times should be attributed to algorithms which I will refer to as the Asymptotic Running Time Thesis [ARTT]. This thesis is motivated by the view that it makes no sense to attribute an *exact* running time $time_A(|x|)$ to an individual algorithm A because however we choose to do so, it will almost certainly be possible to construct an intuitively acceptable implementation M of A such that $time_M(|x) \neq time_A(|x|)$. However, continues the proponent of ARTT, we certainly do think of individual algorithms as having running time complexity as confirmed, for instance, by the fact that we routinely compare the running times of algorithms A_1 and A_2 which compute the same function in order to decide which will be more efficient to employ in a given situation.

In order to ground this practice, the proponent of ARTT proposes that what may be properly attributed to individual algorithms is not exact running time measure, but rather asymptotic running time as defined in Chapter 4.2. This is to say that if we calculate that the running time complexity of A is $f(n)$ either by reasoning informally about A or by reasoning formally about one of its representations, then the property which we may justifiably attribute to A is not having running time complexity $f(n)$, but rather that it is has running time complexity in $O(f(n))$. This proposal is supposed to avoid the problem posed by the possibility that an algorithm may have implementations with distinct running times by agreeing to predicate of algorithms only asympoptic running times, thereby "factoring out" or "abstracting away from" the implementation specific

details on which such differences are (implicitly) claimed to depend.

The ARTT has several consequences which are relevant in the current context. First, it entails that all acceptable implementations of the same algorithm have the same asymptotic running time (although they may of course differ in exact running time). This is a well confirmed hypothesis in practice and is supported by the sort of anecdotal evidence provided in standard sources like [24], [125] and [57]. And although in initially proposing the use of asymptotic measures Knuth [67], [66] does attempt to adduce some foundational considerations in its favor, it is hard to see how ARTT could be formally demonstrated in the absence of a mathematical theory of algorithms of the sort I argued in Chapter 2 that an algorithmic realist must endeavor to construct. What is more significant to our current concerns, however, is that it follows from ARTT that since exact (as oppose to asymptotic) running time complexity should not be attributed to individual algorithms, a statement like "A has running time $f(|x|)$" cannot be taken as expressing a fundamental property of A. According to the proponent of ARTT this is, of course, because A is likely to have implementations whose running time differs from $f(n)$.

Of course it is also a consequence of ARTT that statements of the form "A has running time $O(f|x|)$" ought to express as fundamental properties of algorithms. This conclusion is consistent with RT since it follows from Corollary 1 that $O(f|x|)$ running time will be invariant under $\underline{\leftrightarrow}_{\mathcal{R}}$. But it is the converse direction of RT – i.e. the claim that all $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant properties are fundamental to individual algorithms – which is more problematic. For note that it follows immediately from this result that *exact* running time with respect to the complexity index f_i is also a $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant property. But the sort of examples mentioned above illustrated that this is not the case – i.e. there are concrete instances in which we are willing to regard implementations like the RAM machines M_1 and M_1 as implementations of the same algorithm, despite the fact that they differ in exact running time.

If the implementations M_1 and M_2 were themselves recursors, this observation would be would be sufficient to demonstrate (5.3ii), which in turn would invalidate RT itself. But as things stand, it may appear that since register-based implementations of this sort are quite distinct from recursors, that their divergence in exact running time has little bearing on

the status of RT. However, we have also seen that register-based models can, in general, be assimilated to recursion-based ones such as the recursion schemes of Chapter 5.5. And for this reason, there is a prima facie reason to fear that there will exist recursors α_{M_1} and α_{M_2} deriving from these M_1 and M_2 which will still be such that $tr^i_{\alpha_{M_1}}(|x|) \neq tr^i_{\alpha_{M_2}}(|x|)$. But in this case, RT would predict that α_{M_1} and α_{M_2} represent different algorithms in virtue of the fact that they fail to satisfy the same $\underline{\leftrightarrow}_{\mathcal{R}}$ invariant properties. I will return to this situation in the next section, where will we see that Moschovakis's definition of recursor reduction does in fact commit him not only to the existence of α_{M_1} and α_{M_2} also to the view that they should indeed be viewed as implementations of A.

5.2.4 Recursors and identity

Recall that the onus falls on the defender of RT to show that the definition of recursor isomorphism $\underline{\leftrightarrow}_{\mathcal{R}}$ is extensionally adequate. In much the same manner as in Chapter 3, this claim can be taken to have to have two components:

(5.9) i) For all recursors α_1, α_2, if $\alpha_1 \underline{\leftrightarrow}_{\mathcal{R}} \alpha_2$, then we intuitively accept that $imp(\alpha_1) = imp(\alpha_2)$.

ii) If we intuitively accept that recursors α_1 and α_2 represent the same algorithm (i.e. $imp(\alpha_1) = imp(\alpha_2)$), then $\alpha_1 \underline{\leftrightarrow}_{\mathcal{R}} \alpha_2$.

As I will now attempt to demonstrate, however, the definition of $\underline{\leftrightarrow}_{\mathcal{R}}$ given in Definition 5.2.3 can be seen to fail with respect to both requirements (5.9i) and ii).

To see this, note that recursors, as their name suggests, most directly mirror the structure of recursive algorithms which are standardly presented in recursive form such as Euclid's algorithm or MERGESORT. However, the Thesis is intended to have general scope – i.e. it asserts that *all* algorithms are faithfully modeled by recursors, not just those which are conveniently expressed in recursive form. On the face of things, one might think that Moschovakis's theory of algorithms was poorly equipped to deal with the general case. But in addition to the recursor model, he also proposes a theory about how algorithms which would most naturally be expressed in other ways can be represented as recursors.

In order to get an impression of how this auxiliary theory works, we may begin by noting that Moschovakis [94] introduces another basic model of computation which he refers to as an *iterator*. Iterators correspond very closely to our prior definition of transition systems and for present purposes I will assume that they are given in the same form – i.e. as a tuple $\phi = \langle X, W, S, \delta, T, in, out\rangle$ consisting of an input set X, and output set W, a set of state S, a set of terminal states $T \subset S$, an input function $in : X \to S$ and an output function $out : S \to Y$. Moschovakis presents iterators as a natural analysis of the informal notion of what he takes to be the informal notion of implementation, which, in our terms, can be taken to correspond to an instance of a transition- or register-based model of computation. He goes on acknowledge that iterators may be used to directly represent algorithms which we would express with iterative constructs such as while and for-do loops. This includes not only the "naive" (i.e. $O(n^2)$) sorting algorithms (e.g. SELECTIONSORT, INSERTIONSORT, etc.) which are but also a great many "sophisticated" procedures like the Ford-Fulkerson maximum flow algorithm, many efficient Fast Fourier Transform algorithms and the AKS primailty test.[15]

It may readily be shown, however, that an arbitrary iterator ϕ may be represented as a recursor. This is achieved by "decompiling" iteration into tail recursion as expressed by the FLR term

(5.10) $\alpha_\phi = p(in(x))$ where $\{p(s(x)) =$ if $s(x) \in T$ then $out(s(x))$ else $p(\delta(s(x))\}$.

Thus by RT, we should hold that two iterators ϕ_1, ϕ_2 express the same algorithm just in case the corresponding recursors α_{ϕ_1} and α_{ϕ_2} are isomorphic. However, as we are now about to see, recursor isomorphism is both too fine and too coarse an equivalence relation to individuate algorithms represented by iterators in this manner.

We may first note that it is also possible to define a notion of isomorphism for iterators directly:

Definition 5.2.4. Iterators ϕ_1 and ϕ_2 from X to W are isomorphic (symbolically $\phi_1 \underline{\leftrightarrow}_\mathfrak{I} \phi_2$)

[15] It is conjectured that algorithms solving P-complete problems like the Ford-Fulerson algorithm are *intrinsically iterative* in the sense that they cannot be parallelized. If this is correct, then there are a great many algorithms which admit only "degenerate" representations as recursors of the sort described in the next paragraph.

just in case there exists a map $\rho : S_1 \rightarrow S_2$ such that i) $\rho(in_1(x)) = in_2(x)$ for all $x \in S$; ii) $\rho(\delta_1(s(x))) = \delta_2(\rho(s(x)))$ for all $s \in S$; and iii) $\rho(out_1(s)) = out_2(s)$ for all $s \in T$ such that there is an $x \in X$ and n such that $s^n(in_1(x)) = s$.

It is shown in [96] that two iterators are isomorophic just in case their corresponding recursors are isomorphic – i.e. $\phi_1 \underline{\leftrightarrow}_{\mathcal{I}} \phi_2$ if and only if $\alpha_{\phi_1} \underline{\leftrightarrow}_{\mathcal{R}} \alpha_{\phi_2}$. Note, however, any two iterators ϕ_1 and ϕ_2 of type $X \rightarrow W$ which are such that $len_{\phi_1}(x) = len_{\phi_2}(x)$ for all $x \in X$ are trivially isomorphic. In order to see this, note that we may *define* $\rho : S_1 \rightarrow S_2$ inductively by $\rho(in_1(x)) := in_2(x)$ and $\rho(\delta_1(s)) := \delta_2(\rho(s))$.[16]

Now consider any two informally specified iterative algorithms A_1 and A_2 which have the same exact running time $time(x)$. It will generally be straightforward to represent A_1 and A_2 as iterators ϕ_{A_1} and ϕ_{A_2} such that $len_{\phi_{A_1}}(x) = time(x) = len_{\phi_{A_2}}(x)$. From this it follows that $\phi_{A_1} \underline{\leftrightarrow}_{\mathcal{I}} \phi_{A_2}$. But since ϕ_{A_1} and ϕ_{A_2} will uniquely determine recursors α_{A_1} α_{A_2}, it also follows by (5.9ii) that $\alpha_{A_1} \underline{\leftrightarrow}_{\mathcal{R}} \alpha_{A_2}$. And from this we may conclude via (5.9i) that the algorithm represented by α_{A_1} (i.e. A_1) is *identical* to the algorithm represented by α_{A_2} (i.e. A_2). In the case where A_1 and A_2 are chosen to be intuitively distinct,[17] this means that RT yields the wrong conclusion about their identity conditions.

It is in this sense that recursor isomorphism turns out to be too coarse grained a relation over recursors and as such violates the adequacy condition (5.9i). A particularly salient example of these problems arises for recursors expresesed by FLR terms of the form

(5.11) $\mu(x) = m(x)$ where $\{m(x) = f(x)$ if $P(x)$ else $g(m(h_1(x), m(h_2(x))\}$

As we have seen in Chapter 4, a recursor of this form would naturally be used to represent an algorithm A such as MERGESORT whose execution on x requires the construction of a full binary tree of depth $|x|$. However, it can shown that μ may be uniformly transformed into two distinct iterators ψ_1 and ψ_1 which (intuitively) differ in that ψ_1 traverses this tree

[16] Note that by this route, we also see that *exact* running time complexity is a $\underline{\leftrightarrow}_{\mathcal{I}}$-invariant property of iterators.

[17] It is easy to construct examples of intuitively distinct algorithms A_1 and A_2 with the same exact running time. For instance, we might start out with two $O(n^2)$ sorting algorithms – e.g. SELECTIONSORT and INSERTIONSORT – and simply "pad" one or the other with linearly many dummy steps to ensure equality of running time for all inputs. However, this is only the tip of the iceberg. For instance, note that we may have $\forall x[time_{A_1}(x) = time_{A_2}(x)]$ even in case where A_1 and A_2 solve different problems on different mathematical structures.

by always going left until it must go right while ψ_2 goes right until it must go left. We would informally speak of these iterators as both being different *implementations* of A. However, the recursors μ_1 and μ_2 which are reconstrutable from ψ_1 and ψ_2 in the sense of (2) will be such that $\mu_1 \underline{\leftrightarrows} \mu$, $\mu_2 \underline{\leftrightarrows}_{\mathcal{R}} \mu$. And thus via (5.9i), we ought to conclude that μ_1 and μ_2 both represent algorithms which are *not* identical to μ. But this violates (5.9ii) in that μ_1 and μ_2 are both derived from machines which, like μ, we would naturally describe as implementations of A.

The foregoing observations may be formalized within the framework of the theory of program schemes and recursion schemes introduced developed in Greibach [47]. Relative to the substitution-based semantics for recursion schemes presented therein, μ does not correspond to a completely determinate procedure for computing a value. In addition to the operational (or "intensional") semantics for recursors presented in [92], a so-called *reduction strategy* for recursors is required which specifies the order in which substitutions are to be performed.[18] Two distinct strategies of this sort are embodied by ψ_1 and ψ_2. Using the techniques of Paterson and Hewitt [106], it may be shown formally that these iterators must both employ a stack-like data structure to keep track of the recursive calls made by μ. The explicit specifications of ψ_1 and ψ_2 will, however, be much more complicated than μ. And it is thus doubtful that the definition of $\underline{\leftrightarrow}_{\mathcal{I}}$ cited above can be modified so as to accomodate the requirement that $\psi_1 \underline{\leftrightarrow}_{\mathcal{I}} \psi_2$ in any manner which does not, *ipso facto* also relate other iterators representing intuitively distinct algorithms.

---

[18] As stressed by Gurevich and Blass [9], the fact that the natural fixed-point semantics for recursors do not provide such a strategy may potentially be seen as a negative feature of this model. For if the meaning assigned to (5.11) by such a semantics does not tell us explicitly how to compute the values of the corresponding recursors in a step-by-step manner, then it may be argued that recursors do not represent represent determinate algorithms after all. Moschovakis has argued that this a virtue of the recursor model in that computational features of this sort are not relevant to the standard correctness and complexity arguments employed in the analysis of algorithms. But at the same time, the items which we call algorithms in practice appear to bear possess such properties intrinsically. This demonstrates that there may be some tension between Moschovakis desire to *identify* algorithms in terms of recursors and his claim that they bear only those computational properties which figure in such proofs.

5.3 Gurevich and Abstract State Machines

5.3.1 The ASM Thesis

If we continue to employ the terminology of Chapters 2 and 3, Gurevich's proposal about the mathematical status of algorithms can be described as a form of reductionism whereby it is claimed that individual algorithms can be represented by particular members of a class of model known as Abstract State Machines. As I will explain more fully below, ASMs bear a close historical and technical relationship with many register- and transition-based models of computation we have encountered previously. Since I have argued in Chapter 4.4 that all models of these sorts can be specified as transition systems in the sense defined in Chapter 2.4, Gurevich's overall proposal can be understood as approximately corresponding to the claim that every informally specified algorithm can be correlated with at least one such system.

Gurevich refers to this proposal as the ASM Thesis. This claim was first announced under this name in [54], although closely related statements can also be found in [156] (wherein a predecessor of ASMs are referred to as *dynamical structures*) and in [50] (wherein another closely related predecessor is referred to as *evolving algebras*). He and others have subsequently revisited and expanded upon the Thesis in a number of articles including [52], [51], [9], [10] as well as the recent book by Böger and Stark [14]. Due to the variety of presentations which both the ASM model and the ASM Thesis have received, we face a minor textual challenge because both the model and Thesis have been stated differently in different sources. This will not matter much with respect to the definition of the model itself since the variants differ in ways which are not relevant to current interests. But the precise form of Gurevich's various statements of the Thesis is significant because it reveals his position about the ontological relationship between algorithms and ASMs.

Before even defining ASMs precisely, it will thus be useful to examine different forms of

the "official" statement of the ASM Thesis:

> The *abstract state machine thesis* asserts that abstract state machines express algorithms on their level of abstraction in a direct and coding-free manner. [52], p. 1
>
> The thesis is that every sequential algorithm, on any level of abstraction, can be viewed as a sequential abstract state machine. [54], p. 2
>
> Every sequential algorithm can be step-for-step simulated by an appropriate sequential ASM. [51], p. 78
>
> The ASM Thesis [is] the claim that every algorithm can be expressed, on its natural level of abstraction, by an ASM. [9], p. 96

The point which should be noted concerns the contribution of the modifier "sequential" with respect to both algorithms and ASMs. In both instances, the relevant contrasting term is "parallel" as it is used to describe a class of algorithm (and a corresponding class of AMSs) whose execution consists of several different subprocesses which are carried out at the same time and whose results (and possibly intermediate states) can be put back together to obtain a single result as output. Most of the algorithms in traditional algorithmic analysis and which have been considered in this work have been sequential. And thus although Gurevich [10] has stated a corresponding version of the ASM for parallel algorithms, the qualification appearing in the given statements of the ASM Thesis do not restrict its scope relative to our current goals.

The other two observation which I wish to initially record about the ASM Thesis pertain the relationship which Gurevich suggests ASMs stand in with respect to algorithms. First, note that the ASM Thesis does not state that algorithms *are* ASMs, but rather that each algorithm alternately "can be viewed as", "can be step-for-step simulated by" or "can be expressed by" an ASM. On the basis of his selection of these terms, it reasonable to assume that the relationship which Gurevich takes to exist between algorithms and ASMs is something short of strict identity. And second, note that in addition to this, none of the given versions of the ASM Thesis states that for every algorithm A there is a *unique* ASM M which bears any of these relations to it, but merely that there exists some such model.

It is thus reasonable to view the ASM Thesis as akin to a form of what I have called Extended Church Thesis in previous chapters. This is to say that rather than making a strict ontological claim that algorithms are literally identical to the members of a given model of computation, the ASM Thesis merely states that they may be represented or

modeled by instances of such a formalism with a given degree of fidelity. As we have seen previously, however, the strength and plausibility of such a view are largely determined by the sort of representational relationship which is claimed to exist between the two classes of entities. And sadly, despite the expositions of the ASM Thesis which are available, Gurevich is much less clear on these points than Moschovakis.

In particular, we may also note that although the ASM Thesis may superficially appear to resemble the Recursor Thesis, this impression is diminished if we take into account other aspects of Moschovakis's overall proposal. For first note that although RT stops short of explicitly identifying algorithms with recursors, it does says that the latter serves "faithful models" of the former up to recursor isomorphism. One important difference between the ASM Thesis and RT is that Moschovakis says substantially more about how he understands the notion of faithful representation than Gurevich ever says about the notions of "expression" or "step-for-step simulation" which appear the above statement of the former Thesis. In particular, as we saw in the prior section, Moschovakis says explicitly that for one class of mathematical objects, M_C faithfully models another class of objects C with respect to an equivalence relation $\sim_C$ just in case the $\sim_C$ invariant properties of M_Cs correspond to the fundamental mathematical properties of the Cs. It thus follows from RT that if a recursor α faithfully models an algorithm A, then A and α share all of their fundamental properties. And similarly (by the argument suggested in the previous section) if two recursors faithfully model the same algorithm, than they must be isomorphic. It is for this latter reason that I suggested above that RT should be understood as a form of abstractionism and also why it can be taken to address the algorithmic realist's central claim that algorithms are genuinely identical to mathematical objects.

Since Gurevich neither presents an account of the properties which must be preserved across the ASMs $M_1, M_2, \ldots$ which are taken to represent a single algorithm A, nor does he present an equivalence relation which is alleged to hold between them, his proposal ought to be viewed as substantially weaker than that of Moschovakis. As such, it is simultaneously more likely to be a demonstrably true statement about algorithms and also of less direct interest to our current interests with respect to algorithmic realism. With this said, however, that are several reasons why it will be valuable for us to examine both the ASM model and

Thesis in more detail. Among these are the following. First, by examining the definition of ASMs in more detail, it is possible to note a number of systematic affinities between these structures and the more traditional models of computation studied in Chapter 4. In particular, it will become evident that AMSs just are transitions whose states are described using some of the vocabulary of first-order logic. Second, on this basis we can see that ASMs bear essentially the same relationship to recursors as do program schemes to recursion schemes. And for this reason, it is possible to argue that the ASM thesis is *false* modulo the proper classification of recursive procedures like MERGESORT as sequential algorithms. And finally, despite this apparent weakness of the ASM model, Gurevich and Blass [9] use some related considerations to argue that the recursor model is itself defective as an allowable analysis of our background notion of implementation. And thus although I believe that the ASM Thesis must ultimately be taken to orthogonal to that of algorithmic realism, it does offer some valuable insight into the status of this other programme.

5.3.2 On ASMs

As I have already mentioned, the official definition of an ASM has undergone several minor modifications since its introduction in [156]. For present purposes, however, I will mostly standardize on [51] as giving the canonical presentation while relying on the modern presentation in [14] as providing further specification which fills in several minor gaps in this presentation. Proceeding in the manner of [51] it useful to preface the definition of ASM by starting out by considering a somewhat wider class of structures which I will refer to as *sequential systems*[19]:

Definition 5.3.1. A *sequential system* A is a triple $\langle \mathcal{S}(A), \mathcal{I}(A), \tau_A \rangle$ such that

i) $\mathcal{S}(A)$ is a class of mathematical structures called the *states* of A;

ii) $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ are called the *initial states* of A;

iii) $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$ is called the *one step transformation* of A.

The *run* of a sequential system is a sequence of states $\sigma_0, \sigma_1, \ldots$ where $\sigma_0 \in \mathcal{I}(A)$ and $\sigma_{i+1} = \tau_A(\sigma_i)$.

It should be immediately apparent that this definition is similar to that of a transition system as given in Chapter 2.4. What is missing, of course, are explicit specifications of input and output sets X and Y, corresponding to input and output mappings between such sets and $\mathcal{S}(A)$, and a designation of certain states within $\mathcal{S}(A)$ as being terminal. However, it is evident from Gurevich's discussion of sequential systems that these features will generally be recoverable from explicit presentations of sequential systems in an obvious manner. It is also clear that in addition to the definition of a run of a sequential system (which corresponds to what I called the execution of a transition system) we may give a corresponding definition of the function induced by a executing a given sequential structure.[20] As such, I will assume that there is essentially no technical difference between sequential systems and transitions systems as defined in Chapters 2 and 3.

The precise definition of an ASM is obtained from that of a sequential structure by limiting the classes of allowable states and one-step transformation functions. Gurevich's fundamental idea in both respects is to use the resources of first-order logic to precisely specify both the structure of states and also what it means to perform a bounded or "small step" transformation on a state. The characterization of states is achieved by what is referred to as the *abstract state postulate*:

Definition 5.3.2 (Abstract State Postulate)**.** Let $A = \langle \mathcal{S}(A), \mathcal{I}(A), \tau_A \rangle$ be a sequential system and $\mathcal{L}_A$ be a first-order language. Then A satisfies the abstract state postulate just in case:

i) the states of $\mathcal{S}(A)$ correspond to a class of pairs $\langle \mathfrak{A}, v \rangle$ where $\mathfrak{A}$ is a $\mathcal{L}_A$ structure and

[19] In [51] Gurevich refers to such structures as *sequential algorithms*, a notion which he has previously explained as that corresponding to what is given by "a finite text [such as a program] that explains the algorithm without presupposing any special knowledge." This ultimately allows him to prove a theorem([51], Theorem 6.13) whose statement reads as follows: "For every sequential algorithm A, there exists an equivalent sequential abstract state machine B." As we see below, however, ASMs are defined to be a particular class of sequential systems. If we stick to Gurevich's own use of terminology, the result just stated has only a narrow technical signficance (whose interpretation is complicated by the fact that the relevant notion of "equivalence" is not defined). But in both [51] as well as several later papers such as [10], Gurevich purports to be giving an *analysis* of the general notion of algorithm (at least in the weak sense discussed above). And since some considerations can be adduced in favor of this hypothesis, retaining Gurevich's convention of referring to sequential systems as sequential *algorithms* makes his account appear circular in a way which is not only presumably unintended, but also slightly unfair.

[20] Gurevich concedes this as well – cf. [51], p. 8, 9 – but states that he leaves these elements out of the definition of sequential system to eliminate inessential detail.

v is a variable assignment for $\mathcal{L}_A$ (i.e. a mapping from the set of variables $Var = \{x_1, x_2, \ldots\}$ to the domain $|\mathfrak{A}|$ of $\mathfrak{A}$;

ii) each structure $\langle \mathfrak{A}, v\rangle \in \mathcal{S}(A)$ is such that $\mathfrak{A}$ has the same domain;

iii) the class of $\mathcal{L}_A$ corresponding to the first component of $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphism and any isomorphism between these components of a states $\mathfrak{A}, \mathfrak{A}' \in \mathcal{S}(A)$ is also an isomorphism between the first components of $\tau_A(\langle \mathfrak{A}, v\rangle)$ and $\tau_A(\langle \mathfrak{A}', v\rangle)$.[21]

The motivation for this characterization of states is grounded in several observations which Gurevich suggests are grounded in computational practice. First, any specification of an algorithm A (as either a program, a pseudocode specification, or an implementation), can be taken as being given in some fixed first-order vocabulary of functions $f_1, \ldots, f_{n_2}$, predicates $P_1, \ldots, P_{n_2}$, constants $c_1, \ldots, c_{n_3}$ and variables $x_1, x_2, \ldots = Vars$ which can be taken to correspond to the language $\mathcal{L}_A$. And second, it will generally be possible to present the intermediate states of A as corresponding to an $\mathcal{L}_A$-structures $\mathfrak{A}$ together with a variable assignment function v. For present purposes, we may assume that the one-step transformation of A modifies only the variable assignment component of a state $\langle \mathfrak{A}, v\rangle$. This means that all transformations will be of the form $\tau_A(\langle \mathfrak{A}, v\rangle) = \langle \mathfrak{A}, v'\rangle$.[22]

The second requirement which Gurevich wishes to impose on a sequential structure A pertains to the manner in which its one step transformation τ_A is defined. In particular, he wishes to impose the restriction that if $\tau_A(\langle \mathfrak{A}, v\rangle) = \langle \mathfrak{A}, v'\rangle$ then value is determined from σ by performing only a bounded amount of "exploration" – i.e. by testing only the values only finitely many terms over $\mathfrak{A}$. In order to formalize this requirement, let $\Delta_A(\langle \mathfrak{A}, v\rangle) = \{x \in Var \ : \ v(x) \neq v'(x) \text{ where } \tau_A(\langle \mathfrak{A}, v\rangle) = \langle \mathfrak{B}, v'\rangle\}$. Gurevich now proposed the following criterion:

---

[22]The present formulation of sequential systems deviates from that given in [51] wherein states are taken to consist of first-order structures $\mathfrak{A}$ alone. In this presentation, the interpretation of arbitrary function symbols $f(x_1, \ldots, x_n)$ is allowed to change between states and it by modifying the interpretation of terms of the form $f(a_1, \ldots, a_n)$ that states are "updated" via τ_A. However this assumption has several inconvenient consequences and was latter modified in [10] so as to allow for only a specified class of *dynamic functions* to be updated. The approach I have adopted wherein it is variable assignment for $\mathcal{L}_A$ over $\mathfrak{A}$ which is updated is based on that employed by Böger and Stark [14]. In this context ii) and iii) can be dropped in favor of stipulation that all states of A are based on the same first-order structure. This modification both requires only minor modifications to the general theory of ASMs and has the advantage of highlighting the similarity of ASMs to earlier models such as the program schemes of Chapter 4 and register machine model employed in Dynamic First Order Logic in the sense of [56].

Definition 5.3.3 (Bounded Exploration Postulate)**.** A sequential system A is said to satisfy the *Bounded Exploration Postulate* just in case there exists a finite set of variables $V \subseteq Var$ such that for all states $\langle \mathfrak{A}, v\rangle, \langle \mathfrak{B}, v'\rangle \in \mathbb{S}(A)$ if $\forall x \in V[v(x) = v'(x)]$, then $\Delta_A(\langle \mathfrak{A}, v\rangle) = \Delta_A(\langle \mathfrak{B}, v'\rangle)$.[23]

The apparent intention of this proposal is that the set of variables V is intended to serve as the set of locations in the state $\langle \mathfrak{A}, v\rangle$ which are "explored" by A during the course of computation on a given initial state. In particular, this condition appears to be at least as strong as the analogously motivated family of boundedness condition which I considered in Chapter 4.3.[24]

The third and final restriction which Gurevich wishes to place on AMSs is grounded in observations about the ways in which the states of an algorithm may allowably evolve from step to step. The first of these is grounded in the observation that in paradigmatic informal specification of algorithms, the fundamental computational operation is that of assignment of a value to a variable. As we have seen previously, this operation is expressed in imperative programming languages by statements of the form $\mathsf{Let}\ x = f(t_1, \ldots, t_n)$ and in the context of program schemes by a statement of the form $x \leftarrow f(t_1, \ldots, t_n)$. Gurevich adopts a version of this syntax in the form of what he refers to as a $\mathfrak{L}_A$-*local update rule* which has the form $f(t_1, \ldots, t_n) := x$ where f is any n-ary functional expression of $\mathfrak{L}_A$ and $t_1, \ldots, t_n$ are all terms over this language. The intended effect of executing a statement of this form is to update the value associated with the variable x so that it corresponds to the value denoted by the term $f(t_1, \ldots, t_n)$ evaluated in the current state. We may thus take the interpretation of such an assignment statement to be a function on the state of A given by $\delta^x_{f(t_1,\ldots,t_n)} : \mathbb{S}(A) \to \mathbb{S}(A)$ defined by $\delta^x_{f(t_1,\ldots,t_n)}(\langle \mathfrak{A}, v\rangle) = \langle \mathfrak{A}, v'\rangle$ where $v'(y) = v(y)$ for all $y \neq x$ and $v'(x) = f^{\langle \mathfrak{A}, v\rangle}(t_1^{\langle \mathfrak{A}, v\rangle}, \ldots, t_n^{\langle \mathfrak{A}, v\rangle})$. I will refer to such a function as a *local*

[24] One important feature of this definition is that the class of variables V is fixed independently of the states $\langle \mathfrak{A}, v\rangle$ and $\langle \mathfrak{B}, v'\rangle$. In informal terms, this means that the size of the class of variables values on which the value of τ_A can depend cannot grow over the course of a computation. It can be shown on this basis that most of the instances of the transition- and register-based models of computation discussed in Chapter 4 (e.g. multi-tape, multi-head Turing machines, RAM machines, program schemes) are real-time simulatable with ASMs in the sense of [121]. However this presumably does not hold for the Storage Modification Machine model of Schönhage, as certain machines in this class will not satisfy the Bounded Exploration Postulate (essentially for the same reasons discussed in Chapter 4.4). While Gurevich (e.g. [53], [10]) is certainly aware of this result, it is unclear what significance he takes it to hold.

update of A.

Gurevich claims that over the course of a single step in the execution of an algorithm, its state many change with respect to the values stored in at most finitely many locations (i.e. associated with finitely many variables). Paradigmatically, such a *bounded update* may be achieved by executing at most finitely many local updates rules in parallel. If $R_1, \ldots, R_k$ are all local update rules, such a parallel update is expressed in the formal language described in [51] as what I will refer to as a $\mathfrak{L}_A$-*bounded update rule* of the `par` $\{R_1, \ldots, R_k\}$ `endpar`. Such a rule is said to be *consistent* if for no pair $i, j \leq k$ of distinct indices do we have $f(t_1, \ldots, t_n) = x$ and $g(s_1, \ldots, s_m) = x$ for distinct functional expressions f, g or terms $t_1, \ldots, t_n, s_1, \ldots, s_m$. We can now define a bounded update as one which is the product of executing a consistent bounded update rule. Formally, if $R \equiv$ `par` $\{R_1, \ldots, R_k\}$ `endpar` and δ_i is the local update associated with R_i, then the bounded update $\delta^R : \mathcal{S}(A) \to \mathcal{S}(A)$ is given by the composition $\delta_1(\delta_2(\ldots(\delta_k(\sigma))))$.

Note that if A were such that $\tau_A = \delta^R$ for some bounded consistent rule R, then we would have that $\Delta(\langle \mathfrak{A}, v \rangle)$ was finite and hence that A satisfied the Bounded Exploration Postulate. However, such a condition would be overly restrictive since it would not allow A to mediate which update was applied according to the structure of the current state. For this reason, Gurevich proposes to extend the class of $\mathfrak{L}_A$-bounded update rules to what he calls an ASM *program* over $\mathfrak{L}_A$ which are defined and interpreted as follows:

Definition 5.3.4. Let $\mathfrak{L}_A$ be a first-order language, $\mathfrak{A}$ an $\mathfrak{L}_A$ structure and v a variable assignment relative for $\mathfrak{A}$. Then the class of ASM programs over $\mathfrak{L}_A$ is smallest class of expressions $Prog_A$ such that

i) $R \in Prog_A$ for all bounded update rules R over $\mathfrak{L}_A$;

ii) if $\Pi_1, \Pi_2 \in Prog_A$, then so is $\Pi \equiv$ `if` φ `then` Π_1 `else` Π_2.

The interpretation $\delta^\Pi : \mathcal{S}(A) \to \mathcal{S}(A)$ of a program $\Pi \in Prog_A$ relative to $\langle \mathfrak{A}, v \rangle$ is given as follows:

i) if $\Pi \equiv R$ for some bounded update rules R, then $\delta^\Pi = \delta^R$;

ii) if $\Pi \equiv \texttt{if } \varphi \texttt{ then } \Pi_1 \texttt{ else } \Pi_2$ then δ^{Π} is defined as follows:

$$\delta^{S}(\langle \mathfrak{A}, v \rangle) = \begin{cases} \delta^{R_1}(\langle \mathfrak{A}, v \rangle) & \text{if } \mathfrak{A} \models_v \varphi \\ \delta^{R_2}(\langle \mathfrak{A}, v \rangle) & \text{else} \end{cases}$$

On the basis of the foregoing definition, the full definition of an ASM can now be stated as follows:

Definition 5.3.5. Let $\mathfrak{L}_B$ be a first-order language. Then an *abstract state machine* is a structure $B = \langle \mathcal{S}(B), \mathcal{I}(B), \Pi_B \rangle$ such that

i) Π_B is an ASM program over $\mathfrak{L}_B$;

ii) $\mathcal{S}(A)$ is a class of pairs of the form $\langle \mathfrak{A}, v \rangle$ for $\mathfrak{A}$ an A structure and v a variable assignment over $\mathfrak{A}$ which is closed under isomorphism and under the map δ^{Π};

iii) $\mathcal{I}(B)$ is a subset of $\mathcal{S}(B)$ and is closed under isomorphism.

It will be immediately noted that a sequential ASM B over $\mathfrak{L}_B$ is simply a sequential structure whose transition mapping τ_B is given by some ASM program $\Pi \in Prog_B$. Note that such a sequential system of this kind will satisfy the Abstract State and Bounded Exploration Postulates by construction. And it is on the basis of this observation which Gurevich [51], [9], [10] has motivated this model as a maximally natural and general model of computation.

Quite a bit of inductive evidence can in fact be cited for this claim. For note that since ASM programs are defined inductively, it is clear that they can be used to emulate arbitrary case definitions by expressions of the form

$$\texttt{if } \varphi_1 \texttt{ then } R_1 \texttt{ else } \{\texttt{if } \varphi_2 \texttt{ then } R_2 \texttt{ else } \{\ldots \texttt{ if } \varphi_k \texttt{ then } R_k\} \ldots\}$$

This immediately enables us to demonstrate the following:

Proposition 5.3.1. Let $B = \langle \mathcal{S}(B), \mathcal{I}(B), \Pi_B \rangle$ be an ASM over the language $\mathfrak{L}_B$ and $M = \langle X, Y, St, \Delta, H, in, out \rangle$ a transition system in the sense of Definition 4.1 of Chapter 3. Then:

i) There exists a transition system $M_B = \langle X_B, Y_B, St_B, \Delta_B, H_B, in_B, out_B\rangle$ such that there exists a bijection $s : \mathfrak{S}(B) \rightarrow St_B$ such that for all states $\langle \mathfrak{B}, v\rangle \in \mathfrak{S}(B)$, $s(\delta^{\Pi}(\langle \mathfrak{B}, v\rangle)) = \Delta_B(s(\langle \mathfrak{B}, v\rangle))$.

ii) If M is such that St can be described and a class of finite vectors $\langle u_1, \ldots, u_n\rangle$ over some class U, then there is an first-order language $\mathfrak{L}_M$ and an ASM $B_M = \langle \mathfrak{S}(B_M), \mathfrak{I}(B_M), \Pi_{B_M}\rangle$ over $\mathfrak{L}_M$ such that there exists a bijection $t : St_M \rightarrow \mathfrak{S}(B_M)$ such that for all $\sigma \in St_M$, $t(\Delta_M(\sigma)) = \delta^{\Pi}(t(\sigma))$.

The first component of this result shows that for any ASM B, there is a transition system M_B which bisimulates it in the strong sense that there is a bijective mapping between the states of B and M_B whose behavior is preserved by the one-state transformation of M.[25] The second component states that there is a partial converse in the case that the states of M are describable as finite vectors of objects (which can be taken to be the values assigned by a variable assignment function over an appropriate first-order structure with domain U).[26]

Since the definition of an ASM and a transition system are so similar, this result is hardly surprising and its proof is entirely routine. However, since all of the individual transitions systems which we have considered in Chapters 1, 3, and 4 man be shown to satisfy the hypothesis of Proposition 3.1.ii), it follows that they may be naturally represented as ASMs.

---

[25]Note, however, that if we wish to view B and M_B as equivalent descriptions of the same computational system, the following caveat needs to be taken into account. Since the definition of ASM does not include a specification of input or output sets or functions, the transition system M_B is not uniquely determined by the structure of B. This means that the sets X_B, Y_B and functions $in_B : X_B \rightarrow St_B$ and $out_B : St_B \rightarrow Y_B$ map be chosen arbitrarily. And as such, the function determined as $App(M_B, x)$ may diverge from that which B was intended to compute.

[26]The restriction on St is required only to ensure that S_M can be constructed in a manner so that its one-step transformation is given by an ASM program which operates to updates variables. Note, however, that is a very minor restriction since the elements of $\langle u_1, \ldots, u_n\rangle$ can be arbitrary mathematical structures. This requirement is obviously satisfied by the transition system representations of algorithms we have considered in previous chapters. For recall that in many of these examples, states have already been presented as finite vectors of values. In addition to this, however, it is also clear that transition systems whose statements we informally describe as being composed of one or more infinitary structures can be treated in this manner. For note that all we may simply take such structures as the objects constituting the domain $\mathfrak{A}_A$ of a first-order structures which can serve as $\mathfrak{S}(B_M)$. This observation suggests that the requirement that states of an ASM be described as first-order structures is in fact very week. This constraint should be contrast with the requirement that the states of an ASM must be described as first-order *definable* structures, say as a subclass of the class of structures $Mod(T)^{\kappa}$ which are definable (up to isomorphism) as the models of a first-order theory T of cardinality κ. It is clear, however, that without imposing this and other restrictions on the definition of AMS there will be such systems which compute non-recursive functions. For an attempt to modify the definition of AMS to rule out such presumably pathological cases, see [11].

This observation can readily be generalized to show that virtually all of the transition- and register-based models of computation consider considered in Chapter 4 are such that their instances can readily be assimilated to ASMs (per Proposition 3.1.ii)). And there is thus reason to believe that with respect to representing individual implementations, the ASM model is every bit as general as the class of transitions systems.

This observation also goes a fair distance towards allowing us to understand what Gurevich is intending to claim via the ASM Thesis. In particular, as I have attempted to demonstrate throughout this work, transitions systems appear to provide a "natural" or "direct" medium in which to precisely represent the intended interpretation of most informal specifications of algorithms. In particular, if A is an algorithm informally specified using pseudocode, it is generally possible to construct a transition system M_A which represents A in the sense that the formally defined transitions among its states match our intuitive demarcation of the operation of A into stages. By Proposition 3.1, we know that there will be a transition B_A which bisimulates M_A (and hence) A. And since it seems reasonable to interpret Gurevich as having essentially this sort of step-by-step relation in mind in his various statements of the ASM Thesis, it also seems reasonable to regard this proposal as well confirmed.

The validity of the ASM Thesis does, however, come along with two fairly significant caveats. The first of these is by now quite familiar: although the relationship between ASMs with transition systems codified by Proposition 3.1 means that a great many informally specified algorithms will have intuitively acceptable implementations as ASMs, we have not seen any reason to think that the task of choosing a "canonical" ASM B_A to represent a given algorithm A is any more constrained than it is in the case of transitions. In fact the close relationship between ASMs and transition systems suggests that just as there will generally be an open class of formally distinct transitions $M_1, M_2, \ldots$ which can be taken to implement a given algorithm A with equal plausibility, there is every reason to think there will be a corresponding open class of ASMs $B_1, B_2, \ldots$. In other words, the move from transition systems to ASMs does not solve the problem of multiple implementability.

If we are to view the ASM Thesis simply as a form of Extended Church's Thesis as I suggested above, then this observation does not detract from its plausibility. However, we

may also ask if there is any way in which the view can be naturally strengthened or modified so that it can be taken to have a full blown form of algorithm realism as its conclusion. For reasons which are also now familiar, one response would be to attempt to formulate a definition of $\underline{\leftrightarrow}$ over the class of all ASMs and then argue that it was both extensionally and intensionally adequate. Although the transition-based structure of AMSs suggests that this can be accomplished using the familiar methodology of Chapter 3, Gurevich does not take up the issue of how the details of such a definition could be worked out.[27] However, the close relationship between algorithms and transition systems again suggests that there is no reason to expect that an adequate definition can be formulated. For in particular, since we saw in Chapter 3 that an arbitrary Turing machine $T \in \mathcal{T}$ can be uniquely associated with a transition system M_T with the same transitional behavior, it is clearly also possible to similarly associate an ASM B_T. And in this way, it will be possible to reiterate the argument of Chapter 3 that no extensionally adequate definition of $\underline{\leftrightarrow}$ over the class of ASMs can be given.

Of course the ASM Thesis was never explicitly marketed as part of an argument for algorithmic realism. And so the foregoing considerations do not threaten its status as an argument for a clear proposal such as Extended Church's Thesis. However, it should also be evident that another familiar consideration can be adduced which at least appears to limit the scope of the algorithm with can be directly assimilated to the ASM Thesis in the manner which the foregoing overview suggests. This is, of course, the existence of algorithms whose informal specifications are given recursively – e.g. by informal recursive definitions of the sort considered in Chapter 4.5. The considerations adduced there suggested that while such algorithms can be readily represented as recursions schemes, Theorem 16 of Chapter 4 showed that there were instances of such schemes which could not be represented by program schemes (at least in the sense of strong equivalence which was explained there). I

[27] In [51] he does, however define a formal notion of equivalence for sequential structures: $A_1 = \langle \mathcal{S}(A_1), \mathcal{I}(A_1), \tau_{A_1} \rangle$ is equivalent to $A_2 = \langle \mathcal{S}(A_2), \mathcal{I}(A_2), \tau_{A_2} \rangle$ just in $\mathcal{S}(A_1) = \mathcal{S}(A_2)$, $\mathcal{I}(A_2) = \mathcal{I}(A_1)$ and $\tau_{A_1} = \tau_{A_2}$. This is (obviously) the finest equivalent possible. Although Gurevich observes that coarser equivalence are possible, he does not appear to have considered adopting one. One could, however, naturally take advantage of the first-order setting in which sequential structures and AMSs are defined so as to define a coarser equivalence using standard notions from first-order model theory which are weaker than isomorphism (e.g. elementary equivalence, n-quantifier depth equivalence, etc.). However, there appear to have no attempts to refine the AMS model in this way.

have argued previously that MERGESORT is paradigmatic of such algorithms and we have now also seen in Section 2 that this algorithm can be represented as a recursor in a manner that arguably reflects its most salient computational properties.

The question thus arises whether this algorithm can also be naturally expressed as an ASM. Gurevich's view on this question appears to have shifted somewhat over time. For in his earlier work on AMS (e.g. [156] [50]) no explicit mention is made of recursion. In [52], however, he explicitly acknowledges that recursive algorithms such as MERGESORT cannot be subsumed under the version of the ASM Thesis we have been considering. In this paper he proposes instead that the ASM model be modified with a construct which allows a single ASM B_0 to recursively construct one or more distinct sub-ASMs $B_{01}, \ldots, B_{0k}$. The sub-ASMs are employed to carry out sub-computations which they are assigned by B_0, during the course of which they too can create additional slave machines. If at some point during its computation, one of the slave machine converges (i.e. computes an answer to the subproblem which it has been assigned), then it reports this answer back to machine which has created it, which in turn can combine it with answers reported by the other sub-machines which it has created.

This is a familiar form of informal description of how a recursive computation can be carried out. However the means by which Gurevich proposes to analyze even a relatively simple recursive procedure like MERGESORT employ a number of metaphors from parallel and distributed computing. The invocation of these devices in turn brings in a variety of complex issues concerning the technical notions of fairness and synchronicity. These issues are not resolved in this paper. Ad their proper treatment would involve a substantial digression into parallel models of computation I will thus not discuss the status of Gurevich's proposal to analyze recursive algorithm such as a form of distributive algorithm further here. It seems likely, however, that the general theory of parallel and distributive algorithms (as developed in, e.g., [79]) is already so complex as to render it doubtful that such analysis could ever be considered more fundamental than the sort of semantic treatment of such procedures via recursion schemes or recursors we have already considered.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. *PRIMES* is in *P*. *Annals of Mathematics*, 160(2):781–793, 1004.

[2] K. Appel and W. Haken. Every planar map is four colorable part I: Discharging. *Illinois Journal of Mathematics*, 21:429–490, 1977.

[3] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable part II: Reducibility. *Illinois Journal of Mathematics*, 21:491–567, 1977.

[4] Krzystof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag graduate texts in computer science series. Springer-Verlag, New York, NY, 2nd ed. edition, 1997.

[5] Jon Barwise and John Etchemendy. *Turing's World 3.0*. Center for the study of Language and Information, 1995.

[6] George Bealer. Theories of properties, relations, and propositions. *The Journal of Philosophy*, 76:634–648, 1979.

[7] P. Benacerraf. What numbers could not be. In H. Benacerraf, P. & Putnam, editor, *Philosophy of mathematics*. Cambridge University Press, 1983.

[8] Patrick BLackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001.

[9] Andreas Blass and Yuri Gurevich. Algorithms vs. machines. *Bulletin of the EATCS*, 77:96–119, 2002.

[10] Andreas Blass and Yuri Gurevich. Algorithms: A quest for absolute definitions. *Bulletin of the EATCS*, 81:195–225, 2003.

[11] U. Boker and N. Dershowitz. A formalization of the Church-Turing Thesis for state-transition models. Manuscript.

[12] G. Boolos. Is hume's principle analytic? In R. Jeffrey, editor, *Logic, logic, and logic*, pages 171–182. Harvard University Press, 1998.

[13] G. Boolos. Saving frege from contradition. In R. Jeffrey, editor, *Logic, logic, and logic*, pages 171–182. Harvard University Press, 1998.

[14] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[15] U. Bottazinni. *The higher calculus*. Springer Verlag, 1986.

[16] G. Burgess, J. & Rosen. *A subject without an object*. Oxford University Press, 1997.

[17] R. Carnap. *Meaning and Necessity*. University of Chicago Press, 1956.

[18] Jean-Luc Chabert, editor. *A history of algorithms: From the pebble to the microchip*. Springer-Verlag, 1999.

[19] L. Childs. *A Concrete Introduction to Higher Algebra*. Springer-Verlag, Berlin, 1979.

[20] A. Church. A simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[21] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd IEEE Symposium on the foundations of computer science*, pages 151–158, 1971.

[22] S. A. Cook and S. O. Aanderaa. On the minimum computation time of functions. *Transactions of the AMS*, 142:291–314, 1969.

[23] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. *J. Comput. System Sci.*, 7:354–375, 1973.

[24] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1991.

[25] M. Cresswell. *Creswell, Structured Meaning*. MIT Press, 1985.

[26] Martin Davis. Why Gödel didn't have Church's thesis. *Information and Control*, 54:3–24, 1982.

[27] R. Dedekind. *Was sind und was sollen die Zahlen*. Friedrich Vieweg., Braunschweig, 1888.

[28] M. Detlefsen, M; Luker. The four-color theorem and mathematical proof. *The Journal of Philosophy*, 77, 1980.

[29] M. Dummett. *Frege: Philosophy of language*. Duckworth, 1973.

[30] M. Dummett. What is a theory of meaning? In S. Guttenplan, editor, *Mind and language*. Oxford Univeristy Press, 1975.

[31] M. Dummett. *Elements of intuitionism*. Oxford, second edition, 2000.

[32] Joost Engelfriet. *Simple Program Schemes and Formal Languages*, volume 20 of *Lecture Notes in Computer Science*. Springer, 1974.

[33] Fetzer. Philosophical aspects of program verification. *MANDMS: Minds and Machines*, 1, 1991.

[34] H. Field. *Science Without Numbers: A Defense of Nominalism*. Basil Blackwell and Princeton University Press, 1980.

[35] S. Ford, L & Johnson. A tournament problem. *American Mathematical Monthly*, 66:387–389, 1959.

[36] G. Frege. What is a function? Translated by P. T. Geach in Geach and Black (1970): 107–116.

[37] G. Frege. 'Über sinn und bedeutung'. *Zeitschrift für Philosophie und philosophische Kritik*, pages 25–50, 1892. Translation by Max Black in Geach and Black (1970): 56–78.

[38] G. Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet*. Hildesheim, 1893. Translation by Montgomery Furth: *The Basic Laws of Arithmetic: Exposition of the system*, University of California Press, Berkeley (1967).

[39] G. Frege. *The foundations of airhtmetic: a logico-mathematical enquiry into the concept of numbers*. Blackwell, 1950.

[40] G. Frege. Compound thoughts. *Mind*, 72:1–17, 1963. Translated by R. Stoothoff.

[41] Haim Gaifman. Naming and diagonalization, from cantor to gödel to kleene. *Logic Journal of the IGPL*, 14(5):709–728, 2006.

[42] Robin Gandy. Church's thesis and principles for mechanisms. In K. Jon Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 123–148. North-Holland, Amsterdam, 1980a.

[43] Robin Gandy. The confluence of ideas in 1936. In Rolf Herkin, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 55–111. Oxford University Press, Oxford, 1988.

[44] K. Gödel. On undecidable propostions of formal matehmatical systems. In M. Davis, editor, *The undecidable*. Raven Press, 1965.

[45] Kurt Gödel. On formally undecidable propositions of *Principia Mathematica* and related systems I. In Solomon Feferman et al., editors, *Kurt Gödel. Collected Works*, volume I, pages 144–195. Oxford University Press, 1986.

[46] Michael T. Goodrich and Roberto Tamassia. *Algorithm design*. John Wiley and Sons, Inc., 2002.

[47] Sheila A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*, volume 36 of *Lecture Notes in Computer Science*. Springer, 1975.

[48] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[49] Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. Elsevier Science Publishers, 1990.

[50] Gurevich. Evolving algebras; a tutorial introduction. *Bulletin EATCS*, 43:264–284, 1991.

[51] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM transactions on computational logic*, 1(1):77–111, 2000.

[52] Y. Gurevich and M. Spielmann. Recursive abstract state machines. *Journal of Universal Computer Science*, 3(4):233–246, 1997.

[53] Yuri Gurevich. Kolmogorov machines and related issues. *Bulletin of the European Association for Theoretical Computer Science*, 35:71–82, June 1988. Columns: Logic in Computer Science.

[54] Yuri Gurevich. The sequential ASM thesis. *Bulletin of the EATCS*, 67:93, 1999.

[55] R. Hale. *Abstract Objects*. Blackwell, 1987.

[56] David Harel. *First-order dynamic logic*. PhD thesis, MIT, 1978.

[57] David Harel. *Algorithmics: the spirit of computing*. Addison-Wesley, 1987.

[58] R. Harnish. *Minds, Brains, Computers: An historical introduction to the foundations of cognitive science*. Blackwll, 2002.

[59] Juris Hartmanis. Computational complexity of random acess stored program machines. *Mathematical Systems Theory*, 5(3):232–245, 1971.

[60] G. Hellman. *Mathematics Without Numbers 1989.* Oxford University Press, 1989.

[61] F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, December 1965.

[62] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, 1969.

[63] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[64] Neil D. Jones. *Computability and Complexity From a Programming Perspective*. The MIT Press, 1997.

[65] S. C. Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, Amsterdam, 1952.

[66] Donald E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18–23, 1976.

[67] Knuth, D. E. *Mathematical Analysis of Algorithms*. Information Processing, 1971.

[68] A. N. Kolmogorov and V. A. Uspenskiĭ. On the definition of an algorithm. *American Mathematical Society Translations. Series 2*, 29:217–245, 1963.

[69] Georg Kreisel. Some reasons for generalizing recursion theory. In *Logic Colloquium '69*, pages 139–198, Amsterdam, 1969. North Holland.

[70] S. Kripke. *Wittgenstein on rules and private language*. Harvard University Press, 1982.

[71] K. Kunen. *Set Theory: An Introduction to Independence Proofs*. North-Holland, Amsterdam, 1983.

[72] D. Kunth. *The art of computer programming*, volume I-III. Addison Wesley, 1973.

[73] Leong and Seiferas. New real-time simulations of multihead tape units. *JACM: Journal of the ACM*, 28, 1981.

[74] H. Lewis and C. Papadimitiou. *Elements of the theory of computation.* Prentince Hall, 1981.

[75] M. Li and P. Vitanyi. *An introduction to Kolmogorov complexity theory and its applications.* Springer Verlag, 1998.

[76] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley Longman, Inc., second edition, April 1999.

[77] E.J. Lowe. The metaphysics of abstract objects. *Journal of Philosophy*, 92(10):509–524, 1995.

[78] Luckham, Park, and Paterson. On formalised computer programs. *JCSS: Journal of Computer and System Sciences*, 4, 1970.

[79] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[80] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms.* North-Holland, Amsterdam, 1978.

[81] P. Maddy. Does $v = l$ß? *Journal of Symbolic Logic*, 58:15–41, 1993.

[82] A.A. Markov. On constructive mathematics. *American Mathematical Society Translations, II*, 198:1–9, 1967.

[83] D. Marr. *Vision.* Freeman, 1982.

[84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibioplois, Napoli, 1984.

[85] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–69. North Holland, 1963.

[86] Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1201–1242. Elsevier Science Publishers, 1990.

[87] Marvin L. Minsky. *Computation: Finite and infinite machines.* Prentice-Hall, Englewood Cliffs, 1967.

[88] J. Mitchell. *Foundations for programming languages.* MIT Press, 2000.

[89] R. Montague. Pragmatics and intensional logic. *Synthèse*, 22:68–94, 1970.

[90] Richard Montague. On the nature of certain philosophical entities. *The Monist*, 53:159–194, 1969.

[91] Moschovakis. A mathematical modeling of pure, recursive algorithms. In *LFCS: The 1st International Symposium on Logical Foundations of Computer Science*, 1989.

[92] Y. Moschovakis. The formal language of recursion. *Journal of symbolic logic*, 64:1215–1262, 1989.

[93] Y. Moschovakis. Sense and denotation as algorithm and value. In J. Oikkonen & J. Vaananen, editor, *Lecture Notes in Logic*, pages 210–249. Springer, 1994.

[94] Y. Moschovakis. The logic of functional recursion, 1997.

[95] Y. Moschovakis. On the founding of a theory of algorithms. In H. G. Dales & G. Oliveri, editor, *Truth in mathematics*, pages 71–104. Clarendon Press, 1998.

[96] Y. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics unlimited – 2001 and beyond*, pages 919–936. Spring, 2001.

[97] Y. Moschovakis. On primitive recursive algorithms and the greatest common divisor. *Theoretical Computer Science*, 301:1–30, 2003.

[98] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[99] Y. N. Moschovakis. *Abstract Recursion as a Foundation for the Theory of Algorithms*. Department of Mathematical, University of California, Los Angeles, CA, 1984.

[100] Nielson R. Nielson, F. *Principles of Program Analysis Springer*. Springer Verlag, 2005.

[101] Harold W. Noonan. Count nouns and mass nouns. *Analysis*, 38, 1978.

[102] Harold W. Noonan. Dummett on abstract objects. *Analysis*, 36, 1978.

[103] Piergiorgio Odifreddi. *Classical Recursion Theory*. Elsevier, Amsterdam, 1999.

[104] G. E. Ostrin and S. S. Wainer. Proof theoretic complexity, 2002. Unpublished.

[105] Charles Parsons. Mathematical intuition. *Proceedings of the Aristotlean society*, 80:145–168, 1980.

[106] Paterson and Hewitt. Comparative schematology. In *MACCCSPC: Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970.

[107] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[108] Rozsa Péter. *Recursive Functions*. Academic Press, New York, 1967.

[109] Dag Prawitz. Meaning approached via proofs. *Synthese*, 148(3):507–524, 2006.

[110] Z. Pylyshyn. *Computation and congition*. MIT Press, 1986.

[111] W.V.O. Quine. On what there is. In *From a Logical Point of View 2nd ed.* Harvard University Press, 1980.

[112] W.V.O. Quine and N. Goodman. Steps toward a constructive nominalism. *Journal ofSymbolic Logic*, 12, 1947.

[113] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Book Company, New York, 1967.

[114] B. K. Rosen. Tree-manipulation systems and church-rosser theorems. *Journal of the Association for Computing Machinery*, 20:160–187, 1973.

[115] B. Russell. *The principles of mathematics*. Cambridge University Press, 1903.

[116] R.W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics: Mathematical Aspects of Computer Science*, volume 19, pages 19–31, 1967.

[117] N. Salmon. *Frege's puzzle*. Ridgeview, 1991.

[118] J. Savage. *Models of computation: Exploring the power of computing*. Addison Wesley, 1998.

[119] Stephen Schiffer. *The things we mean*. Oxford University Press, Oxford/New York, 2003.

[120] A. Schonage. Real-time simulation of multidimensional turing machine by storage modification machines. Technical Report MIT/LCS/TM-37, Massachusetts Institute of Technology, December 1973.

[121] Schönhage. Storage modification machines. *SICOMP: SIAM Journal on Computing*, 9, 1980.

[122] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[123] John R. Searle. A taxonomy of illocutionary acts. In K. Gunderson, editor, *Language, Mind and Knowledge*. University of Minnesota Press, 1976.

[124] John R. Searle and Daniel Vandervecken. *Foundations of Illocutionary Logic*. Cambridge University Press, Cambridge, England, 1985.

[125] R. Sedgewick. *Algorithms*. Addison Wesley, 1988.

[126] G. Senizergues. The equivalence problem for deterministic pushdown automata is decidable. *Lecture Notes in Computer Science*, 1256:671–681, 1997.

[127] Stewart Shapiro. *Philosophy of Mathematics: Structure and Ontology*. Oxford University Press, 1997.

[128] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, April 1963.

[129] Wilfred Sieg. Mechanical procedures and mathematical experiences. In A. George, editor, *Mathematics and Mind*, pages 71–117. Oxford University Press, 1994.

[130] Wilfried Sieg and John Byrnes. G?del, turing, and K-graph machines, January 26 2000.

[131] Stephen G. Simpson. *Subsystems of second order arithmetic*. Perspectives in Mathematical Logic. Springer-Verlag, 1999.

[132] Thoralf Skolem. The foundations of elementary arithmetic. In J. Van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1967.

[133] Slisenko. Computational complexity of string and graph identification. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1979.

[134] S. Soames. Direct reference, propositional attitudes and semantic content. *Philosophical Topics*, 15:47–87, 1987.

[135] Robert I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, Berlin, 1987.

[136] Robert Stalnaker. *Inquiry*. Bradford Books, MIT Press, 1984.

[137] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[138] P. Strawson. *Individuals: An Essay in Descriptive Metaphysics*. Methuen, London, 1959.

[139] Ch. Sturm. Mémoire sur la résolution des équations numériques. *Institut de France de Sciences Mathématiques et Physiques*, 6:271–318, 1835.

[140] C. Swoyer. Structural representation and surrogative reasoning. *Synthese*, 87:449–508, 1991.

[141] Alfred Tarski, A. Mostowski, and Raphael Robinson. *Undecidable Theories*. North-Holland, Amsterdam, 1953.

[142] A. S. Troelstra. Realizability. In Samuel R. Buss, editor, *Handbook of Proof Theory*, pages 407–473. Elsevier Science Publishers, Amsterdam, 1998.

[143] D. Troelstra, A. S. & van Dalen. *Constructivism in Mathematics: An Introduction, vol. I*. North Holland, 1988.

[144] Turing. On computable numbers, with an application to the entscheidungsproblem. In *Martin Davis, The Undecidable*. Raven Press, 1965.

[145] T. Tymoczko. The four-color problem and its philosophical significance. *Journal of Philosophy 76 (1979) 57-83*, 76(57-83), 1979.

[146] Peter Unger. I do not exist. In G. F. Macdonald, editor, *Perception and Identity*. Macmillan, London, 1979.

[147] Peter Unger. There are no ordinary things. *Synthése*, 41:117–154, 1980.

[148] Johan van Benthem. *Modal and Classical Logic*. Bibliopolis, 1983.

[149] Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. Elsevier, 1990.

[150] J. van Heijenoort. Frege on sense identity. In *Collected papers*. Bibliopolis, 1985.

[151] Webster, editor. *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster, Springfield, 1988.

[152] A. N. Whitehead and B. Russel. Principia mathematica, 1910–1913.

[153] Walt Whitman. Song of the open road. In *Leaves of Grass*. Small, Maynard & Company, 1897.

[154] C. Wright. *Frege's conception of numbers as objects*. Aberdeen University Press, 1983.

[155] R. Wright, C. & Hale. *The reasons proper study*. Oxford University Press, 2004.

[156] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

[157] S. Yablo. Go figure: A path through fictionalism. In P. French and H. Wettstein, editors, *Midwest Studies in Philosophy Volume XXV: Figurative Language*, pages 72–102. Blackwell, 2001.

[158] Noson S. Yanofsky. A universal approach to self-referential paradoxes, incompleteness and fixed points. *Bulletin of Symbolic Logic*, 9(3):362–386, 2003.

[159] A.P. Youschkevitch. The concept of function up to the middle of the 19th century. *Archive for the history of the exact sciences*, 16:37–85, 1976.

Vita

Walter Dean

2007	PhD in Philosophy, Rutgers University
1999	MS in Logic from the University of Amsterdam
1995	AB in Philosophy and Mathematics from Brown University
1995	ScB in Cognitive Science from Brown University

2000-2004	Teaching Assistant, Department of Philosophy, Rutgers University
2005-2006	Adjunct Lecturer, Department of Philosophy, Queens College of the City University of New York
2007	Adjunct Assistant Professor, Department of Philosophy, Baruch College of the City University of New York