Linear-C: A Data-Parallel Extension to C

Chung-Hsing Hsu and Donald Smith and Saul Levy
Department of Computer Science
Rutgers University
LCSR-TR-273

September 9, 1996

Abstract

Linear-C is a data-parallel extension to C. It extends C by treating arrays (or more precisely, aggregate value) as first-class citizens. It provides activity association and segment association to control aggregate values in a computation. It also supplies new operators for aggregate values. Linear-C is designed to be intuitively simple to learn, to use, yet powerful enough to express many kinds of data parallelism. In this report we show that (1) the rationale behind our decision to choose a particular manipulation rule, and (2) the whole picture of the language in a systematic way. In the first part, we identify the trade-offs between the simplicity, the expressiveness, and the implementability of Linear-C. After that, the entire language is introduced systematically, to show the simplicity of the language model and the consistency of the manipulation rules. Finally, we conclude the report in terms of possible future research directions.

1 Introduction

Linear-C is a data-parallel, C-based language. It extends C by treating arrays (or more precisely, aggregate values) as first-class citizens. Furthermore, several kinds of association are attached to arrays to control the computation. Activity association, for example, prohibits particular elements of an array from being involved in a computation. Besides that, Linear-C provides a set of unary operators to manipulate a multi-value. The benefits of Linear-C over C are

- concise representation of a program,
- explicit data-parallel operation, and
- efficient executables. [1]

Linear-C introduces the concept of imperative substitutives, where the state in Linear-C is identical to that in C. This feature allows us to use our familiar C-debugging tools to examine the state between statements, making Linear-C easy to debug.

Though the intuition of using aggregate value is simple, there are challenges in defining the semantics of Linear-C. For example, how does activity control affect the computation? There are also some trade-offs between the design of Linear-C and its implementation. These are all the considerations of this report.

The rest of report is organized in two parts. The first part, Section 3, discusses the rationale behind the design of Linear-C, and the second part, Section 4, defines the entire language. The report is then concluded by future research directions in Section 5.

2 Notation: A Tiny Introduction of Linear-C

In this section we will introduce most of the notations used in this report. All the notations are derived from a step-by-step consideration of the Linear-C language model.

First of all, when we talk about *an array of some kind*, we refer to the common understanding of *a list of elements of that kind.* And the *size* of an array is referred to the number of elements in that array. In C, a simple array variable stores an array of scalar values. This list of scalar values is called an *aggregate value.*

Scalar values, l, m, n, are the only first-class citizens in C. Namely, each C expression computes a scalar value. To operate on an array in C each scalar element of that array is computed. Constants $1, 2, 3$ are examples of scalar values.

Linear-C considers an aggregate value, (n_i), as a first-class citizen too. Aggregate values are stored in array variables. Therefore, for example, if array A contains three elements $1, 2, 3$, we say array A contains an aggregate value $(1, 2, 3)$.

In Linear-C, *pseudo vectors* and *vectors* are the only types of aggregate values allowed. Both add controls on top of aggregate values. A pseudo vector adds activity control on an aggregate value. A

[1] As a direct support for Linear-C, we have developed a new type of memory chip, the CAM2000, that provides efficient ways to compute data-parallel operations.

vector adds not only activity control but also segment control on an aggregate value. In terms of element type, we can describe pseudo vectors and vectors in the following way. A pseudo vector is an array of activity-controlled scalar values, and a vector is an array of pseudo vectors.

An activity-controlled scalar value, $m : n$, is a scalar value n with an activity indicator m, which is another scalar value. If an activity indicator is non-zero, then its associated scalar value is active; otherwise, the value is inactive. An activity-controlled scalar value can be best summarized in the following equation:

$$m : n = \begin{cases} n & \text{if } m \neq 0 \\ \phi & \text{otherwise} \end{cases}$$

where ϕ represents *the universal identity.*

In this report, we abbreviate the notation of an activity-controlled scalar value $m : n$ as follows: if it is active, we simply write n; otherwise, we write $\phi(n)$. Now we can write equations such as $1 + 2 = 3$, $\phi(1) + 2 = 2$.

A pseudo vector, $[m_i : n_i]$, is an array of activity-controlled scalar values. And a vector $[[m_i : n_i]_j]$ is an array of pseudo vectors. Therefore, for example, $[1, 2, 3]$ and $[1, \phi(2), 3]$ are pseudo vectors, but $[[1], [2, 3]]$ and $[[1, \phi(2)], [3]]$ are vectors. In the pseudo vector $[1, \phi(2), 3]$, both scalar values 1 and 3 are active while scalar value 2 is inactive. In the vector $[[1], [2, 3]]$, two segments $[1]$ and $[2, 3]$ are in this vector.

From time to time we need to talk about pseudo vectors and vectors as a whole, usually for the simplicity of the semantics, we use notations $[a_i]$, $[b_i]$, $[c_i]$ to refer them. If scalar values are considered as well, we use notations α, β instead.

Two more symbols for particular values are introduced. One is the unknown value ?, indicating that the value is unknown and is usually implementation dependent. The other is the don't-care value $-$, meaning that we don't care about the exact value of it at this point.

Linear-C provides ways to manipulate pseudo vectors and vectors. One way is to extend the C operators to handle them. In this report we use $\oplus$ to denote extended C binary operators, such as $+, -*, /$. A special extended operator, ?, is called *activity operator* in Linear-C. It is used to re-associate activity control with a (pseudo) vector. In other words, it allows us to re-activate an inactive scalar or de-activate an active scalar. For example, $1?\phi(2)$ re-activates the inactive scalar 2 and gets the result 2. On the other hand, $0?2$ de-activates the active scalar 2 and gets the result $\phi(2)$.

Another way to manipulate pseudo vectors/vectors are the new (unary) operators, called *collective operators* in Linear-C. A subclass of collective operators is called *reduction operators* which summarize a pseudo vector into a scalar value, or a vector into a pseudo vector. For example, operator /\ denotes the minimum-finding operator. Therefore, we will have $/\backslash[1, 2, 3] = 1$ and $/\backslash[[1], [2, 3]] = [1, 2]$. The minimum operator finds out from $[1, 2, 3]$ the smallest scalar value, which is 1. When it applies to a vector, it finds the smallest scalar value for each segment of the vector, which turns out to be 1 and 2 for each segment.

Symbol ? is used to indicate an unknown value, usually implementation-dependent. Symbol $\surd$ indicates an allowed entry in the table, while symbol $\times$ indicates a forbidden entry. All the other notations and Linear-C operators are introduced when necessary.

Finally, we summarize all the notations mentioned above in the following table.

term	notation	definition	examples
scalar value	m, n		1,2,3
aggregate value	(a_i)	an array of scalar values	(1,2,3)
activity-controlled scalar value	$m : n, n, \phi(n)$	a scalar value with an activity indicator	1:1, 2, $\phi(3)$
pseudo vector	$[m_i : n_i]$	an array of activity-controlled scalar values	[1,2,3], [1,ϕ(2),3]
vector	$[[m_i : n_i]_j]$	an array of pseudo vectors	[[1],[2,3]],[[1,ϕ(2)],[3]]
	$[a_i], [b_i], [c_i]$	a pseudo vector or a vector	
	α, β	a scalar value, a pseudo vector or a vector	
	?	an unknown value	
	-	a don't-care value	
Linear-C binary operator	$\alpha \oplus \beta$	extended C binary operators	[1,2,3] + [1,ϕ(2),3]
activity operator	$\alpha?[b_i]$	associate activity indicator α with $[b_i]$	[0,1,ϕ(2)] ? [1,ϕ(2),3]
minimum-finding operator	$/\backslash[a_i]$	find the smallest scalar value of $[a_i]$	$/\backslash[1,2,3] = 1$
size function	$size([a_i])$	the number of elements in $[a_i]$	$size([1,2,3]) = 3$ $size([[1],[2,3]]) = 2$
	$\surd$	an allowed entry	
	$\times$	a forbidden entry	

3 Rationale behind the Language

Linear-C is designed to be intuitively simple for its language model, concise and consistent for its manipulation rules, and powerful enough to express many kinds of data parallelism. However, in practice, one goal may conflict with another so that decisions must be made for the trade-offs. For example, intuitively powerful model may result in many exceptions for the manipulation rules in order to satisfy a user's expectation. Furthermore, the trade-off between implementation and design must be considered as well. Sometimes powerful design may result in complicated implementation or introduce additional cost for the code. Along the way in defining Linear-C, many such trade-offs emerge. This section intends to record the reasons of all the important decisions we have made when defining Linear-C.

The section is organized in a topic-by-topic manner. Topics may refer to each other during the discussion.

3.1 The Implementation's View of Linear-C Values

Linear-C provides three kinds of values: scalar value, pseudo vector, and vectors.

type	activation	segmentation	value
scalar value	no	no	single-valued
pseudo vector	yes	no	multi-valued
vector	yes	yes	multi-valued

Vectors are aggregate values with activity control and segment control, while pseudo vectors only have activity control. Both kinds of values are treated as first-class citizens, part of special features of Linear-C. This section will examine them from an implementor's view.

In the implementation, a vector can be thought of as a structure with three fields: one for segment control, one for activity control, and one for its multi-valued content.

$$vector \equiv <seg, act, val>$$

Here, *seg*, *act*, and *val* represents three different aggregate values. Value *seg* is used for segment control: non-zero segment indictor starts a new segment; zero segment indictor continues the current segment. Value *act* is used for activity control: non-zero activity indictor activates the corresponding content; zero activity control de-activates it.

Pseudo vectors can be implemented in a similar way. The only difference is that pseudo vectors do not have segmentation.

$$pseudovector \equiv <\times, act, val>$$

The following is a set of examples showing the correspondence between a particular Linear-C value and its implementation. Take the value <(1,0,1),(1,0,1),(1,2,3)> for an example. Its *seg* field $(1, 0, 1)$ says that the value has two segments: the first two elements are in one segment while the third element starts the other segment, i.e., [[-,-],[-]]. The *act* field $(1, 0, 1)$ says that the first and the third elements are active while the second element is inactive, i.e., [[-,ϕ(-)],[-]]. And the *val* field $(1, 2, 3)$ fills in the value for each element. As a result, <(1,0,1),(1,0,1),(1,2,3)> implements the value [[1,ϕ(2)],[3]].

value	implementation
[1,2,3]	<×,(1,1,1),(1,2,3)>
[1,ϕ(2),3]	<×,(1,0,1),(1,2,3)>
[[1,2],[3]]	<(1,0,1),(1,1,1),(1,2,3)>
[[1,ϕ(2)],[3]]	<(1,0,1),(1,0,1),(1,2,3)>

In the following sections, we will use both notations interchangeably to help illustration.

3.2 Pseudo Vectors as a New Type

A pseudo vector comes as the result of applying a reduction on a vector. It is pseudo because its arithmetic rule is context-dependent. In some contexts it is treated as a scalar, and in other cases it is considered as a vector. The following paragraph gives a description of what a pseudo vector behaves [1].

> [A pseudo vector] is a multiple-value object, with one value for each segment, but syntactically (and semantically) it is treated as a scalar, with its values being automatically broadcast over the corresponding segments in vector operations. On the other hand, operations between true scalars and pseudo vectors broadcast the scalar across the values of the pseudo vector, and it's possible to do a collective function over a pseudo vector.

The decision we have to make is then as follows.

Is the type of a pseudo vector, a scalar or a vector, determined at run-time, or is a new static type introduced?

We choose pseudo vector as a new type due to the following. Pseudo vectors are intuitively simple but difficult to implement efficiently. First, the size of a pseudo vector can only be determined dynamically, depending on the number of segments of its vector input when it is produced. Second, the arithmetic rule for a pseudo vector can only be determined dynamically, depending on the type of the other operand in a binary computation. This non-static property of pseudo vector makes it hard to be incorporated in a statically type-checking language such as C. As a result, Linear-C introduces pseudo vector as another kind of type, distinguishable from vector and scalar. And the reductions on different types are thus statically determinable.

$$\text{vector} \overset{\text{reduction}}{\longrightarrow} \text{pseudo vector} \overset{\text{reduction}}{\longrightarrow} \text{scalar} \overset{\text{reduction}}{\longrightarrow} \text{error}$$

Introducing pseudo vector as a new type only solves the second problem of determining the arithmetic rule for it. The first problem of determining its size cannot be solved statically. Such a statically-indeterminable behavior may result in inconsistencies.

expression	evaluation	result
/\([[1,2,3],[4]] + [[1,2],[3,4]])		reject
/\([[1,2,3],[4]]) + /\([[1,2],[3,4]])	[6,4] + [3,7]	[9,11]

The example shows that reduction /\ is *not distributive*. As a result, optimization will be difficult since it cannot arbitrarily distribute reduction over the input operand.

3.3 The Addition of Two Aggregate Values

In principle, we want the arithmetic rule for addition to be simple yet powerful enough to cover all the cases of vector-vector addition, vector-pseudo addition, and pseudo-pseudo addition. The rule for addition could be described, in the most general form, as

$$[a_i] \oplus [b_i] \rightarrow [a_i \oplus b_i]$$

where $\oplus$ is addition (or any binary C-operator), and $[a_i], [b_i]$ are either pseudo vector or vector.

Now we have to decide

Can aggregates of different sizes be combined using a binary C-operator?

Our decision is to enforce the equal-size restriction, $size([a_i]) = size([b_i])$, for the addition rule. The reason is that, the addition of operands of different sizes does not preserve the associativity of addition and it makes the language less statically-checkable. The following paragraphs give some examples.

The first choice for relaxation is to extend the shorter input by filling the extended part with all ϕ's.

expression	evaluation	result
[1,2,3] + [1,1]	[1,2,3] + [1,1,ϕ]	[2,3,3]

This choice is then abandoned due to the following anomalies.

expression	evaluate	result
[[1,2],[3]] + [[1],[2,3]]	[[1,2]+[1],[3]+[2,3]]	[[2,2],[5,3]]
([1,2,3]+1) + [1,2]	[1+1,2+1,3+1] + [1,2]	[3,5,4]
[1,2,3] + (1+[1,2])	[1,2,3] + [1+1,2+1]	[3,5,3]

The first anomaly indicates the static indeterminacy of the size of the result after a binary operation. It will make debugging and optimization harder. A second anomaly shows that the extending approach does not preserve the associativity of addition. It is this second anomaly which makes us discard the extending approach so as to facilitate the optimization.

The second choice for relaxation is to truncate the longer input.

expression	evaluation	result
[1,2,3] + [1,1]	[1,2] + [1,1]	[2,3]

The truncating approach preserves the associativity of addition, but still suffers from the static indeterminacy of the result's size.

expression	evaluate	result
[[1,2],[3]] + [[1],[2,3]]	[[1,2]+[1],[3]+[2,3]]	[[2],[5]]
([1,2,3]+1) + [1,2]	[1+1,2+1,3+1] + [1,2]	[3,5]
[1,2,3] + (1+[1,2])	[1,2,3] + [1+1,2+1]	[3,5]

3.4 When an Aggregate Value Meets a Scalar

A binary operation on aggregate values basically follows the *for-each* (element-wise) semantics; namely, do the operation on each corresponding pair of elements of the two operands. When an aggregate value meets a scalar, there comes a different story. The scalar follows the *for-all* (broadcasting) semantics; namely,

for all elements of the aggregate value, do the operation with the scalar.

or, in short,

$$[a_i] \oplus s \rightarrow [a_i \oplus s]$$

As a result, the following computation is expected.

expression	evaluation	result
[1,2,3] + 1	[1+1,2+1,3+1]	[2,3,4]

Now we have to make the decision that

> Is scalar s promoted first and then the addition rule applied, or is the (pseudo) vector-scalar addition a separate case?

Our answer is to consider the above semantic rule as a separate rule. In other words, there is no scalar promotion in Linear-C. The reason is due to the difficulty introduced by the size of the promoted scalar. On one hand, setting the size context-independently, say one, does not satisfy our expectation of how a scalar behaves.

expression	evaluation	result
[1,2,3] + 1	[1,2,3] + [1]	reject (different size)

Even though the rule is relaxed for operands of different sizes, the result may not be the one expected.

On the other hand, determining the size of the promoted scalar in a context-dependent way makes the scalar more evaluation-order sensitive.

expression	evaluate	result
([1,2,3]+1) + [1,2]	([1,2,3]+[1,1,1]) + [1,2]	?
[1,2,3] + (1+[1,2])	[1,2,3] + ([1,1]+[1,2])	?

The first and second expressions promote the scalar 1 into different aggregate values, i.e., [1,1,1] and [1,1] respectively. This way of scalar promotion makes debugging and optimization harder because they all have to be aware of (or, be restricted to) the evaluation order. This difficulty is the reason why Linear-C only allows the manipulation of aggregate values of the same size. Therefore, Linear-C does not use scalar promotion.

Recall that the functionality of the context is to determine the size of the promoted scalar, and consider the following expression.

A ≡ [0,0]		
expression	evaluate	result
A = 1 ? 2		

If scalar promotion is allowed, then expression 1 ? 2 computes an aggregate value by first promoting both scalars into aggregate values and then following the rule for activity operation. The question is then "what is the size of this resulting aggregate value?" A default-size approach causes the same trouble as context-independent scalar promotion does. Linear-C resolves the problem by considering such an assignment as a binary operation between an array and a scalar (see Section 3.5 for details). In summary, Linear-C never introduces scalar promotion.

3.5 Assignment and Array Promotion

Linear-C, in general, extends C's assignment for aggregate values. In other words, an array, an object which holds a multi-value, can be referred to as the left-hand side of a Linear-C assignment. In contrast, array will be promoted as vector before using on the right-hand side of the assignment. The many kinds of assignment and array promotion are thus the focus of this section.

Are all kinds of assignments, regardless the types of operands allowed?

Linear-C, like C, has an imperative model, viewing program execution as a sequence of state changes, each of which is done through an assignment (=).

state = *computation*

Unlike C, the state in Linear-C may be updated on an aggregate-value basis. More specifically, an assignment in Linear-C is one of the following kinds.

lhs/rhs	scalar value	pseudo vector	vector
scalar object	$\surd$	$\times$	$\times$
array object	$\surd$	$\surd$	$\surd$

As an ordinary binary operator, an assignment requires the sizes of its two operands to be *compatible*. In the current implementation it is saying that the number of values in the assigning operand should be equal to the number of holes, for holding them, in the assigned array. Furthermore, assignment throws out all the segment information of assigning operand. Activity information is used only at the time of assignment, and will be thrown away as well after the operation.

A $\equiv$ (97,98,99)		
expression	evaluate	result
A = [[1,2],[3]] =	A = [1,2,3]	A $\equiv$ (1,2,3)
A = [1,ϕ(2),3]	A = [1,ϕ(2),3]	A $\equiv$ (1,98,3)
A = [1,2]	A = [1,2]	reject (different length)
A = [[1,2],[3],[4]]	A = [1,2,3,4]	reject (different length)

Note that *length* is different than size. Though [[1,2],[3],[4]] has size 3 (three segments), it contains 4 scalar values and has length four. Since A is of length 3, any aggregate value of length other than 3 cannot be assigned into A.

The scalar-to-array assignment still follows the for-all semantics in the sense that the array will be filled with the same scalar value.

A $\equiv$ (97,98,99)		
expression	evaluate	result
A = 2		A $\equiv$ (2,2,2)

The vector-to-scalar assignment is not allowed in Linear-C for a simple reason: there is not enough space to hold all the values of a multi-value. The pseudo-to-scalar assignments may be the most troublesome. Due to the dynamically-sizable feature of a pseudo vector, it is impossible to determine, at compile time, if space will be enough or not. As a consequence, this kind of assignment is rejected in Linear-C.

At final, we consider array promotion. In order to use the multi-value stored in an array, the array will be promoted as one-segment, all-active vector. When a to-array assignment is involved as a subexpression, it is treated in a similar way: side-effects on the array and then array promotion.

A $\equiv$ (1,2), B $\equiv$ (3,4)		
expression	evaluate	result
A + B	[[1,2]] + [[3,4]]	[[4,6]]
(A = 2) + B	[[2,2]] + [[3,4]]	[[5,6]]

3.6 Field Substitution and Activity Operator

This section considers the semantics of activity operator ?. We expect that the operator follows the two rules:

(rule 1) Explicit activity replaces any activity which a value may have, i.e., $l?m : n \equiv l : n$, and

(rule 2) Inactive activity indicators have no effect of replacement, i.e., $\phi(l)?m : n \equiv m : n$.

Though intuitively we understand that it re-associates activity control to a (pseudo) vector, it may have many interpretations.

If we consider a (pseudo) vector as a field-based value, as described in Section 3.1, we have to decide whether $[a_i]?[b_i]$ tells us to compose certain fields of $[a_i]$ and $[b_i]$ into a new value, or to make a new copy of $[b_i]$ and then modify it according to $[a_i]$. The earlier scheme is called *field composition*, and the latter is called *field modification*.

Is the activity operator based on field composition or field modification?

In our design, the activity operator is based on field modification, and its modification is through an assignment to a particular field, rather than the substitution of the whole field. In this way the above two rules are completely satisfied. Detail considerations are as follows.

Suppose we adopt field composition scheme for $[a_i]?[b_i]$. A reasonable field composition is to extract the value fields of $[a_i]$ as the activity field and value field of $[b_i]$ as the new value, separately.

$$<-, act_1, val_1> \ ? \ <-, act_2, val_2> \rightarrow <-, val_1, val_2>$$

where $[a_i] \equiv< -, act_1, val_1 >$ and $[b_i] \equiv< -, act_2, val_2 >$. Unfortunately, such a definition violates **(rule 2)** in the following example.

expression	evaluate	result	expectation
$[\phi(1)] \ ? \ [\phi(2)]$	<-,(0),(1)> ? <-,(0),(2)>	<-,(1),(2)> = [2]	$[\phi(2)]$

In the above example, we expect $\phi(1)$ have no effect on the inactiveness of $\phi(2)$ according to **(rule 2)**. However, it re-activates $\phi(2)$ by the field composition scheme. The problem comes from the composition scheme only extracts the value field of $[a_i]$ without knowing its activity control (act_1). As a result, it cannot tell where the original inactive activity indicators are. The same mis-computation occurs when the modification is simply the replacement of the field with another field.

For the formal semantics of activity operator, see Section 4.4 for details.

3.7 Scan: Inclusive Form or Exclusive Form

Linear-C provides a set of collective operators based on scan (>). These scan operators take an operand as input and return the modified version of it. Since a scan operator, in general, may take either *inclusive form* or *exclusive form* [2] , we must decide

Are Linear-C scan operators based on inclusive form or exclusive form?

The answer is that there is no single form which can cover all Linear-C scan operators. Some of them are in inclusive form while others are in exclusive form. In general, scan operators doing arithmetic computation are defined in *inclusive form*, and scan operators doing data movement are

[2] For the general definition of scan in inclusive form or exclusive form, see Section 4.6 for details.

defined in *exclusive form*. In the following we will use sum-scan operator (>+) and shift-right operator (>>) as the two representatives; sum-scan operator adopts *inclusive form* and shift-right operator adopts *exclusive form*.

For sum-scan operator (>+), the two possible definitions are as follows.

in inclusive form:

$$>+_{IS}(a_1, a_2, \ldots, a_n) \rightarrow (a_1, a_1 + a_2, \ldots, a_1 + \cdots + a_n)$$

in exclusive form:

$$>+_{ES}(a_1, a_2, \ldots, a_n) \rightarrow (?, a_1, a_1 + a_2, \ldots, a_1 + \cdots + a_{n-1})$$

The second definition carries less information than the first definition. It drops out the value $a_1 + \cdots + a_n$ and introduces another unknown value (?) in the first place. This loss of information causes us to choose inclusive form for sum scan operator, and, in general, arithmetic-computing scan operators.

>+(1, 2, 3)	evaluation	result	comments
inclusive form	$(1, 1+2, 1+2+3)$	$(1, 3, 6)$	
exclusive form	$(?, 1, 1+2)$	$(?, 1, 3)$	less information

On the other hand, data-moving scan operators have to take exclusive form to match our expectation that "they are really moving data". Take shift-right operator (>>) for example. We have two choices.

in inclusive form:

$$>>(a_1, a_2, \ldots, a_n) \rightarrow (a_1, a_2, \ldots, a_n)$$

in exclusive form:

$$>>(a_1, a_2, \ldots, a_n) \rightarrow (?, a_1, a_2, \ldots, a_{n-1})$$

Clearly, only the shift-right in exclusive form moves data to its right. Therefore, scan operators for data movement take exclusive form, not inclusive form.

>>(1, 2, 3)	evaluation	result	comments
inclusive form		$(1, 2, 3)$	not moving data
exclusive form		$(?, 1, 2)$	

3.8 Scan on Inactive Elements: Shift-Into or Skip-Over

The semantics of scan operators must define how to deal with inactive elements. One way is for the inactive element to participate in the scan, with value ϕ, and, as a consequence, the element may be modified after the scan. This way of semantics is termed *shift-into semantics*. Another way is to skip over the inactive element in the scan process, and, as a result, the element will stay the same after the scan. We term this way of semantics *skip-over semantics*.

The following example should give a clearer picture about the difference between shift-into semantics and skip-over semantics.

>+[1,ϕ(2),ϕ(3),4]	evaluation	result
shift-into	[1,1+ϕ(2),1+ϕ(2)+ϕ(3),1+ϕ(2)+ϕ(3)+4]	[1,1,1,5]
skip-over	[1,ϕ(2),ϕ(3),1+4]	[1,ϕ(2),ϕ(3),5]

Suppose we want to do sum-scan on pseudo vector $[1, \phi(2), \phi(3), 4]$. In shift-into semantics, even though elements $\phi(2)$ and $\phi(3)$ are inactive, they are involved in the scan process, and they are replaced by 1's and *turned on* after the scan process. Note that, both their values and activity indicators are modified. On the other hand, in skip-into semantics, elements $\phi(2)$ and $\phi(3)$ stay the same (even their activity controls) after the scan operation.

Another key question is

Are Linear-C scan operators based on shift-into semantics or skip-over semantics?

Linear-C adopts shift-into semantics because it correctly interacts with shift operators. More specifically, our decision is based on the following expectation:

$$>>(>+_{IS}[a_i] + >+_{IS}[b_i]) \quad = \quad >+_{ES}[a_i] + >+_{ES}[b_i]$$

where $>+_{IS}, >+_{ES}$ represents the inclusive form and exclusive form of sum-scan, respectively.

Intuitively, we can do sum-scan in inclusive form to both $[a_i]$ and $[b_i]$, add the results and shift the sum to the right, which should give us the same value as we do sum-scan in exclusive form and then add the results. The importance of this equation is that it gives us more opportunity to perform optimization.

The following example shows that while shift-into semantics preserves the equation, skip-over semantics does not. Therefore, skip-over semantics of scan operation is not used by Linear-C.

in shift-into semantics:

$$\begin{array}{ll} & >>(>+_{IS}[1,2,\phi(3)] + >+_{IS}[1,\phi(2),3]) \\ = & >>([1,1+2,1+2+\phi(3)] + [1,1+\phi(2),1+\phi(2)+3]) \\ = & >>([1,3,3] + [1,1,4]) \\ = & >>([1+1,3+1,3+4]) \\ = & >>([2,4,7]) \\ = & [\phi(?),2,4] \end{array} \qquad \begin{array}{ll} & >+_{ES}[1,2,\phi(3)] + >+_{ES}[1,\phi(2),3] \\ = & [\phi(?),1,1+2] + [\phi(?),1,1+\phi(2)] \\ = & [\phi(?),1,3] + [\phi(?),1,1] \\ = & [\phi(?)+\phi(?),1+1,3+1] \\ = & [\phi(?),2,4] \end{array}$$

in skip-over semantics:

$$\begin{array}{ll} & >>(>+_{IS}[1,2,\phi(3)] + >+_{IS}[1,\phi(2),3]) \\ = & >>([1,1+2,\phi(3)] + [1,\phi(2),1+3]) \\ = & >>([1,3,\phi(3)] + [1,\phi(2),4]) \\ = & >>([1+1,3+\phi(2),\phi(3)+4]) \\ = & >>([2,3,4]) \\ = & [\phi(?),2,3] \end{array} \qquad \begin{array}{ll} & >+_{ES}[1,2,\phi(3)] + >+_{ES}[1,\phi(2),3] \\ = & [\phi(?),1,\phi(3)] + [\phi(?),\phi(2),1] \\ = & [\phi(?),1,\phi(3)] + [\phi(?),\phi(2),1] \\ = & [\phi(?)+\phi(?),1+\phi(2),\phi(3)+1] \\ = & [\phi(?),1,1] \end{array}$$

There may be some other ways for doing shift-into. For example, scan can be done by temporarily ignoring activity control and then attaching the bits back to the resulting value. In this way the activity control won't be affected by the scan operation, unlike the above shift-into implementation. Nevertheless, these methods are either too ad hoc and/or too complicated and are not implemented in Linear-C.

4 Linear-C: The Language

Linear-C is designed as a data-parallel extension to C. The basic idea is to treat an aggregate value, stored in an array, as a first-class citizen. C operators are extended to handle aggregate values. Linear-C also provides new operators for manipulating aggregate values, and for temporarily and partially activating and segmenting a computation. All these features allow Linear-C to express many different kinds of data parallelism.

In this section four basic elements of the Linear-C model are examined. First of all is the kinds of values supported by Linear-C. Secondly, the operational semantics of each kind of value is defined. In this way, an expression involving several kinds of values can be interpreted fairly easily. After that, the activation control and segmentation control are specified. Linear-C uses the association of an aggregate value with another aggregate value as the control method. Finally, new operators and their semantics are introduced. A summary then closes the section.

4.1 The Kinds of Values Supported by Linear-C

Linear-C supports three kinds of values: *scalar values*, *pseudo vectors*, and *vectors*. Scalar values do not have activity association and form the basis of aggregate values. Pseudo vectors and vectors are *activity-controlled* aggregate values; however, pseudo vectors are *unsegmented* while vectors are always *segmented*.

value type	activation	segmentation	notation	example
scalar value	no	no		1,2,3
pseudo vector	yes	no	[...]	[1,2,3]
vector	yes	yes	[[...],···,[...]]	[[1,2],[3]]

A vector can be constructed from either an array reference or a vector computation. An array, as described before, contains an aggregate value. When it is referred, the value stored there is promoted as a fully-activated and single-segmented vector.

A ≡ (1,2,3)		
expression	evaluation	result
A	(*promotion*)	[[1,2,3]]
[[1,2,3]] + [[4,5,6]]	[[1,2,3]+[4,5,6]] = [[1+4,2+5,3+6]]	[[5,7,9]]
[[1,2,3]] + 1	[[1,2,3]+1] = [[1+1,2+1,3+1]]	[[2,3,4]]

In contrast, a pseudo vector is the result of a reduction on a (segmented) vector. A pseudo vector has a context-sensitive semantics: it behaves like a scalar value when used with a vector; it behaves like a vector when used with a scalar value. The point will be made more clear in Section 4.2 when the operational semantics of pseudo vector is discussed.

expression	evaluation	result
/\ [1,2,3]	(*minimum-finding*)	1
/\ [[1,2,3]]	[/\[1,2,3]]	[1]
/\ [[1,2],[3]]	[/\[1,2],/\[3]]	[1,3]
[1,2,3] + 1	[1+1,2+1,3+1]	[2,3,4]
[[1,2],[3]] + [1,3]	[[1,2]+1,[3]+3]	[[2,3],[6]]

4.2 The Operational Semantics of Values

The operational semantics of Linear-C values over binary operation is quite simple: scalar values perform *for-all* semantics while aggregate values perform *for-each* semantics. The rules can be semi-formally described as follows:

(rule 1) $m \oplus n \rightarrow m \oplus n$.

(rule 2) $[a_i] \oplus n \rightarrow [a_i \oplus n]$.

(rule 3) $[a_i] \oplus [b_i] \rightarrow [a_i \oplus b_i]$ when $size([a_i]) = size([b_i])$.

Generally speaking, **(rule 1)** indicates the ordinary case of both operands being scalar values, **(rule 2)** shows how the scalar value operand broadcasts itself to the other aggregate value operand, and **(rule 3)** says that two aggregate value operands are operated in an element-wise manner. It can only be done when "both operands have the same number of elements" to avoid element mismatch. The condition $size([a_i]) = size([b_i])$ denotes such a restriction.

To avoid element mismatch, the *size* of an aggregate value is defined. Intuitively, it is defined as *the number of elements* in the value. An element could be a scalar value or a segment. In other words, it counts one level down, in contrast to the total number of scalars in an aggregate value. In terms of vector and pseudo vector, the size turns out to be (1) the number of segments in a vector, or (2) the number of scalar values in a pseudo vector.

aggregate value	type	size
[1,2,3]	pseudo	3
[[1,2,3]]	vector	1
[[1,2],[3]]	vector	2

The following is a set of Linear-C expressions with their evaluations. Rejections shown in the examples are due to one or more of its elements being rejected.

type	expression	evaluate	result
scalar + scalar	1 + 4		5
pseudo + scalar	[1,2,3] + 4	[1+4, 2+4, 3+4]	[5,6,7]
pseudo + pseudo	[1,2,3] + [4,5,6]	[1+4, 2+5, 3+6]	[5,7,9]
	[1,2,3] + [4,5]		reject (different size)
vector + scalar	[[1,2,3]] + 4	[[1,2,3] + 4]	[[5,6,7]]
	[[1,2],[3]] + 4	[[1,2]+4, [3]+4]	[[5,6],[7]]
vector + pseudo	[[1,2,3]] + [4]	[[1,2,3] + 4]	[[5,6,7]]
	[[1,2],[3]] + [4,5]	[[1,2]+4, [3]+5]	[[5,6],[8]]
	[[1,2],[3]] + [4]		reject (different size)
vector + vector	[[1,2,3]] + [[4,5,6]]	[[1,2,3] + [4,5,6]]	[[5,7,9]]
	[[1,2,3]] + [[4,5]]	[[1,2,3]+[4,5]]	reject
	[[1,2],[3]] + [[4,5],[6]]	[[1,2]+[4,5], [3]+[6]]	[[5,7],[9]]
	[[1,2],[3]] + [[4,5,6]]		reject (different size)
	[[1,2],[3]] + [[4],[5,6]]	[[1,2]+[4], [3]+[5,6]]	reject

It is interesting to note that scalar value 4 and pseudo vector [4] are different, i.e., they have different behaviors.

$\alpha + \beta$	β = 4	β = [4]
α = [[1,2,3]]	[[5,6,7]]	[[5,6,7]]
α = [[1,2],[3]]	[[5,6],[7]]	reject (different size)

Scalar 4 is always compatible with a vector of arbitrary size, whereas pseudo vector [4] may not be compatible. More details can be found in Section 3.2.

4.3 The Kinds of Association with Aggregate Value

Linear-C provides two kinds of association with aggregate value: *activity control* and *segment control*. Activity control determines whether an element will be involved in a computation or not. Segment control splits an aggregate value into several segments. Operators such as reductions will respect these segment boundaries and generate pseudo vector as the result. These unique features of Linear-C are suitable for many situations.

An inactive element has the semantics of *universal identity* (ϕ). It indicates that such an element won't contribute itself to the computation.

$\phi(v)$ denotes scalar v is inactive		
expression	evaluation	result
[ϕ(1),ϕ(2),3] + [ϕ(4),5,6]	[ϕ+ϕ,ϕ+5,3+6]	[ϕ(?),5,9]
[ϕ(1),ϕ(2),3] + 4	[ϕ+4,ϕ+4,3+4]	[4,4,7]
/\[ϕ(1),ϕ(2),3]		3
/\[ϕ(1),ϕ(2),ϕ(3)]		ϕ(?)

Note that, if all operands of a computation are inactive, the result is guaranteed to be inactive (and thus have the semantics of ϕ), but its value is implementation-dependent. Such a case is represented as ϕ(?) in the above illustration.

Segment control partitions elements of an aggregate value into segments; each of them is respected as a single aggregate value. In other words, a segmented aggregate value is considered as a collection of aggregate values.

expression	evaluate	result
[[1,2],[3]] + [[4,5],[6]]		[[5,7],[9]]
/\[[1,2],[3]]	[/\[1,2],/\[3]]	[1,3]
/\[1,2,3]		1

4.4 The Mechanisms for Re-association

Linear-C provides two new binary operators, *activity operator* (?) and *segment operator* (!), for re-associating activity control and segment control with an aggregate value. It is *re-association* because the new control will supersede the old control currently associated. More specifically, the two operators have the form:

$$[k_i : l_i] \oplus [m_i : n_i]$$

where $\oplus$ = ? or !. It simply says, each element of a (pseudo) vector $(m_i : n_i)$ is re-associated with the new corresponding control bit $(k_i : l_i)$. Operator $\oplus$ is used to distinguish the kind of association. In general, both operators have the same manipulation rules as ordinary binary operators.

In the presence of *inactive* control bit $(k_i = 0)$, the corresponding re-association will not be done. In other words, the re-association of $k_i : l_i$ to $m_i : n_i$ obeys the following rule:

$$k_i : l_i \oplus (m_i : n_i) = \begin{cases} l_i : n_i & \text{if } k_i \neq 0 \\ m_i : n_i & \text{if } k_i = 0 \end{cases}$$

In Linear-C, not all kinds of values can be re-associated. For example, re-segmentation of a pseudo vector is not permitted since a pseudo vector does not have segment control. The following table summarizes all the feasible combinations.

	operator ?		operator !	
control bits	pseudo	vector	pseudo	vector
scalar	$\surd$	$\surd$	$\times$	$\surd$
pseudo	$\surd$	$\surd$	$\times$	$\surd$
vector	$\times$	$\surd$	$\times$	$\surd$

4.5 The Use of Re-association

Linear-C supports activity re-association and segment re-association. Activity re-association deactivates or re-activates elements of an aggregate value, while segment re-association splits segments or merges consecutive segments. This section will show how to specify control bits so as to get the desired activation and re-segmentation.

Intuitively, $(0\ ?\ m:n)$ deactivates any activity-controlled scalar $m:n$. In contrast, $(1\ ?\ m:n)$ re-activates it. Similarly, $(0\ ?\ [a_i])$ and $(1\ ?\ [a_i])$ deactivates and re-activates every element (or more precisely, every scalar value) [3]. of $[a_i]$, respectively.

If we want to keep the original activity of any activity-controlled scalar, we can use $(\phi?\ m:n)$. Since this activity association has no effect in replacing the activity indicator (m) of scalar $m:n$, after the association, the activity-controlled scalar will keep the same.

The following has five examples, each of which has the input (column "before") and the desired output (column "after") for an activity association. We intend to find out what activity we should supply so that after the association of this activity with "before" we can get the desired output "after".

aggregate value		control bits		
before	after	vector	pseudo	scalar
$[[-],[-,-]]$	$[[-],[-,\phi(-)]]$	$[[\phi],[\phi,0]]$	$\times$	$\times$
	$[[-],[\phi(-),\phi(-)]]$	$[[\phi],[0,0]]$	$[\phi,0]$	$\times$
	$[[\phi(-)],[\phi(-),\phi(-)]]$	$[[0],[0,0]]$	$[0,0]$	0
$[-,-,-]$	$[-,\phi(-),-]$	$\times$	$[\phi,0,\phi]$	$\times$
	$[\phi(-),\phi(-),\phi(-)]$	$\times$	$[0,0,0]$	0

In general, we can specify the activity by examining the differences between the input and the desired output. For example, the difference between $[[-],[-,-]]$ and $[[-],[-,\phi(-)]]$ is at the second element of the second segment: it is inactive afterwards. Therefore, we keep all the other elements the same except that we deactivate this particular element by ϕ. And activity $[[\phi],[\phi,0]]$ is exactly this intention. If all the elements in a segment are of the same value, we can use the scalar of the same value instead. It is essentially an abbreviation.

Re-segmentation is a little complicated *visually* but still uses the same approach. Expression $(1\ !\ m:n)$ starts a new segment from $m:n$. In contrast, $(0\ !\ m:n)$ concatenates $m:n$ with the previous segment (or element). When applied to segmented values, $(1\ !\ [a_i]$ makes every element a_i as a new segment, i.e., $[[a_1],\ldots,[a_n]]$. On the other hand, $(0\ !\ [a_i])$ combines all elements a_i in one segment.

In the following we give six examples for re-segmentation. Again, we examine the differences between the input and the desired output, and use the above rules to get the correct segment controller.

aggregate value		control bits		
before	after	vector	pseudo	scalar
$[[-,-,-]]$	$[[-],[-,-]]$	$[[\phi,1,\phi]]$	$\times$	$\times$
	$[[-],[-],[-]]$	$[[\phi,1,1]]$	$[1]$	1
$[[-],[-,-]]$	$[[-],[-],[-]]$	$[[\phi,\phi,1]]$	$[\phi,1]$	1
$[[-],[-],[-]]$	$[[-],[-,-]]$	$[[\phi],[\phi],[0]]$	$[\phi,\phi,0]$	$\times$
	$[[,,]]$	$[[\phi],[0],[0]]$	$[\phi,0,0]$	0
$[[-],[-,-]]$	$[[-,-,-]]$	$[[\phi],[0,\phi]]$	$[\phi,0]$	0

[3]The intuition follows directly from the semantics of ordinary binary operators. See Section 4.2 for details.

4.6 Collective Operators

Linear-C provides a set of unary operators, called *collective operators*, to manipulate a single aggregate value in a full or partial summarization manner. Collective operators can be further classified as *reductions*, *scans*, and *reverse scans* (abbreviated as *r-scans*). The operator `/\`, for example, is a reduction operator for summarizing an aggregate value by extracting its minimum element.

The syntax of collective operators contains two parts:

<red,scan,r-scan> <summarization>

<red,scan,r-scan> indicates that it is either a reduction, a scan, or a r-scan; <summarization> indicates how it is summarized. Take `/\` as an example. Its prefix, `/`, says it is a reduction, and its postfix, `\`, says it is a minimum-finding summarization. The full set of collective operators is summarized as follows.

- <red,scan,r-scan>: `/` (reduction), `>` (scan), `<` (r-scan).
- <summarization>: `+` (sum), `&` (bitwise and), `|` (bitwise or), `^` (bitwise exclusive-or), `&&` (logical and), `||` (logical or), `^^` (max), `\` (min), `<` (left), `>` (right).

Reduction operators, like the other kinds of operators, respect the segment boundaries of their operands. In other words, when applied to a vector, the reduction results in a pseudo vector; when applied to a pseudo vector, the result is a scalar value. Reductions also form the basis of scans and r-scans:

$$> \oplus([a_1, a_2, \ldots, a_n]) \rightarrow [/ \oplus ([a_1]), / \oplus ([a_1, a_2]), \ldots, / \oplus ([a_1, \ldots, a_n])]$$

where $\oplus$ represents <summarization>. R-scan is defined similarly but in a reverse direction. Note that scan and r-scan defined here are both in inclusive form [4]. Linear-C provides a set of shift operators to get the exclusive form from its inclusive counterpart. Shift operators will be discussed in the next section.

The following is a set of Linear-C expressions showing the semantics of collective operation. Two points are worth mentioning. First, reduction, scan, or r-scan on scalar value is not allowed. Second, $/ \oplus ([])$ is defined as $\phi(?)$, so is the case when all elements of the operand are inactive.

type	expression	evaluate	result
red (scalar)	/+(2)		reject
red (pseudo)	/+([])		ϕ(?)
	/+([1,2,3])	1+2+3	6
	/+([1,ϕ(2),3])	1+ϕ+3	4
	/+([ϕ(1),ϕ(2),ϕ(3)])	$\phi+\phi+\phi$	ϕ(?)
red (vector)	/+([[1,2,3],[4,5]])	[/+([1,2,3]),/+([4,5])]	[6,9]
scan (pseudo)	>+([1,2,3])	[/+([1]),/+([1,2]),/+([1,2,3])]	[1,3,6]
scan (vector)	>+([[1,2,3],[4,5]])	[>+([1,2,3]),>+([4,5])]	[[1,3,6],[4,9]]
r-scan (pseudo)	<+([1,2,3])	[/+([1,2,3]),/+([2,3]),/+([3])]	[6,5,3]
r-scan (vector)	<+([[1,2,3],[4,5]])	[<+([1,2,3]),<+([4,5])]	[[6,5,3],[9,5]]

[4] The reason inclusive form is chosen is that exclusive form of scan and r-scan "lose" information about the "last" element.

4.7 Shift Operators

Shift operators are collective operators which shift aggregate values back and forth in an element-wise manner. They deserve special attention for two reasons, one syntactically and the other semantically: (1) It is not intuitively simple to identify their behaviors from their syntactic patterns; (2) shift operators use *exclusive form* as their semantic basis.

$$> \oplus([a_1, a_2, \ldots, a_n]) \rightarrow [/ \oplus ([]), / \oplus ([a_1]), / \oplus ([a_1, a_2]), \ldots, / \oplus ([a_1, \ldots, a_{n-1}])]$$

The first reason comes from the introduction of two new <summarization>'s: < (left) and > (right). The second reason distinguishes them from ordinary (arithmetic) collective operators described previously. In Linear-C, six shift operators are supported.

summarization	/ (red)	> (scan)	< (r-scan)
< (left)	/< (leftmost active)	>< (broadcast right)	<< (left shift into)
> (right)	/> (rightmost active)	>> (right shift into)	<> (broadcast left)

The two reductions are simple: operator /< gets the leftmost active element of the aggregate value operand; operator /> gets the rightmost active one. Therefore, for example, $[\phi(1), 2, 3]$ has the leftmost active element 2 and rightmost active element 3. And the equations $/<[\phi(1), 2, 3] = 2$ and $/>[\phi(1), 2, 3] = 3$ are true. As usual, reduction on a scalar value is not allowed. Reduction on [] or on (pseudo) vector with all inactive elements has the result of $\phi(?)$.

expression	result
/>([1,2,3])	1
/<([ϕ(1),2,3])	2
/<([ϕ(1),ϕ(2),ϕ(3)])	$\phi(?)$
/<([])	$\phi(?)$
/>([1,2,3])	3
/>([ϕ(1),2,3])	3
/<([ϕ(1),ϕ(2),ϕ(3)])	$\phi(?)$
/<([])	$\phi(?)$

Operators, <> and ><, broadcast [5] a certain extreme element to their left or right. Operator >< broadcasts its leftmost active element to the right. Similarly, operator <> broadcasts its rightmost active element to the left. If we follow the semantics, what $><[a_i]$ really does is:

$$><[a_1, \ldots, a_k, a_{k+1}, \ldots, a_n] \rightarrow [\phi(?), \cdots, \phi(?), a_k, \cdots, a_k]$$

it finds out the leftmost active element a_k, broadcasts it to the right over all active and inactive elements, and updates the left of it and itself with $\phi(?)$. Therefore, the result for $><[\phi(1), 2, 3]$ should be $[\phi(?), \phi(?), 3]$ since 2 is the leftmost active element.

[5] The correct term should be *shift-broadcast* since the broadcasted element itself will be destroyed as $\phi(?)$. This unnatural interpretation is due to the exclusive form semantics.

expression	evaluate	result
><([1,2,3])	[/<([]),/<([1]),/<([1,2])]	[ϕ(?),1,1]
><([ϕ(1),2,3])	[/<([]),/<([ϕ(1)]),/<([ϕ(1),2])]	[ϕ(?),ϕ(?),2]
><([ϕ(1),ϕ(2),ϕ(3)])		[ϕ(?),ϕ(?),ϕ(?)]
><([ϕ(1),ϕ(2),3,4,ϕ(5),6,ϕ(7)])		[ϕ(?),ϕ(?),ϕ(?),3,3,3,3]

Note that, in operators >< (similar in <>), (1) the broadcast starts from the element *next* to the leftmost active element, (2) all elements left to the leftmost active element are replaced by ϕ(?). These two situations can be avoided in Linear-C by the combination of shift operators and activity-controlled assignment.

Operators << and >> are used for element-wise shift. When all elements are active, operator << does the usual element shift, as in C, by shifting all its elements to their left. Similarly, operator >> shifts all the elements to their right. Things get a little strange when some of the elements are inactive. According to the semantics, those two operators will *broadcast* the to-be-shifted element over inactive elements until it is "blocked" by another active element.

$$>>[\ldots, n_i, \phi(n_{i+1}), \cdots, \phi(n_j), n_{j+1}, \ldots] \rightarrow [\ldots, -, n_i, \cdots, n_i, n_i, \ldots]$$

Element n_i is active and is to be shifted to the right. Since the elements from $\phi(n_{i+1})$ to $\phi(n_j)$ are all inactive, they are all replaced by the element n_i. Finally, the element n_i is "blocked" by active element n_{j+1}. It replaces n_{j+1} and stops broadcasting. Element n_{j+1} then starts another round of shift-broadcast.

expression	result
>>([1,2,3,4,5,6,7])	[ϕ(?),1,2,3,4,5,6]
<<([1,2,3,4,5,6,7])	[2,3,4,5,6,7,ϕ(?)]
>>([1,ϕ(2),3,ϕ(4),ϕ(5),6,ϕ(7)])	[ϕ(?),1,1,3,3,3,6]
>>([1,ϕ(2),ϕ(3),ϕ(4),ϕ(5),ϕ(6),ϕ(7)])	[ϕ(?),1,1,1,1,1,1]

It may be noticed that shift operators have exotic behaviors. They are due to their *skip-into* and *exclusive-form* semantics. Section 3.7 has a detailed discussion about it.

4.8 Summary

This section gives an overview of data parallel capability of Linear-C. Four main issues are examined in details.

issue	findings in this section
values	scalars, vectors, pseudo vectors
operational semantics	for-all/for-each semantics
associations	activity control, segment control
new operators	activity/segment operator, collective operators

Linear-C supports three kinds of values: scalar values, vectors, and pseudo vectors. Scalar values follow the for-all semantics while vectors have the for-each semantics. Depending on the surrounding

context, pseudo vectors may behave either like scalar values or vectors. Activity control and segment control can be attached onto aggregate values. Segment control gives the power of treating a vector as a collection of vectors. It is also the main distinction between vectors and pseudo vectors. A fixed set of reductions, scans, and reverse scans, called collective operators, are introduced. They are new operators for the manipulation of a single aggregate value. Shift operators, a distinguished subset of collective operators, are mainly used for moving aggregate values around. They are different from other collective operators in their exclusive-form definitions.

5 Conclusion and Future Research

In this report, we first introduce the basic elements of Linear-C and their notations. Starting from simple scalar values, activity-controlled scalar values, pseudo vectors, and vectors are specified one by one. We also describe Linear-C operators for handling pseudo vectors and vectors. One category is the set of the extended C binary operators, and the other category is the set of new unary operators. After the introduction, the rationale behind language design, especially the semantics of operators, are discussed. Each problem is described and the reasons for the decision-making are provided. Finally, the whole language is specified in a systematic manner.

In general, Linear-C is a C extension capable of expressing and manipulating many kinds of data parallelism. It has a simple yet powerful operational semantics. For the user, Linear-C is easy to learn, use, and debug. Furthermore, it expresses data parallelism explicitly, resulting in a concise yet informative program representation. On the implementation side, Linear-C primitives are supported efficiently by the CAM-2000 hardware, which implements data parallelism expressed in Linear-C effectively.

Future researches in Linear-C can proceed into two coupled areas: one is the refinement of the language, and the other is the code optimization of current definition. In the current implementation the naively compiled code incurs high software costs, both in size and in time. Some improvement may be accomplished through the language refinement, but a large amount of it depends on the optimization procedure. The balance between design choices and between implementation and design must be maintained to get the maximum benefit.

References

[1] Hall, J.S. (1994). *Associative Processing: Architectures, Algorithms, and Applications*, Ph.D. Dissertation, Computer Science Department, Rutgers University.

[2] Hsu, C.H., Smith, D.E., and Levy, S. (1996). *A Linear-C Implementation of Dijkstra's Algorithm*, LCSR-TR-274, Computer Science Department, Rutgers University.

[3] Smith, D.E., Hall, J.S., and Miyake, K. (1993). *Rutgers' CAM 2000 Chip Architecture*, LCSR-TR-196, Computer Science Department, Rutgers University.