A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*

William Landi†    Barbara G. Ryder‡

Abstract

Aliasing occurs at some program point during execution when two or more names exist for
the same location. In a language which allows pointers, the problem of determining the set
of pairs of names at a program point which may refer to the same location during program
execution is \textit{NP}-hard. We present an algorithm which safely approximates Interprocedural
May Alias in the presence of pointers. This algorithm has been implemented in a prototype
analysis tool for C programs.

---

*The research reported here was supported, in part, by Siemens Research Corporation and NSF grant CCR-
8920078.

†Department of Computer Science, Rutgers University, New Brunswick, NJ 08803
Contents

1 Introduction 3

2 Problem Representation 6
   2.1 Interprocedural Control Flow Graph .......................... 6
   2.2 Types ......................................................... 6
   2.3 Object Names ................................................. 8
   2.4 Aliases ....................................................... 10
   2.5 Some Terminology ......................................... 11

3 The may-hold Relation 12
   3.1 Conditional May Alias: Definition .......................... 12
   3.2 Representation .............................................. 14
   3.3 Alias Consequences ....................................... 15
   3.4 Modeling Parameter Bindings ............................... 16
      3.4.1 Computing bind(call,θ) ................................ 18
      3.4.2 Computing bind(call,(x,y)) ......................... 18

4 Computing may-hold 20
   4.1 Aliases_introduced_by_assignment(node) ................. 20
   4.2 Aliases_introduced_by_call(node) ......................... 24
   4.3 Alias_at_call_implies(call,assumed_alias,possible_alias) ........................................ 24
   4.4 Alias_at_exit_implies(exit,assumed_alias,possible_alias) .............................. 32
   4.5 Any_other_alias_implies(node,assumed_alias,possible_alias) .................. 33

5 May Alias 39

6 Conclusion 41

Appendix A : Dictionary of Functions 42

Appendix B : Approximate-when-both-non_visible 45

Appendix C : Alias_at_call_implies 47

Appendix D : Alias_at_exit_implies 48

Appendix E : Any_other_alias_implies 49

References 51
1 Introduction

Aliasing occurs at some program point during program execution when two or more names exist for the same location. The aliases of a name at a program point \( t \) are all other names that may refer to the same memory location on some execution path to \( t \). When this execution path traverses more than one procedure, we are solving the Interprocedural May Alias Problem. While the calculation of aliases for FORTRAN is well understood [2, 6, 7, 17], if general pointers are added as a language construct, the problem of computing aliases becomes \( \mathcal{NP} \)-hard and no good approximation algorithms exist. Moreover, aliases complicate most data flow analysis problems, and the absence of alias information can prevent many optimizations.

Comparison to FORTRAN Aliasing In FORTRAN, the only dynamic method for creating aliases is through the use of reference formals. Solving for aliasing in the presence of pointers presents several complications. The most obvious difference concerns reference formals; aliases that hold at invocation of a procedure, hold during the entire execution of the called procedure; however, when pointers are present, this is not the case, because pointer assignments can change aliases during execution of the called procedure. In addition, with reference formals, a call to a procedure cannot affect the aliases in the calling procedure, but if pointers are present, this is no longer true. Both of these facts indicate that existing FORTRAN alias algorithms are not extensible to handle pointers.

Related Work Weihl devised an algorithm for finding aliases in the presence of pointers [19, 20]. Unfortunately his algorithm is very imprecise. In fact on a simple program (straight line code) which has \( O(n) \) aliases, where \( n \) is the number of variables in the program, Weihl reports \( O(n^2) \) aliases[14]. We have solved the modification side effect problem on several C programs using Weihl's algorithm to determine the aliases. Our analysis generally reports that almost all variables could be modified. This is because Weihl's algorithm does not give information of sufficient precision to be useful for this problem. Some empirical observations on the precision of Weihl's algorithm can be found in [14].

Chow and Rudnik [4] also presented an algorithm for finding aliases in the presence of pointers. Their algorithm suffers because they treat interprocedural alias as an intraprocedural problem; that is, they do not consider that when a procedure returns, control passes to the call site which invoked
it, and they handle local variables incorrectly. Benjamin Cooper [5] has developed an algorithm which uses explicit path information in the form of alias histories to insure (for interprocedural paths) that a procedure returns to the call site that invoked it. This method seems unsuitable for an implementation and is reminiscent of the work of Sharir and Pnueli [18].

A related area of research is the work done by the compiling community on conflict detection in recursive structures [3, 8, 9, 11, 16]. A conflict occurs between two statements when one statement writes a location and the other accesses (reads or writes) the same location, thus preventing the possibility of those statements being executed in parallel (i.e., in arbitrary order). Thus, aliasing occurs at a program point, and conflicts, in general, occur between two or more program points.

Overview In this paper we present a detailed description of our algorithm for finding a safe approximation for Interprocedural May Alias in the presence of general pointers and call by value parameter passing. For efficiency, our algorithm is demand driven (i.e., we calculate information only for aliases that hold on some path) and must maintain data structures for quickly accessing certain kinds of information about the alias solution. For safety, our algorithm must account for various sources of approximation, and never miss an alias that occurs on some path.

There must be at least four distinct approximations in any practical alias algorithm for pointers. In any program that contains recursive data structures, there are a potentially infinite number of objects which can have aliases. Any aliasing algorithm will have to represent all possible objects by a finite (polynomial) number of objects. We use a system of k-limiting similar to that defined by Jones and Muchnick [10].

There is a second source of approximation illustrated by the following scenario. Suppose there is an assignment $p = x$ at program point $t$, alias pair $\langle p, q \rangle$ holds on some path\(^1\) to an immediate predecessor of $t$ and $\langle *x, *y \rangle$ also holds on some path to an immediate predecessor of $t$. Does $\langle *q, *y \rangle$ hold on some path to $t$?

$$
\begin{align*}
\langle p, q \rangle & \quad \langle *x, *y \rangle \\
t: & \quad (p = x) \\
\langle *q, *y \rangle & ?
\end{align*}
$$

If both $\langle p, q \rangle$ and $\langle *x, *y \rangle$ occur on the same path, then $\langle *q, *y \rangle$ holds on that path extended by $t$;

\(^1\)Remember that holds is defined after execution of the last statement on the path.
our algorithm safely concludes this, even though it may not be true.

The third source of approximation is similar to the second. Consider the assignment \( p = x \) at program point \( t \). Suppose \( \langle p, q \rangle \) holds on some path to an immediate predecessor of \( t \) and \( \langle *q, *z \rangle \) holds on some path to an immediate predecessor of \( t \). Does \( \langle *q, *z \rangle \) hold on some path to \( t \)?

\[
\begin{array}{c}
\langle p, q \rangle \\
\text{t: } p = x \\
\langle *q, *z \rangle \\
\end{array}
\]

If on at least one path to an immediate predecessor of \( t \) \( \langle *q, *z \rangle \) holds, and neither \( \langle p, q \rangle \) nor \( \langle p, z \rangle \) does, then \( \langle *q, *z \rangle \) holds on that path extended by \( t \). However, if on all those paths \( \langle *q, *z \rangle \) and \( \langle p, q \rangle \) both hold, then \( \langle *q, *z \rangle \) does not necessarily hold on any path to \( t \). In this case, however, our algorithm safely assumes that \( \langle *q, *z \rangle \) holds on some path to \( t \).

The fourth involves two distinct aliases of the LHS of an assignment.

\[
\begin{array}{c}
\langle p, *u \rangle \\
\text{t: } p.n = v \rightarrow n \rightarrow n; \\
\langle *u, *n \rangle, \langle *v, *n \rangle \\
\end{array}
\]

Normally, \( \langle *u \rightarrow n \rangle, \langle *v \rightarrow n \rightarrow n \rangle \) should hold on a path to \( t \) because assigning \( v \rightarrow n \rightarrow n \) to \( p.n \) is also an assignment to \( u \rightarrow n \) on the path on which \( \langle p, *u \rangle \) holds. This, however, is not necessarily the case. If, for example, on the same path \( \langle p, *v \rangle \) holds then \( \langle *u \rightarrow n \rangle, \langle *v \rightarrow n \rightarrow n \rangle \) does not necessarily hold:

\[
\begin{align*}
\text{node:} & \quad u \rightarrow p \rightarrow v \\
\text{succ:} & \quad p.n = v \rightarrow n \rightarrow n;
\end{align*}
\]

This fourth kind of approximation is different than the other three in that it requires either recursive data structures or type casting to occur. Even in those cases, it seems likely that this approximation will rarely occur.
The main ideas of our algorithm are highlighted in the text and the interested reader can find pseudo-code for the algorithm in the figures and Appendices. Our algorithm is program-point-specific and thus more precise than Weihl's algorithm[19]. We currently have a prototype implementation in C for analyzing C programs.

2 Problem Representation

The Source Language  Our algorithm finds aliases for a language that is a subset of C. The main attributes of C that are not handled by our algorithm are: union types, nested structures, casting\(^2\), pointers to functions, and exception handling. We do allow arrays and pointer arithmetic, however we deal with these on a very simple and naive level.

2.1 Interprocedural Control Flow Graph

We represent programs by interprocedural control flow graphs, (ICFGs) originally presented in [15]. An ICFG is intuitively, the union of the control flow graphs (CFGs)\(^3\) [1] for each procedure, with calls connected to the procedures they invoke. Formally, an ICFG is a triple \((\mathcal{N}, \mathcal{E}, \rho)\) where: \(\rho\) is the entry node for \textit{main}; \(\mathcal{N}\) contains one node for each statement in the program, an \textit{entry} and \textit{exit} node for each procedure, a \textit{call} and \textit{return} node for each call site; and \(\mathcal{E}\) contains all edges in the CFG for each procedure, with a slight modification of edges involving call sites. In the ICFG, a call site is split into a \textit{call} and a \textit{return} node. An intraprocedural edge into a call node represents execution flow into a call site, while an intraprocedural edge out of a return node represents flow from a call site. In addition to the intraprocedural edges, two interprocedural edges are added for each call site: one from the call node to the entry node of the invoked procedure, and one from the exit node of the procedure to the return node of the call site. See Figure 1 for an example of an ICFG.

2.2 Types

An object is a location that can store information (for example, variables). Objects in C have types, and we need to perform a handful of operations on types. They are all straightforward, but

\(^2\)Casting of \textit{malloc()} is not a problem for our implementation.

\(^3\)Each node in our CFG is a source code statement.
int *q;
void A(f)
    int *f;
    { q = f; }
main()
    { int *q, *r;
      r = null;
      A(q);
      r = q;
      A(p);
    }

Figure 1: A C program and its ICFG
will be listed here to avoid confusion:

**address_type**(*type*) returns the type of the objects which can point to *type*.

**can_deref**(*type*) is true iff *type* can be legally dereferenced without casting.

**deref_type**(*type*) If *type* can be dereferenced, **deref_type** returns the type of the objects to which *type* may point.

**field_type**(*type*,*field*) If *type* is a structure with *field*, **field_type** returns the type of *field*.

**is_field_off**(*field*,*type*) is true iff *type* is a structure type and *field* is a legal field of *type*.

**is_struct**(*type*) is true iff *type* is a structure.

### 2.3 Object Names

Because objects are locations that can store information, in (terminating) programs containing recursive data structures, there are arbitrarily many potentially addressable objects. For example, in a program with a linked list *l1ist, l1ist(->next)*" are all possible names for distinct run-time objects. Thus, any practical alias algorithm will have to represent the set of all possible objects and the alias relationships between those objects with a (small) finite data structure. We use a solution that is roughly analogous to *k-limited* as defined by Jones and Muchnick[10]. Less naive schemes have been developed [3, 8, 9, 16], but we have yet to examine their suitability for our purposes.

Object names provide ways to refer to objects in a program. An object name is a variable and a (possibly empty) sequence of dereferences and field accesses. A syntax-directed definition[1] for object names is in Figure 2. Because we have prohibited casting, we are only interested in object names which do not have type *error*.

If there are any recursively defined data structures (for example, linked lists) then the number of object names that do not have type *error* is infinite. However, we can not deal with an infinite number of object names in a finite amount of time, so we will limit to some constant, *k*, the number of applications of the rule 1 of Figure 2. This will restrict the number of objects names which do not have type *error* that can be generated to a finite number \[O(number\_vars\ast(max\_number\_fields\_in\_one\_struct)^k)\]. This raises the issue of how to deal with
<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| 1    | OBJECT₁ → *(OBJECT₂) | if can_deref(OBJECT₂.type) 
then OBJECT₁.type := deref.type(OBJECT₂.type) 
else OBJECT₁.type := error |
| 2    | OBJECT₁ → (OBJECT₂).field† | if is_field_of(field,OBJECT₂.type) 
then OBJECT₁.type := field.type(OBJECT₂.type,field) 
else OBJECT₁.type := error |
| 3    | OBJECT → var‡ | OBJECT.type := type of var |

† field is any declared field of any structure in the program.
‡ var is any variable declared in the program.

Figure 2: Syntax-directed definition for object names for a given program

object names with more than \( k \) dereferences. We do this rather simplistically by considering any object name with \( l > k \) dereferences to be represented by the object name obtained by ignoring the last \( l - k \) dereferences yielding a unique \( k \)-limited name. Thus, for \( k = 1 \), \( p \rightarrow f₁ \rightarrow f₂ \) would be represented by \( p \rightarrow f₁ \) (and not by \( \ast p \)). We will borrow Jones and Muchnick[10] terminology and call this \( k \)-limiting, even though they \( k \)-limit dynamic structures while we \( k \)-limit object names because the two processes are analogous.

Unfortunately, despite being interested in aliases between object names, there is one construct which is not an object name but can affect alias information, the address operator (for example, \&\( p \)). Thus, we will define a class object.name which is either an object name or an \&(object.name).

To state the algorithm concisely and accurately in pseudo-code, we need the following non-trivial functions for class object.name:

\[
is\_prefix(object\_name₁, object\_name₂) \text{ returns } true \text{ iff } object\_name₁ \text{ can be transformed into } object\_name₂ \text{ by a (possibly empty) sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2).}
\]

\[
is\_prefix\_with\_dereference(object\_name₁, object\_name₂) \text{ returns } true \text{ iff } object\_name₁ \text{ can be transformed into } object\_name₂ \text{ by a sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2) with AT LEAST ONE dereference.}
\]
apply_trans(object_name₁, object_name₂, object_name₃): object_name₁ and object_name₃ must have the same type and is_prefix(object_name₁, object_name₂) must be true. The function applies to object_name₃ the sequence of dereferences and field accesses necessary to transform object_name₁ into object_name₂. It returns true iff any dereferences occurs somewhere in the sequence. Some examples of apply_trans can be found in Figure 3.

We will also use the following, fairly straightforward functions:

amp_object_name(object_name) is & (object_name), if object_name does not start with an &, otherwise error.

deref(object_name) removes the & if object_name starts with an & and otherwise returns *(object_name). This corresponds to rule 1 of Figure 2.

field_access(object_name, field_name) is (object_name).field_name, if object_name does not start with an &, otherwise error. This corresponds to rule 2 of Figure 2.

object_type(object_name) is the data type which can be stored in objects with object_name, if object_name does not start with an address operator. If for some name, object_name = &name then it is address_type(name).

is_k-limited(object_name) is true iff object_type(object_name) can be dereferenced, object_name contains k dereferences, and object_name does not start with an address operator.

simple_object_name(variable) is the object name which is simply variable, i.e. rule 3 of Figure 2.

2.4 Aliases

An alias occurs when two or more object names refer to the same location at some point during program execution. As in [15] we will represent aliases by unordered pairs of object names (for example, ⟨v, *p⟩). The order is unimportant because the alias relation is symmetric. While for the single level pointer case this is adequate, now because of k-limiting of object names it is not. Since we have k-limited object_names, in order to safely represent aliases, we must assume that an

---

4This is the same as the value of the type attribute for object_name obtained from Figure 2.
<table>
<thead>
<tr>
<th>object_name_1</th>
<th>object_name_2</th>
<th>initial object_name_3</th>
<th>final object_name_3</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>*p</td>
<td>&amp;q</td>
<td>q</td>
<td>true</td>
</tr>
<tr>
<td>p-&gt;next</td>
<td>p-&gt;next-&gt;next</td>
<td>&amp;q</td>
<td>q.next</td>
<td>true</td>
</tr>
<tr>
<td>*(p-&gt;next)</td>
<td>p-&gt;next-&gt;data</td>
<td>q</td>
<td>q.data</td>
<td>false</td>
</tr>
<tr>
<td>*(p-&gt;next)</td>
<td>p-&gt;next-&gt;data</td>
<td>*r</td>
<td>r-&gt;data</td>
<td>false</td>
</tr>
<tr>
<td>p-&gt;next</td>
<td>p-&gt;next-&gt;data</td>
<td>r</td>
<td>r-&gt;next</td>
<td>true</td>
</tr>
<tr>
<td>p-&gt;next</td>
<td>p-&gt;next-&gt;next</td>
<td>q.next</td>
<td>(q.next)-&gt;next</td>
<td>true</td>
</tr>
<tr>
<td>p-&gt;next</td>
<td>p-&gt;next-&gt;next</td>
<td>r-&gt;next</td>
<td>r-&gt;next-&gt;next</td>
<td>true</td>
</tr>
</tbody>
</table>

Figure 3: Some examples of apply_trans(object_name_1, object_name_2, object_name_3)

alias \langle a, b_k \rangle with k-limited component \( b_k \), represents not only the alias \langle a, b_k \rangle but also any alias \langle a, b'_k \rangle such that \( b_k \) can be transformed into \( b'_k \) by a sequence of dereferences and field accesses (i.e., is_prefix(\( b_k, b'_k \)) = true). Also an alias \langle a_k, b_k \rangle with two k-limited components, represents not only itself, but all aliases \langle a'_k, b'_k \rangle such that \( a_k \) is a prefix of \( a'_k \) and \( b_k \) is a prefix of \( b'_k \).

An alias \langle a, b \rangle represents the fact that \( a \) and \( b \) refer to the same object. In C objects have types and since we are not allowing casting it would seem that the type of the object would be the type of both \( a \) and \( b \). However, with k-limiting this is not necessarily true. Say there is an alias, at a program point, \langle x, *y \rangle where \( x \) is type “int” and \( y \) is type “int **”. If we are k-limiting with \( k = 1 \) this alias is represented by \langle x, *y \rangle. We use the function alias_type to determine the type of the object represented by an alias.

alias.type(alias) is the type of the object which is referred to by both object names in alias. This function correctly accounts k-limiting.

2.5 Some Terminology

The following are definitions which will be used throughout the paper:

program point: We use program point to refer to the part of a program represented by an ICFG node.

realizable: A path is realizable iff it is a path in the ICFG and whenever a procedure on this path returns, it returns to the call site which invoked it.
holds: Alias \( \langle a, b \rangle \) holds on the realizable path \( \rho n_1 n_2 \ldots n_i \) iff \( a \) and \( b \) refer to the same location after execution of program point \( n_i \) whenever the execution sequence defined by the path occurs. Aliases are symmetric, that is \( \langle a, b \rangle \) holds on a path iff \( \langle b, a \rangle \) also holds on that path; we will not distinguish between \( \langle a, b \rangle \) and \( \langle b, a \rangle \).

visible: At a call site, an object name (for example, \( \star x \)) of the calling procedure is visible in the called procedure iff the called procedure is in the scope of the object name and at run time the object name refers to the same object in both the calling and called procedure. This means that if \( x \) is a local variable of procedure \( P \), then the \( x \) in \( P \) before a recursive call is not visible after the call, since at execution time it is a different instantiation.

3 The may-hold Relation

3.1 Conditional May Alias: Definition

In [15], we present a two-step algorithm for finding Interprocedural May Alias on programs that have single level pointers as the only mechanism for creating aliases. The first step of this algorithm is to compute Conditional May Alias which is defined as the answer to the question: If there is a path to the entry node of the procedure containing \( n_i \) on which every alias in the set \( \mathcal{AA} \) holds, then may object name \( a \) be aliased to object name \( b \) on some path to \( n_i \)? We represent the answer to this question by the boolean relation \( \text{holds}([n_i, \mathcal{AA}, \langle a, b \rangle]) \). Fortunately, it is only necessary to consider \( \mathcal{AA} \) (sets of aliases) with cardinality less than or equal to one[13]. In the second step, we use Conditional May Alias to solve for May Alias using a simple fixed point calculation. This algorithm is precise under the standard assumptions of data flow analysis (i.e., up to symbolic execution).

We will use the single level pointer algorithm for determining May Alias as the basis of a safe Interprocedural May Alias algorithm for programs with general pointers. There are two major obstacles to this goal. The first obstacle is one of precision and safety. Once the restriction to single level pointers is lifted, the assumption that it is only necessary to consider sets of assumed aliases (\( \mathcal{AA} \)) of size less than or equal one is no longer valid. However, if some alias \( \mathcal{PA} \) depends on an \( \mathcal{AA} \) with multiple assumed aliases, then all of them are sufficient and any single assumption necessary. Thus any single assumption can safely be considered as the only assumption. Thus,
we will continue to deal with sets of assumed aliases of size one or less, but this will now be an approximation.

The second obstacle is one of practicality. As the algorithm is described in [15], we must compute $\text{holds}((\text{node}, \mathcal{A}_A), \mathcal{P}_A)$ for every possible triple: node in the ICFG, $\mathcal{A}_A$, and $\mathcal{P}_A$. So for a program with $O(n)$ ICFG nodes and $O(v)$ variables (and thus $O(v^2)$ different possible $\mathcal{A}_A$ and $O(v^2)$ possible $\mathcal{P}_A$), this means a minimum of $O(n \cdot v^4)$ work for even just the single level pointer case$^5$.

To insure efficiency, we will present our algorithm differently than in [15]. We will calculate $\text{holds}$ in a demand driven fashion so that we will only concern ourselves with $\text{holds}((\text{node}, \mathcal{A}_A), \mathcal{P}_A)$ which have the value $\text{true}$. Since most of the holds relation will have the value $\text{false}$, this will improve the average time complexity of our algorithm immensely, although unfortunately, it does not help the worst case complexity.

We also improve efficiency by changing our definition of Conditional May Alias from:

If there is a path to the entry node of the procedure containing $n_i$ on which the assumed alias $\mathcal{A}_A$ holds, then there is a path to $n_i$ on which $\langle a, b \rangle$ holds.

to:

If there is a path to the entry node of the procedure containing $n_i$ on which every alias in the set $\mathcal{A}_A$ holds, then there is a path to $n_i$ on which $\langle a, b \rangle$ holds and there is a path to the entry node of the procedure containing $n_i$ on which the assumed alias set $\mathcal{A}_A$ holds.

Since we have changed the definition of Conditional May Alias, we will change our notation: $\text{may-hold}([n_i, \mathcal{A}_A], \langle a, b \rangle)$ will be used to represent the answer to the new Conditional May Alias question. Formally:

$\text{may-hold}([n_i, \mathcal{A}_A], \langle a, b \rangle)$ is true iff $\langle a, b \rangle$ holds on some path from $\text{entry}(n_i)$, the entry of the procedure containing $n_i$, to $n_i$ assuming there is a path from entry of main to $\text{entry}(n_i)$ on which the assumed alias $\mathcal{A}_A$ holds and there is a path from entry of main to the $\text{entry}(n_i)$ on which $\mathcal{A}_A$ holds.

---

$^5$Actually, when you consider the work necessary at return nodes the complexity is $O(n \cdot v^6)$
We have also introduced a minor amount of imprecision in the algorithm in order to increase efficiency and understanding. Consider the statement, “$p = q$”. We assume that $(p, q)$ holds on any path through “$p = q$” but this is only valid if $q$ is not $NULL$. For example, if “$q = NULL$” was the only immediate predecessor of “$p = q$” then $(p, q)$ would not hold on any path to “$p = q$”. We consider the assumption that $q$ will be non-$NULL$ on some path to “$p = q$” reasonable in practice, and thus use it in our approximate algorithm. In general, if the existence of an alias depends on the fact that some object name is non-$NULL$ on some path, we assume, without verification, that it is.

3.2 Representation

In order to have an efficient implementation for our alias algorithm, we must be able to do the following operations in constant time:

- Set $may\cdot hold([[node, AA], PA])$ to $false$ for all possible node, AA, and PA.
- Find the value of $may\cdot hold([[node, AA], PA])$ for a given node, AA, and PA.
- Set the value of $may\cdot hold([[node, AA], PA])$ for a given node, AA, and PA.

One solution to this problem is to use a constant time initialization array. We do not do this because, even for single level pointers, this would require at least $O(n * v^4)$ space.

The solution we choose is to do dynamic hashing. We use the dynamic hashing scheme presented in [12] (except we did not implement table shrinking as, for our purposes, this is never needed) giving us constant time operations in the average case. We implement the needed operations as follows:

- Set $may\cdot hold([[node, AA], PA])$ to $false$ for all possible node, AA, and PA. Simply initialize the hash table which takes constant time.
- Find the value of $may\cdot hold([[node, AA], PA])$ for a given node, AA, and PA. Look up $(node, AA, PA)$ in the hash table, if it is there, then return the value associated with it; otherwise return $false$. 

---

6 This “optimization” can be removed from the algorithm without great difficulty.
7 This is constant $NULL$.
8 $n$ is the number of nodes in the ICFG and $v$ is the number of variables in the program.
• Set the value of $may\text{-}hold([\text{node, AA}, \mathcal{P}, \mathcal{A}])$ for a given $\text{node, AA, and } \mathcal{P}$.
If $(\text{node, AA, }, \mathcal{P})$ is in the hash table, then simply change the value associated with it; otherwise add it.

Note that $may\text{-}hold([\text{node, } \emptyset, \langle a, b \rangle])$ implies $may\text{-}hold([\text{node, assumed alias}, \langle a, b \rangle])$ for every assumed alias. Thus if we want to know the value of $may\text{-}hold([\text{node, assumed alias}, \langle a, b \rangle])$ (and assumed alias $\not= \emptyset$) we first look up $(\text{node, } \emptyset, \langle a, b \rangle)$. If its associated value is there return true, otherwise we return the value associated with $(\text{node, assumed alias, } \langle a, b \rangle)$, unless that’s not in the table in which case we return false.

3.3 Alias Consequences

Consider an alias between $*p$ and $*q$ where $p$ and $q$ are declared by

```c
struct list {int data; struct list *next} *p,*q;
```

This alias yields the following situation:

```
  p
  |
  v
  q
```

Thus the alias between $*p$ and $*q$ implies the existence of other aliases; for example, between $p->next$ and $q->next$ and between $p->next->next$ and $q->next->next$. The existence of this last alias is based on the implicit assumption that $p->next$ and $q->next$ are not NULL. The consequences of an alias $(*p,*q)$ is defined as the set of all other aliases which are implied by $(*p,*q)$. Alias consequences can be computed by a simple recursive procedure (see Figure 4). This procedure will terminate on aliases with recursive types because of k-limiting.

In the following sections, we present a detailed description of the main ideas in our interprocedural algorithm for computing $may\text{-}hold$, including pseudo-code descriptions. First, we present mechanisms for modeling the affects of parameter bindings on aliases. Second, we present an algorithm for computing the $may\text{-}hold$ relation.
SET alias_consequences((a, b))  /* a and b must be the same type */
{
    answer = {(a, b)}
    if (can_deref(alias_type((a, b))))
        answer = answer \ alias_consequences((deref(a), deref(b)))
    if (is_struct(alias_type((a, b))))
        for each field, field, of alias_type((a, b))
            answer = answer \ alias_consequences((field_access(a, field), field_access(b, field)))
}

Figure 4: Computing the Consequences of an Alias

3.4 Modeling Parameter Bindings

In order to do interprocedural analysis, we will have to be able to model the affects of parameter bindings on aliases. We will do this with a function bind(call, assumed-alias). Intuitively, bind(call, \emptyset) will be all the aliases on entry of a called procedure that must exist because of parameter bindings, while bind(call, (a, b)) will be the set of aliases at entry of a called procedure whose existence is implied by a being aliased to b at call. Consider the following example (where q and r are global to P and q, r, and f are all type “int *”):

\[
\begin{array}{c}
\text{call}_P(q) \\
\text{store}_1: q \quad r \\
\text{entry}_P(f) \\
\text{store}_2: q \quad r \quad f
\end{array}
\]

bind(call\_\_P(q), \emptyset) = \{(\star q, \star f)\}
bind(call\_\_P(q), (\star q, \star r)) = \{(\star q, \star r), (\star f, \star r)\}

Unfortunately, this definition is not sufficient because sometimes we have to deal with object names that are not visible (Section 2.5). A procedure call can both create and destroy an alias in the calling procedure, involving an object name not visible in the called procedure. For example, the call P() in Figure 5 creates the alias (\star b, x) and destroys the alias (\star a, x) at return\_\_P() where x is not visible in P. However, only references to the visible object name in an alias pair can affect whether the alias holds on a path (i.e., there can be no direct references to an object name which
int *a,*b;
int y;

P()
{
    b=a
    a=&y;
}

main () {
    int x;
    a=&x;
P();
}

Note: The call to P in main creates the alias pair (*b, x) and destroys the alias pair (*a, x).

Figure 5: Calls affecting alias pairs involving non-visible objects

is not visible). Fortunately, a procedure has the same effect on all alias pairs which contain visible object name w and any non-visible object name. To be able to correctly account for non-visible objects, we will need one object for each type to represent all non-visible objects of that type; we call it non_visible(type). non_visible(type) is considered an ordinary object, that is, it can be dereferenced or field accessed if the typing rules permit it. Although in an implementation explicit type information is necessary, in this paper we just use non_visible and leave the the type implicit.

This leads to a change in the bind function. For reasons that will become apparent later, if any of the aliases in the bind set involve non_visible we also want to know which object in the calling procedure corresponds to non_visible. Consider the following example (where q is global to P but r is not and q,r, and f are all type “int *”):

9In [15] we referred to this by “.”.
The occurrence of \((\langle *q, \text{non-visible} \rangle, *r \rangle)\) in \(\text{bind}(\text{call}_{P(q)}, \langle *q, *r \rangle)\) represents the fact that \(*q\) is aliased to some non-visible object at the entry of the called procedure \(P\), and that, in this case, the non-visible object is \(*r\).

### 3.4.1 Computing \(\text{bind}(\text{call}, \emptyset)\)

Thus, there are two ways to aliases can be implied by parameter bindings. The first alias corresponds to a simple formal to actual pairing. For example, if \(P\) is a function with formal "\(f\)" of type "\(\text{int } \ast\)" and \(\text{call}\) is an invocation of \(P\) with actual "\(a\)" then \(\langle *f, *a \rangle\)\(^{10}\) is in \(\text{bind}(\text{call}, \emptyset)\). The second occurs if two distinct formals are passed two actuals where one actual is a prefix of the other. For example, if \(P\) is a function with two formals "\(f_1\)" (type "\(\text{int } \ast \ast\)"") and "\(f_2\)" (type "\(\text{int } \ast\)") and \(\text{call}\) is \(P(a, *a)\) then \(\langle **f_1, *f_2 \rangle\) is in \(\text{bind}(\text{call}, \emptyset)\). The algorithm for computing \(\text{bind}(\text{call}, \emptyset)\) is straightforward and is in Figure 6.

### 3.4.2 Computing \(\text{bind}(\text{call}, \langle x, y \rangle)\)

There are three ways that an alias at a call site may imply an alias on entry to a procedure. The first is trivial, if the two object names are global to the called procedure then they are also aliased on entry to the called procedure. The other two can be illustrated by the following example (both \(a_1\) and \(a_2\) are global to \(P\)):

\(^{10}\langle *f, \text{non-visible} \rangle \) if \("a\) is not visible in \(P\).
implied_by_binding(call, f, a)
{ if a is visible in the called procedure
  return alias_consequences(\langle\text{deref}(f), \text{deref}(a)\rangle)
  else return \{\langle\text{alias}, a\rangle | \text{alias} \in \text{alias_consequences}(\langle\text{deref}(f), \text{deref(\text{non_visible})}\rangle)\}
}

bind(call, \emptyset)
{ bind = \emptyset
  /* Alias effects of each parameter binding */
  for each formal (f) actual (a) pair:
  { if (can_deref(object.type(f)))
    bind = bind \cup \text{implied_by_binding}(f, a)
  } if (is_struct(object.type(f)))
  { for each field (field) of object.type(f) which can be dereferenced:
    bind = bind \cup \text{implied_by_binding}(\text{field_access}(f, field), \text{field_access}(a, field))
  }
  /* Alias effects of pairs of parameter bindings */
  for each formal (f_i) actual (a_i) pair:
  { for each formal (f_j) actual (a_j) pair (j \neq i):
    if is_prefix(a_i, a_j)
    { let\footnote{`let a = b'; puts a copy of \( b \) in \( a \).} form_i = f_i
      apply_trans(a_i, a_j, form_i)
      if (can_deref(object.type(f_j)))
        bind = bind \cup \text{alias_consequences}(\langle\text{deref}(f_j), \text{deref}(form_i)\rangle)
      if (is_struct(object.type(f_j)))
      { for each field (field) of object.type(f_j) which can be dereferenced:
        bind = bind \cup \text{alias_consequences}(\langle\text{deref(field_access}(f_j, field)), \text{deref(field_access}(form_i, field)\rangle)
      }
    }
  }
  return bind
}

Figure 6: Computing bind(call, \emptyset)
In this example, since \( *a_2 \) is aliased to \( **a_1 \) at call \( P(a_1, a_2) \), \( *f_2 \) is aliased to \( **a_1 \). This example can be generalized to the second way an alias at a call site can imply an alias at the entry of a procedure. Whenever an actual has an alias to an object name, its corresponding formal picks up an alias to that object name or a non-visible, if the object name is not visible in the called procedure. Also in the example, since \( *a_2 \) is aliased to \( **a_1 \) at call \( P(a_1, a_2) \), \( *f_2 \) is aliased to \( **f_1 \) at entry \( P(f_1, f_2) \). This is typical of the third case; when two actuals are aliased (not necessarily directly) at a call site, the corresponding formals are aliased on entry to the called procedure. The algorithm for computing \( \text{bind}(\text{call}, \langle x, y \rangle) \) is a straightforward encoding of these three cases. It can be found in Figure 8.

4 Computing may-hold

The algorithm for computing may-hold is, at a high level, very simple. First, we find all the may-hold relations which are trivially true, for example, \( \text{may-hold}([\langle \text{p} = q, \theta \rangle, \langle \text{p}, *\text{q} \rangle]) \) is true.\(^{11}\) Once we have this initial set, we compute the set of all true may-holds using a worklist algorithm. The algorithm at this level of abstraction is presented in Figure 9.

The remainder of this section gives detailed descriptions of this algorithm. In Section 4.2 to Section 4.4 we address the interprocedural aspects of may-hold which primarily entail insuring that only aliases on realizable paths are considered. In Section 4.1 and Section 4.5 we cover the intraprocedural aspects of may-hold.

4.1 Aliases introduced by assignment (node)

This routine is very simple. Let \( \text{node} \) be the pointer assignment \( \text{"p} = \text{q}" \). Clearly,

\[
\text{may-hold}([\langle \text{node}, \theta \rangle, \langle \text{deref(p)}, \text{deref(q)} \rangle]) \]

and all \( \text{may-hold}([\langle \text{node}, \theta \rangle, \langle \text{u}, \text{v} \rangle]) \) such that \( \langle \text{u}, \text{v} \rangle \) in alias_con-
same_object\(\text{call}, \text{name}_1, \text{name}_2\)

/* Implications of \text{name}_1 and \text{name}_2 being the same object */

\begin{align*}
\text{if (name}_1 \text{ and name}_2 \text{ is visible in the called procedure)}
& \text{ return alias\_consequences}(\langle \text{name}_1, \text{name}_2 \rangle) \\
& \text{ if (name}_1 \text{ or name}_2 \text{ (assume name}_1 \text{) are not visible)} \\
& \text{ in the called procedure but the other is} \\
& \text{ return \{\langle alias, name}_1 \rangle \mid alias \in \text{alias\_consequences}(\langle \text{name}_2, \text{non\_visible} \rangle)\}}
\end{align*}

same_value\(\text{call}, \text{name}_1, \text{name}_2\)

/* Implications of \text{name}_1 and \text{name}_2 referring to the same object */

\begin{align*}
\text{if (can\_deref(alias\_type(\langle \text{name}_1, \text{name}_2 \rangle)))} \\
& \text{ return same\_object(\text{call}, \text{deref}(\text{name}_1), \text{deref}(\text{name}_2))} \\
& \text{ if is\_struct(alias\_type(\langle \text{name}_1, \text{name}_2 \rangle))} \\
& \{ \text{val} = 0 \\
& \text{ for each field (field) of type alias\_type(\langle \text{name}_1, \text{name}_2 \rangle) which} \\
& \text{ can be dereferenced} \\
& \text{ val = val \cup same\_object(\langle \text{call, deref(field\_access(name}_1, field)), \text{deref(field\_access(name}_2, field) \rangle)} \\
& \text{ return val}
\end{align*}

Figure 7: Support functions for computing bind(\text{call}, \langle x, y \rangle)
/* Uses functions defined in Figure 7 */
bind(call,(x,y))
{ bind = ∅
    if both x and y are visible in the called procedure
        bind = bind ∪ {⟨x,y⟩}
    if x or y (assume x) is visible in the called
    procedure but the other is not
        bind = bind ∪ {⟨(x,non_visible),y⟩}
}

for each formal (f_i) actual (a_i) pair:
    if can_deref(object_type(f_i)) or is_struct(object_type(f_i))
        if for x or y (assume x) is_prefix(a_i,x)
            { let f_i = f_i
                if (apply_trans(a_i,x,form_i))†
                    bind = bind ∪ same_object(call,form_i,y)
                else bind = bind ∪ same_value(call,form_i,y)
            }

for each formal (f_i) actual (a_i) pair:
    for each formal (f_j) actual (a_j) pair:
        if is_prefix(a_i,x) and is_prefix(a_j,y)
            { let form_i = f_i; let form_j = f_j
                if (apply_trans(a_i,x,form_i) and apply_trans(a_j,y,form_j))‡
                    bind = bind ∪ same_object(call,form_i,form_j)
                else bind = bind ∪ same_value(call,form_i,form_j)
            }

† "let a = b" puts a copy of b in a.
‡ When at least one dereference occurs form_i and y must be aliases, otherwise they just contain the same value.
§ Both form_i and form_j must be dereferenced at least once for them to be aliases.
   If either form_i or form_j was not dereferenced, they contain the same value.

Figure 8: Computing bind(call,(x,y))
find_aliases()
{ worklist = ∅
  /* Alias Introduction */
  for each node (node) in the ICFG
    if node is an assignment to a pointer
      aliases.introduced.by.assignment(node)
    if node is a call node
      aliases.introduced.by.call(node)

  /* ImpliedAliases */
  while worklist is not empty
  { remove node, assumed_alias, possible_alias from worklist
    if node is a call node
      aliases.at.call.implies(node, assumed_alias, possible_alias)
    else if node is an exit node
      aliases.at.exit.implies(node, assumed_alias, possible_alias)
    else any_other_alias.implies(node, assumed_alias, possible_alias)
  }
}


Aliases Introduced by Assignment (node)

/* Let node be the pointer assignment ‘p = q’ */
{
    if !is_prefix(p, q)
        for each ⟨u, v⟩ in alias_consequences(⟨deref(p), deref(q)⟩)
            set may_hold([⟨node, θ⟩, ⟨u, v⟩]) to true
            add ⟨node, θ, ⟨u, v⟩⟩ to the worklist
    }
}

Figure 10: Aliases Introduced by Assignment (node)

Aliases Introduced by Call (node)

{ let entry be the entry of the called procedure
    for each ⟨u, v⟩ in bind(node, θ)
    set may_hold([⟨entry, ⟨u, v⟩⟩, ⟨u, v⟩]) to true
        add ⟨entry, ⟨u, v⟩⟩, ⟨u, v⟩⟩ to the worklist
}

Figure 11: Aliases Introduced by Call (node)

sequences(deref(p), deref(q)) are true unless p is a prefix of q. For example, the assignment “p = p->next” does not create an alias ⟨*p, *(p->next)⟩; both p and p->next refer to different objects after this assignment but their alias relationship does not change. Figure 10 has the code for this function.

4.2 Aliases Introduced by Call (node)

Given an implementation of the bind function (Section 3.4) this routine is also very simple (see Figure 11).

4.3 Alias at Call Implies (call, assumed_alias, possible_alias)

A call node effectively has two successor nodes. The first is the entry node of the procedure it invokes which is explicitly represented by an edge in the ICFG. The second is the return node which corresponds to the same call site which, although not explicitly represented in the ICFG by
an edge, must be known in order for the aliasing algorithm to function correctly (i.e., so that it only considers realizable paths in the ICFG). \( \text{may-hold}([\text{call, assumed alias}, \text{possible alias}]) \) has effects on both its corresponding entry and return nodes.

**Effects on corresponding entry node (entry)** While \( \text{holds} \) (as defined in [15]) had the nice property that the \( \text{holds} \) relations which were true at a call node did not affect the \( \text{hold} \) relations at the entry of the called procedure, this is not true for \( \text{may-hold} \) since it requires the existence of a path to the entry node with certain characteristics whereas \( \text{holds} \) simply assumed the existence of such a path. The effects are very simple, for each alias \( \langle a, b \rangle \) in \( \text{bind}(\text{call, possible alias}) \), if \( \text{may-hold}([\text{entry, } \langle a, b \rangle]) \) is false\(^{12} \), we set \( \text{may-hold}([\text{entry, } \langle a, b \rangle]) \) to true and add \( (\text{entry}, \langle a, b \rangle, \langle a, b \rangle) \) to the worklist.

**Effects on corresponding return node (return)** Before we can go into the details of how to handle the effects on \( \text{return} \), we have to introduce the functions \( \text{back-bind} \) and \( \text{back-bind}' \) which have the following definitions:

- \( \text{back-bind}_{\text{call}}(\langle a, b \rangle) \) specifies the alias on any path to \( \text{call} \) that guarantees \( a \) is aliased to \( b \) after control flows to corresponding entry node.

- \( \text{back-bind}'_{\text{call}}(\langle a, \text{non-visible} \rangle, o) \) specifies the alias on any path to \( \text{call} \) that guarantees \( a \) is aliased to the non-visible object name \( o \) after control flows to corresponding entry node.

These definitions imply that \( \text{back-bind}_{\text{call}}(\langle a, b \rangle) = \langle c, d \rangle \) iff \( \langle a, b \rangle \in \text{bind}_{\text{call}}(\langle c, d \rangle) \) and \( \text{back-bind}'_{\text{call}}(\langle a, \text{non-visible} \rangle, o) = \langle c, d \rangle \) iff \( \langle a, \text{non-visible} \rangle \in \text{bind}_{\text{call}}(\langle c, d \rangle) \) where the \text{non-visible} is the non-visible object name \( o \).

Figure 12 (which is Figure 5 in [15]) contains the equations for \( \text{holds} \) at a return node (for an explanation of why those rules are valid see Section 3.2 of [15]). \( \text{may-hold} \) is a simple encoding of these rules with a few minor modifications. The \( \text{holds} \) rules are still valid for \( \text{may-hold} \) because \( \text{may-hold} \) is basically \( \text{holds} \) with the added restriction that the assumed alias must hold on some path the the entry node of the procedure of interest. Since, in the equation for \( \text{holds} \), each \( \text{holds} \) at \text{exit} with assumption not \( \emptyset \) is paired as follows:

\[
\text{holds}([\text{exit, A.A}],[a, b]) \land \text{holds}([\text{call, assumed alias}], \text{back-bind}_{\text{call}}(\text{A.A}])
\]

\(^{12}\text{In other words, (entry, } \langle a, b \rangle, \langle a, b \rangle \text{) has not yet been added to the worklist.} \)
1. **Rule 1** If $x$ and $y$ are both not visible in the called procedure:

\[
holds((\text{return, assumed_alias}, \langle x, y \rangle)) = holds((\text{call, assumed_alias}, \langle x, y \rangle))
\]

2. **Rule 2** If $x$ and $y$ are both visible in the called procedure:

\[
holds((\text{return, assumed_alias}, \langle x, y \rangle)) = \\
holds((\text{exit, } \emptyset, \langle x, y \rangle)) \lor \bigvee_{AA \in \text{ASSUMED}} \left( holds((\text{exit, } AA, \langle x, y \rangle)) \land holds((\text{call, assumed_alias, back-bind}_{\text{call}}(AA))) \right)
\]

3. **Rule 3** If $x$ is visible but $y$ is not (the symmetric case is similar):

\[
holds((\text{return, assumed_alias}, \langle x, y \rangle)) = \\
\bigvee_{(o, \text{non-visible}) \in \text{ASSUMED}} \left( holds((\text{exit, (o, non-visible), } \langle x, \text{non-visible} \rangle)) \land holds((\text{call, assumed_alias, back-bind}_{\text{call}}((o, \text{non-visible}), y))) \right)
\]

Figure 12: holds relation at return nodes

The only way that $holds((\text{exit, } AA, \langle a, b \rangle))$ can contribute to the value at a return node is if $holds((\text{call, assumed_alias, back-bind}_{\text{call}}(AA)))$ is true and a simple inductive argument (on path length) will suffice to show that this means there must be a path to the entry of the procedure of exit on which the alias $AA$ holds. Thus this is also a valid definition for may-hold.

We want to figure out what the implications of may-hold($(\text{call, assumed_alias, possible_alias})$) are on may-hold at return. Let $possible\_alias = \langle a, b \rangle$, we need to do a case analysis:

1. **If $a$ and $b$ are both not visible in the called procedure**

   This corresponds to Rule 1 in Figure 12, and thus the desired action is obviously:

   if may-hold($(\text{return, assumed_alias, } \langle a, b \rangle)$) is not true then set it to true and add $(\text{return, assumed_alias, } \langle a, b \rangle)$ to the worklist.

2. **$a$ and $b$ are both visible in the called procedure**

   This corresponds to Rule 2 in Figure 12 which is:
\[
\text{may-hold}([(\text{return}, \text{assumed\_alias}), (x, y)]) = \text{may-hold}([(\text{exit}, \emptyset), (x, y)]) \lor \\
\bigvee_{AA} (\text{may-hold}([(\text{exit}, AA), (x, y)]) \land \text{may-hold}([(\text{call}, \text{assumed\_alias}), \text{back\_bind\_call}(AA)]))
\]

We know \text{may-hold}([(\text{call}, \text{assumed\_alias}), (a, b)]).

The \text{\rightarrow} above denotes implication, that is \text{call} \rightarrow \text{exit} means that given \text{call} we can determine \text{exit}. Since we know \text{may-hold}([(\text{call}, \text{assumed\_alias}), \text{back\_bind\_call}(AA)]) specifies Rule 2 with only one free variable \( (x, y) \), so the obvious action for \text{may-hold}([(\text{call}, \text{assumed\_alias}), (a, b)]) would be:

For each \( AA \) in \text{bind\_call}(\langle a, b \rangle)

for every possible \( (x, y) \):

\text{if} \text{ may-hold}([(\text{exit}, AA), (x, y)]) \text{ is true}

\text{if} \text{ may-hold}([(\text{return}, \text{assumed\_alias}), (x, y)]) \text{ is false then}

\{ \text{ set } \text{ may-hold}([(\text{return}, \text{assumed\_alias}), (x, y)]) \text{ to true;}

\text{ add } (\text{return}, \text{assumed\_alias}, (x, y)) \text{ to the worklist.}
\}

This, however, is not acceptable because it requires work to be done for every possible \( (x, y) \) even though most \( (x, y) \) are not necessary. Since we are performing a conjunction in which we know one half is true, instead of doing work for all \( (x, y) \), we would prefer to only do work for \( (x, y) \) such that \text{may-hold}([(\text{exit}, AA), (x, y)])). We will do this by keeping a set \text{true\_under\_assumption}(TUA) for each (exit node, assumed alias) pair:

\text{true\_under\_assumption}(exit, AA) = \{ (x, y) \mid \text{may-hold}([(\text{exit}, AA), (x, y)]) \text{ is true} \}

Given TUA, we can easily only consider the interesting \( (x, y) \). By using the trick we used for \text{may-hold} (Section 3.2) we can initialize all TUA to \( \emptyset \) in constant time. Whenever we set \text{may-hold}([(\text{exit}', AA'), (x', y')]) to \text{true} we also add \( (x', y') \) to TUA(\text{exit}', AA'). It is easy to see that this will work iff \text{may-hold}([(\text{exit}, AA), (x, y)]) becomes \text{true} before we need it when processing \text{may-hold}([(\text{call}, \text{assumed\_alias}), (a, b)])^{13}. However, if they become true in the reverse

---

13 This may be the case when, for another call site of the same procedure, the assumption \( AA \) was produced at the entry of the called procedure at an earlier iteration of the alias algorithm.
order, this will not work. Fortunately, the other case (i.e., when we process the exit node after the corresponding call node) will be captured in \texttt{alias.at.exit.implies} (Section 4.4).

3. \(a\) or \(b\) (assume \(a\)) is not visible in the called procedure (but \(b\) is)

This corresponds to Rule 3 in Figure 12 and should be analogous to the case where \(a\) and \(b\) are both visible in the called procedure, except now we need to fill in the \texttt{non_visible} at \texttt{exit} with \(a\). Thus we would get the following action for \texttt{may\_hold}([\texttt{call.assumed.alias}, \langle a, b \rangle]):

For each \((\texttt{AA, nv})\) in \texttt{bind.call(\langle a, b \rangle)}, for every \(\langle x, y \rangle\) on \texttt{TUA(exit, AA)} (assume \(x\) contains \texttt{non_visible}):

\[
\begin{array}{l}
\text{let } x' = nv; \text{ apply\_trans(\texttt{non\_visible}, x, x')} \\
\text{if } \texttt{may\_hold}([\texttt{return.assumed.alias}, \langle x', y \rangle]) \text{ is false then} \\
\text{set } \texttt{may\_hold}([\texttt{return.assumed.alias}, \langle x', y \rangle]) \text{ to true} \\
\text{add } (\texttt{return.assumed.alias}, \langle x', y \rangle) \text{ to the worklist.} \\
\end{array}
\]

More Complex Effects on Return Nodes\footnote{For example, \(x = \ast \texttt{non\_visible}\).}: While this is sufficient if only single level pointers are allowed, it is not sufficient in the general case. In general, it is possible to have an alias between two \texttt{non_visible}'s (see Figure 13). Thus we must handle the case of creation in the called procedure of an alias between two \texttt{non_visible} object names. We will do this with a special case of \texttt{may\_hold} with two assumed aliases:

\[
\texttt{may\_hold}([\texttt{exit, } \langle o1, \texttt{non\_visible} \rangle, \langle o2, \texttt{non\_visible} \rangle, \langle nv1, nv2 \rangle])
\]

This represents the fact that if \(o1\) is aliased to non-visible object name \(l1\) and \(o2\) is aliased to non-visible object name \(l2\) on a path to the entry of the procedure then, on some path to \texttt{exit, nv1} (where the \texttt{non\_visible} portion represents \(l1\)) is aliased to \(nv2\) (where the \texttt{non\_visible} portion represents \(l2\)). In Figure 14, we show how \texttt{may\_hold} can be used to represent the stores of Figure 13. At this point the reader is not expected to be able to understand how

\footnote{We are still considering the case where \(a\) is not visible in the called procedure but \(b\) is.}
int **g1, *g2;
P()  
{  
  *g1 = g2;  
}  
main ()  
{  
  int *l1, l2;  
  g1 = &l1;  
  g2 = &l2;  
  P();  
}
### Figure 14: may-hold representation for interesting stores of Figure 13

1. This is an implementation quirk. \( \text{may-hold}([\text{n}_6, \ (<\text{g}_1, \text{non-visible}>), \ (<\text{g}_1, \text{non-visible}>)]) \) is not true because \( *\text{g}_1 \) is redefined at \( n_6 \). \( \text{may-hold}([\text{n}_6, \ (<\text{g}_1, \text{non-visible}>), \ (<\text{g}_1, \text{non-visible}>)]) \) is true by Figure 17 because \( (*\text{g}_1, \text{non-visible})\) is in alias_consequences\( (*\text{g}_1, \text{non-visible})\).
these *may-holds* are derived (they are indeed obtainable from our algorithm), but the figure is simply to show how *may-hold* can represent stores with aliases between non-visible object names. With this encoding scheme we can derive *may-hold*([[ns, t1, l1]]) in Figure 14 because the following are *true*:

- *may-hold*([[ns, (*g1, non_visible) (*g2, non_visible)]], [[ns, non_visible, non_visible]])
- *may-hold*([[ns, (*g1, l1)]) where (*g1, l1) = back-bind′ _ns_ ([[ns, non_visible], l1)]
- *may-hold*([[ns, (*g2, l2)]) where (*g2, l2) = back-bind′ _ns_ ([[ns, non_visible], l2)]

Fortunately, we only need this type of *may-hold* at exit nodes. In order to handle this, we must change TUA slightly. For *may-hold*([[exit, (*o1, non_visible) (*o2, non_visible)]], [[nv1, nv2]]) we would add (*nv1, *nv2, *o2), (*o2, non_visible) being the other condition necessary for (*nv1, *nv2), to TUA(exit, *o1, non_visible)) and (*nv1, *nv2, *o1) to TUA(exit, *o2, non_visible)). To handle *may-hold*([[call, assumed_alias], (a, b)]) correctly we need the following function:

**approximate-when-both-non_visible** (call-node, (*nv1, *nv2), non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in *nv1, substitutes non_visible-name2-is for non_visible in *nv2, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry.

This function is specified in Appendix B, but is not discussed here because its implementation is not theoretically interesting. Finally, the action for *may-hold*([[call, assumed_alias], (a, b)]) is:

For each (*AA, nv) in bind _call_ (a, b)

for every (*x, y) or (*x, y, other_condition) on TUA(exit, *AA)

if both x and y contain non_visible

for each (*AA, *nv2) such that *may-hold*([[call, AA], alias]) and

(*non_visible, other_condition), *nv2) in bind _call_ (alias)

---

However, in Figure 14, we included an alias of this type at *n6*, which is not an exit node, so that the reader could more easily follow the example.
approximate-when-both-non_visible(call,\langle x,y \rangle, n_v, n_v_2, assumed_alias, \mathcal{A}_A) \\
else (assume \( x \) contains non_visible): \\
let \( x' = n_v \); apply_trans(non_visible, x, x') \\
if may_hold([[return, assumed_alias], \langle x', y \rangle]) is false then \\
set may_hold([[return, assumed_alias], \langle x', y \rangle]) to true \\
add (return, assumed_alias, \langle x', y \rangle) to the worklist.

In order to make this action efficient, we need to be able to find all:

\((\mathcal{A}_A, n_v_2)\) such that for some alias, may_hold([[call, \mathcal{A}_A], alias]) and
\((\text{non_visible}, \text{other_condition}), n_v_2\) in bind_call(alias)

in time linear to the number of such \((\mathcal{A}_A, n_v_2)\). We will do this with a data structure back_bind_true (BBT) which is defined as follows:

\[
\text{back_bind_true}(\text{call}, \text{assumed_alias}) = \\
\begin{cases} \\
(\mathcal{A}_A, n_v) \quad (\exists \text{alias}) \text{ may_hold}([[\text{call}, \mathcal{A}_A], \text{alias}]) \text{ and } (\text{assumed_alias, n_v}) \in \text{bind_call(alias)} & \text{if } \mathcal{A}_A \text{ has an object name with non_visible in it} \\
\mathcal{A}_A \quad (\exists \text{alias}) \text{ may_hold}([[\text{call}, \mathcal{A}_A], \text{alias}]) \text{ and } \text{assumed_alias} \in \text{bind_call(alias)} & \text{otherwise}
\end{cases}
\]

In the preceding discussion we ignored the issue of scope of variables. Care must be taken to insure that aliases only appear at program points that are in the scope of the object names. This requires a simple scope check before setting values of may_hold to true. We will continue to omit this check in our code. Appendix C contains the final algorithm for computing Alias_at_call_implies(node, assumed_alias, possible_alias).

4.4 Alias_at_exit_implies(exit, assumed_alias, possible_alias)

An exit node can have any number of successors, however they are all return nodes. This function simply encodes the rules for return nodes in Figure 12 with the additional case of aliases between two non-visible object names. The encoding is analogous to that for Alias_at_call_implies. Given that we know may_hold([[exit, \mathcal{A}_A], \langle z, y \rangle]) is true, specifies the rule in Figure 12 leaving only one
\[
\text{may-hold}([\text{return, assumed_alias}, \{x, y\}]) = \text{may-hold}([\text{exit}, \emptyset, \{x, y\}]) \lor \\
\bigvee_{AA} \left( \text{may-hold}([\text{exit, AA}, \{x, y\}] \land \text{may-hold}([\text{call, assumed_alias}, \text{back-bind, exit}(AA)])) \right)
\]

We know \(\text{may-hold}([\text{exit, AA}, \{x, y\}])\).

Figure 15: Implication of a known \textit{may-hold} at an exit node

free variable, \textit{assumed_alias} (see Figure 15). Since \texttt{Alias.at.exit.implies} presents no problems that weren’t already addressed by \texttt{Alias.at.call.implies}, we omit an informal description of this routine. Pseudocode for it can be found in Appendix D.

4.5 \texttt{Any\_other\_alias\_implies}(\texttt{node, assumed\_alias, possible\_alias})

The implications of \(\text{may-hold}([\text{node, assumed\_alias}, \text{possible\_alias}])\) depends strongly on its successors. These implications must be considered separately for each successor. Since we have already examined the cases where \texttt{node} is a call or exit node, the successors of \texttt{node} must be either a call, an exit, or a statement in the program (i.e., it is not possible for any successors to be an entry or return node).

\texttt{Successor} is a call node, exit node, or a program statement which is not an assignment to a pointer: These nodes simply collect \textit{may-hold} information from their parents. An alias on a path to any of these nodes holds only if it held on the same path up to its parent. Thus the action for \(\text{may-hold}([\text{node, assumed\_alias}, \text{possible\_alias}])\) is simple in the case where its successor, \texttt{succ}, is of one of these types.

\begin{verbatim}
if \text{may-hold}([\text{succ, assumed\_alias}, \text{possible\_alias}]) is false
set \text{may-hold}([\text{succ, assumed\_alias}, \text{possible\_alias}]) to true
add (\text{succ, assumed\_alias, possible\_alias}) to the worklist
\end{verbatim}

\texttt{Successor} (\texttt{succ}) is an assignment to a pointer: This case encompasses the major intraprocedural affects of pointers on aliasing. The effects of \(\text{may-hold}([\text{node, assumed\_alias}, \text{possible\_alias}])\) depends on the relationship of the object names in \textit{possible\_alias} and the object names involved
in the pointer assignment. In the following discussion we will consider \texttt{succ} to the be statement \texttt{p = q}, where \texttt{p} and \texttt{q} are arbitrary object names of pointer type (not necessarily simple variable names). What follows is a case analysis. The effects of \texttt{possible.alias} is the application of all suitable cases. The cases are (let \texttt{possible.alias} = \langle y, z \rangle):

1. Does the assignment preserve the alias?
   This is true when \texttt{p} is a prefix of neither \texttt{y} nor \texttt{z}.

2. What are the effects of an alias of \texttt{*q}?
   This case is applicable when is\_prefix\_with\_deref(q, y).

3. What are the effects of an alias of \texttt{p}?
   This case is applicable when \texttt{y} = \texttt{p}.

Below we do a detailed examination of these three cases.

1. \texttt{possible.alias} = \langle y, z \rangle where \texttt{p} is a prefix of neither \texttt{y} nor \texttt{z}.
   In all cases, \texttt{y} and \texttt{z} point to the same object after the assignment as before since only \texttt{p} changes it's value. For example, when \texttt{y} and \texttt{z} are distinct object names from \texttt{p} and \texttt{q}.

   \[
   \begin{array}{c}
   \text{node:} \begin{array}{c}
   \text{succ:} \{ p = q; \}
   \end{array}
   \\
   \end{array}
   \]

   Clearly the assignment has no effects on \langle \texttt{y}, \texttt{z} \rangle, thus the action in this case is simply:

   \[
   \begin{array}{l}
   \textbf{if may-hold}([\texttt{succ.assume_alias}, \texttt{possible.alias}]) \textbf{ is false} \\
   \textbf{set may-hold}([\texttt{succ.assume_alias}, \texttt{possible.alias}]) \textbf{ to true} \\
   \textbf{add} (\texttt{succ.assume_alias}, \texttt{possible.alias}) \textbf{ to the worklist}
   \end{array}
   \]

   This is clearly safe, but it could also be approximate. Consider the following situation:
Thus $\langle **u, z \rangle$ is killed by "$p = q$" because $\langle u, p \rangle$ occurs at the same time. For safety we always assume that there is some path on which $\langle **u, z \rangle$ holds but $\langle u, p \rangle$ does not. This is the third source of approximation from Section 1.

2. \texttt{possible.alias} = \langle y, z \rangle \text{ where } \texttt{is.prefix.with.deref(q,y)}.

There are 3 different cases that need to be handled (the first two are mutually exclusive, but either can occur in conjunction with the third). They are:

i. \texttt{not is.prefix}(p,z)

ii. \texttt{is.prefix}(p,z)

iii. Interaction of $\langle *q, z \rangle$ with other known aliases (second type of approximation from Section 1).

In general, the effects of the assignment on this alias depend on whether or not \texttt{is.prefix}(p,z).

The two types of effects are characterized by the following examples:

In case 2.i, \texttt{may-hold}([\texttt{node, assumed.alias}, \langle *q, z \rangle]) implies

\texttt{may-hold}([\texttt{succ, assumed.alias}, \langle *q, z \rangle] \text{ and } \texttt{may-hold}([\texttt{succ, assumed.alias}, \langle *p, z \rangle]). \text{ While in}
case 2.ii, \textit{may-hold}([\textit{node, assumed alias}]. \langle *q, \textit{p=next} \rangle)) gives no information about the aliasing that occurs at \textit{succ}. Thus it would seem that the action when \textit{may-hold}([\textit{node, assumed alias}], \langle y, z \rangle) \textit{is true} for \textit{succ} = “p = q” where \textit{is prefix with deref}(q, y) would be:

\begin{verbatim}
if !is_prefix(p, z) /*is_prefix(p, z) \Rightarrow case 2.ii; !is_prefix(p, z) \Rightarrow case 2.i*/
    { let p' = p
      apply_trans(q, y, p)
      if may-hold([succ, assumed alias], \langle p', z \rangle) \textit{is false}
      { set may-hold([ succ, assumed alias], \langle p', z \rangle) to true
        add (succ, assumed alias, \langle p', z \rangle) to the worklist
      }
    }
\end{verbatim}

This, however, can miss some aliases when \textit{p} is not a prefix of \textit{z}. Consider the following (case 2.iii):

\begin{center}
\begin{tikzpicture}
    \node (node) at (0,0) {node:}
    \node (succ) at (1,-2) {succ: \langle p = q; \rangle}
    \draw[->] (node) -- (succ) node[midway, above] {\textit{u} \arrow{p} \textit{q}}
    \draw[->] (node) -- (succ) node[midway, left] {\textit{u} \arrow{p} \textit{q}}
    \draw[->] (node) -- (succ) node[midway, above] {\textit{u} \arrow{z}}
    \end{tikzpicture}
\end{center}

The problem is that the existence of the alias \langle **u, z \rangle at \textit{succ} does not necessarily follow from a single alias at \textit{node}. Instead, \langle **u, z \rangle can hold on \textit{entry main}...[\textit{node}][\textit{succ}] if both \langle *u, p \rangle and \langle z, *q \rangle must hold on the path \textit{entry main}...[\textit{node}]. Unfortunately, we do not keep any information about pairs of aliases holding on the same path. Thus whenever we have \textit{may-hold}([\textit{node, AA}], \langle *u, p \rangle) and \textit{may-hold}([\textit{node, AA}], \langle z, *q \rangle) we have to assume \textit{may-hold}([“p = q”, AA], \langle *u, z \rangle) in order for our solution to be safe. Thus we must extend the

\footnote{The only alias that holds at \textit{succ} in case 2.ii, is \langle *p, *q \rangle which holds regardless of the alias situation at \textit{node}.}
if !is_prefix(p, z) /* is_prefix(p, z) ⇒ case 2.ii; !is_prefix(p, z) ⇒ case 2.i */
{ let p' = p
  apply_trans(q, y, p')
  if may_hold([(succ, assumed_alias), (p', z)]) is false
  { set may_hold([(succ, assumed_alias), (p', z)]) to true
    add (succ, assumed_alias, (y, z)) to the worklist
    for each (AA, (p, v)) such that may_hold([(node, AA), (p, v)]) is true
    /* Case 2.iii: v ≡ u and y ≡ q in text */
      if v contains non_visible or !is_prefix(v, z)
        /* In order to apply case 2.iii, we need (y, z) and (p, v) to hold on the same
          path. However, if is_prefix(v, z) is true this is not possible. For
          example, let z ≡ v->next. On any path to node on which (p, v) holds, v
          will be redefined at succ and thus the alias (y, z(≡ v->next)) cannot hold
          at the same time. If is_prefix(v, z) but v contained non_visible, we
          cannot safely assume that the non_visible in both represent the same
          non_visible object name. */
        { let v' = v
          apply_trans(q, y, v')
          safely_make_alias(succ, v', AA, z, assumed_alias)
        }
  }
}

Figure 16: Action for may_hold([(node, assumed_alias), (y, z)]) for successor, succ ≡ “p = q”, of
node and is_prefix_with_deref(q, y)

action for may_hold([(“p = q”, assumed_alias), (y, z)]) where is_prefix_with_deref(q, y) to account
for this situation. The new action appears in Figure 16.

There are two notable features of Figure 16 which deserve explanation. First is the function
safely_make_alias which is defined as follows:

safely_make_alias(node, on1, AA1, on2, AA2) When safely_make_alias is invoked we know
that ⟨on1, on2⟩ holds on a path to node. However, we also know that two assumptions
AA1 and AA2 are necessary. We do not allow multiple assumptions, so this routine safely
approximates this situation. If both assumptions contain non_visible then this will be
the special case of an alias in which both components contain non_visible (as discussed
on p. 28). Otherwise both assumptions are individually necessary and either can be

37
safely chosen. If one assumption contains non-visible, then use that one; otherwise use either (i.e., only use the assumption 0 if both are 0).

Second, a data structure must be maintained so that

for each \((\mathcal{A}, \langle p, v \rangle)\) such that \(\text{may}-\text{hold}([([\text{node}, \mathcal{A}], \langle p, v \rangle)])\) is true

can be implemented efficiently, similarly to BBT and TUA. As with those two, this action will only work correctly if \(\text{may}-\text{hold}([([\text{node}, \mathcal{A}], \langle p, v \rangle)])\) becomes true before \(\text{may}-\text{hold}([([\text{node}, \mathcal{A}], \langle p, z \rangle)])\) where is\_prefix\_with\_deref\((q, y)\), but as with BBT and TUA, another action will handle the case where they become true in the opposite order (see 3 below).

3. possible\_alias = \(\langle p, v \rangle\)

Again there are three cases to consider:

i. Simple effects that must be done for ALL \(\langle p, v \rangle\).

ii. Secondary effects that also must be done for ALL \(\langle p, v \rangle\).

iii. Interaction of \(\langle p, v \rangle\) with other known aliases (2nd type of approximation from Section 1).

The effects of “\(p = q\)” on \(\langle p, v \rangle\) are characterized by the following example in which \(v\) is \(*u\):

\[
\text{node:} \quad \begin{array}{c}
\bullet
\end{array} \\
\downarrow
\text{succ:} \quad \begin{array}{c}
p = q;
\end{array}
\]

\(\uparrow
\begin{array}{c}
u
\end{array}
\begin{array}{c}
p
\end{array}
\begin{array}{c}
| q
\end{array}
\)
\(\begin{array}{c}

\text{Case 3.i: } \text{may}-\text{hold}([([\text{node, assumed\_alias}], \langle p, *u \rangle)]) \text{ implies}
\text{may}-\text{hold}([([\text{succ, assumed\_alias}], \langle p, *u \rangle)]) \text{ and, unless } u \text{ or } p \text{ is a prefix of } q^{18},
\text{may}-\text{hold}([([\text{succ, assumed\_alias}], \langle q, *q \rangle)])
\]

\text{18 when } u \text{ or } p \text{ is a prefix of } q, \text{ we do not want to create } \langle *u, *q \rangle \text{ for the same reason we do not want to create } \langle *p, *(p->next) \rangle \text{ for } “p = p->next”.

38
Case 3.ii: An alias \(<s, o>\) at node is, in general, implicitly killed (even though it hasn’t been explicitly mentioned, an examination of our actions will show that the alias \(<s, o>\) at node will not have any effects at succ). However, in the case where \(<p, *u>\) holds on some path to node, \(<s, **u>\) will not be killed by the assignment “\(p = q\)” and we have to account for this.

Case 3.iii: The only other effect of may-hold(\([\text{node} \cdot \text{assumed_alias}, \langle p, *u \rangle]\)) comes from handling the other half of case 2.iii. I.e., when may-hold(\([\text{node} \cdot \text{assumed_alias}, \langle *q, z \rangle]\)) becomes true first in:

\[
\text{node: } \begin{array}{c}
\vdots \\
\end{array} \\
\text{succ: } \langle p = q; \rangle
\]

which was presented in the last case. The action for this is presented in Figure 17. Analogous to other cases, a data structure must be maintained to implement

\[
\text{for each } \langle y, z \rangle, AA \text{ such that may-hold(\([\text{node} \cdot AA, \langle y, z \rangle]\)) and} \\
\text{is_prefix_with_deref}(q, y) \text{ and } \text{is_prefix}(p, z)
\]

efficiently.

5 May Alias

Computing May Alias given may-hold is extremely simple:

\[
\text{may-alias}(\text{node}) = \{ PA | \exists AA \text{may-hold}(\text{node}, AA, PA) = \text{true}\}
\]

This can clearly be computed in time linear in the size of the may-hold solution. In fact, given a clever data structure for may-hold, no additional computation (nor space) is needed to get the solution for may-alias.
for each \( \langle a, b \rangle \) in alias\_consequences\((p, v)\) /* Case ii */ or
  (provided \(!is\_prefix(v, q) \) and \(!is\_prefix(p, q)\))
in alias\_consequences\((\langle v, \ast q \rangle)\) /* Case i */
  \{ if may\_hold\(((suc\_assumed\_alias), \langle a, b \rangle)\) is false
    \{ set may\_hold\(((suc\_assumed\_alias), \langle a, b \rangle)\) to true
      add \((suc\_assumed\_alias, \langle a, b \rangle)\) to the worklist
    \}
  \}
for each \((AA, \langle y, z \rangle)\) such that may\_hold\(((node, AA), \langle y, z \rangle)\) and
  \(\) is\_prefix\_with\_deref\((q, y)\) and \(!is\_prefix(p, z)\)

/* is\_prefix\_with\_deref\((q, y)\) signals Case iii and we must account for it.
   However, if is\_prefix\((p, z)\) then we are saying we have to account for cases
equivalent to, for example, \(v = p\rightarrow next(\equiv z)\). In this case, we do not
need to do anything about the relationship between \(v\rightarrow next\) and \(p\rightarrow next\)
since \(v\) and \(p\) are both 'shifted' by \(-\rightarrow next\). */

if \(v\) contains non\_visible or \(!is\_prefix(v, z)\)  /* Rules out effects of cases
  equivalent to, for example,
  \(v = v\rightarrow next(\equiv z)\) */
  \{ let \(v' = v\)
    apply\_trans\((q, y, v')\)
    safely\_make\_alias\((suc\_assumed\_alias, z, AA)\)
  \}

Figure 17: Action for may\_hold\(((node, assumed\_alias), \langle p, v \rangle)\) for successor, suc\_ = "p = q" of node
6 Conclusion

We have presented a safe approximate algorithm for the Interprocedural May Alias Problem induced by general pointer usage. Currently, we are testing our algorithm on small C programs [13]. We believe it will be shown a practical and efficient technique.
Appendix A: Dictionary of Functions

`address_type(type)` returns the type of the objects which can point to `type`. (Section 2.2)

`alias_consequences(alias)` is the set of all aliases that are implied by `alias`. An algorithm for computing `consequences(alias)` can be found in Figure 4. (Section 3.3)

`alias_type(alias)` is the type of the object which is referred to by both object names in `alias`. This function correctly accounts k-limiting. (Section 2.4)

`amp_object_name(object_name)` is `&(object_name)`, if `object_name` does not start with an `&`, otherwise `error`. (Section 2.3)

`apply_trans(object_name1,object_name2,object_name3)`: `object_name1` and `object_name3` must have the same type and `is_prefix(object_name1, object_name2)` must be `true`. The function applies to `object_name3` the sequence of dereferences and field accesses necessary to transform `object_name1` into `object_name3`. It returns `true` if any dereferences occurs somewhere in the sequence. Some examples of `apply_trans` can be found in Figure 3. (Section 2.3)

`approximate_when_both_non_visible` (call-node, ⟨nv1, nv2⟩, non_visible-name1-is, non_visible-name2-is, name1-AA, name2-AA) substitutes non_visible-name1-is for non_visible in `nv1`, substitutes non_visible-name2-is for non_visible in `nv2`, and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry. (Appendix B)

`back_bind_true(call, assumed_alias)` =

\[
\begin{cases}
(AA, nv) & (\exists alias) \text{ may hold } ([call, AA, alias]) \text{ and } \\
(AA, nv) & (assumed_alias, nv) \in \text{ bind}_\text{call}(alias) \\
(AA) & (\exists alias) \text{ may hold } ([call, AA, alias]) \\
& \text{ and } assumed_alias \in \text{ bind}_\text{call}(alias)
\end{cases}
\]

if `AA` has an object name with non_visible in it

otherwise

`BBT` : see `back_bind_true`.

`can_deref(type)` is `true` iff `type` can be legally dereferenced without casting. (Section 2.2)

`deref(object_name)` removes the `&` if `object_name` starts with an `&` and otherwise returns `*(object_name)`. This corresponds to rule 1 of Figure 2. (Section 2.3)

`deref_type(type)` If `type` can be dereferenced, `deref_type` returns the type of the objects to which `type` may point. (Section 2.2)
field_access(object_name, field_name) is (object_name).field_name, if object_name does not start with an &, otherwise error. This corresponds to rule 2 of Figure 2. (Section 2.3)

field_type(type, field) If type is a structure with field, field_type returns the type of field. (Section 2.2)

is_field_off(field, type) is true iff type is a structure type and field is a legal field of type. (Section 2.2)

is_k_limited(object_name) is true iff object_type(object_name) can be dereferenced, object_name contains k dereferences, and object_name does not start with an address operator. (Section 2.3)

is_prefix(object_name1, object_name2) returns true iff object_name1 can be transformed into object_name2 by a (possibly empty) sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2). (Section 2.3)

is_prefix_with_deref(object_name1, object_name2) returns true iff object_name1 can be transformed into object_name2 by a sequence of dereferences and field accesses (i.e. applications of rule 1 and rule 2 of Figure 2) with AT LEAST ONE dereference. (Section 2.3)

is_struct(type) is true iff type is a structure. (Section 2.2)

may_hold([\{n_i, AA\}, \{a, b\}]) is true iff \{a, b\} holds on some path from entry(n_i), the entry of the procedure containing n_i, to n_i assuming there is a path from entry of main to entry(n_i) on which the assumed alias AA holds and there is a path from entry of main to the entry(n_i) on which AA holds. (Section 3.1)

object_type(object_name) is the data type which can be stored in objects with object_name, if object_name does not start with an address operator. If for some name, object_name = &name then it is address_type(name). (Section 2.3)

safely_make_alias(node, on_1, AA_1, on_2, AA_2) When safely_make_alias is invoked we know that \{on_1, on_2\} holds on a path to node. However, we also know that two assumptions AA_1 and AA_2 are necessary. We do not allow multiple assumptions, so this routine safely approximates this situation. If both assumptions contain non_visible then this will be the special case of an alias in which both components contain non_visible (as discussed on p. 28). Otherwise both assumptions are individually necessary and either can be safely chosen. If one assumption contains non_visible, then use that one; otherwise use either (i.e., only use the assumption \emptyset if both are \emptyset).

simple_object_name(variable) is the object name which is simply variable, i.e. rule 3 of Figure 2. (Section 2.3)

\footnote{This is the same as the value of the type attribute for object_name obtained from Figure 2.}
true_under_assumption((exit, AA)) = \{ (x, y) \mid \text{may-hold}([\text{exit, AA}, \langle x, y \rangle]) \text{ is true} \} (\text{Section 4.3})

TUA : see true_under_assumption.
Appendix B: Approximate-when-both-non-visible

approximate-when-both-non-visible (call-node, \(\langle nv_1, nv_2\rangle, \text{non-visible-name1-is, non-visible-name2-is, name1-AA, name2-AA}\)) substitutes non-visible-name1-is for non_visible in \(nv_1\), substitutes non_visible-name2-is for non_visible in \(nv_2\), and uses the assumed aliases name1-AA and name2-AA to establish a safe assumed alias condition at procedure entry.

approximate-when-both-non-visible(call-node,\(\langle nv_1, nv_2\rangle,\text{non-visible-name1-is, non-visible-name2-is, name1-AA, name2-AA}\))
{
    apply.trans(\text{non-visible},nv_1,\text{non-visible-name1-is})
    apply.trans(\text{non-visible},nv_2,\text{non-visible-name2-is})
    if both name1-AA and name2-AA contain a non_visible
    {
        /* Creates another alias in which both object names are non_visible */
        let exit be the exit node of the procedure containing call-node
        if may_hold(\langle exit, name1-AA, non-visible-name1-is, name2-AA, non-visible-name2-is\rangle) is false
        {
            set may_hold(\langle exit, name1-AA, non-visible-name1-is, name2-AA, non-visible-name2-is\rangle) to true
            add (exit, name1-AA, non-visible-name1-is, name2-AA, non-visible-name2-is)
            to worklist
        }
    }
} else

45
{  
/* Both assumptions are individually necessary and either can be safely  
chosen. If one assumption contains non_visible, then use that one  
(because we need it to figure out which non-visible object name non_visible  
represents; otherwise use either (but only use the assumption ∅ if both are  
∅ because requiring both ∅ and A_A is the same as requiring just A_A). */  
if name1-AA is ∅ or name2-AA contains non_visible  
   AA = name2-AA
else AA = name1-AA
for each ⟨x,y⟩ in  
alias_consequences(⟨non_visible-name1-is,non_visible-name2-is⟩)  
   let return be the return node that corresponds to call-node
   if may-hold([[return,AA],⟨x,y⟩]) is false  
   {  
      set may-hold([[return,AA],⟨x,y⟩]) to true  
      add (return,AA,⟨x,y⟩) to worklist  
   }  
}  


1 Both name1-AA and name2-AA are necessary aliasing conditions at the entry of the procedure  
for z aliased to y at return. We only allow one alias condition, and it is safe but approximate to  
arbitrarily pick either one.
Appendix C: Alias_at_call_implies

alias_at_call_implies(call, assumed_alias, ⟨a, b⟩)
{
    let return be the return node that corresponds to call
    let exit be the exit node of the called procedure
    let entry be the entry node of the called procedure
    if both a and b are not visible in the called procedure /* Rule 1 */
        if may-hold([[return, assumed_alias], ⟨a, b⟩]) is false
            set may-hold([[return, assumed_alias], ⟨a, b⟩]) to true
            add (return, assumed_alias, ⟨a, b⟩) to worklist
    } /* Below an expanded version of the code found on page 31 with the effects on entry added. */
    for each AA and (AA, nv) in bind_call(⟨a, b⟩)
        if may-hold([[entry, AA], AA]) is false /* effects on corresponding entry node */
            set may-hold([[entry, AA], AA]) to true
            add (entry, AA, AA) to worklist
    }
    for every ⟨x, y⟩ or (⟨x, y⟩, other_condition) in TUA(exit, AA)
        if neither x nor y contain non_visible /* Rule 2 */
            if may-hold([[return, assumed_alias], ⟨x, y⟩]) is false
                set may-hold([[return, assumed_alias], ⟨x, y⟩]) to true
                add (return, assumed_alias, ⟨x, y⟩) to the worklist.
        }
    } else
        if both x and y contain non_visible /* Rule 3 - special case */
            for each (AA, nv2) in BBT(call, ⟨non_visible, other_condition⟩)
                approximate-when-both-non_visible(call, ⟨x, y⟩, nv, nv2, assumed_alias, AA)
        else (assume x contains non_visible): /* Rule 3 */
            let x' = nv; apply_trans(non_visible, x, x')
            if may-hold([[return, assumed_alias], ⟨x', y⟩]) is false
                set may-hold([[return, assumed_alias], ⟨x', y⟩]) to true
                add (return, assumed_alias, ⟨x', y⟩) to the worklist.
    }
} /* end for each element of TUA */
} /* end for each element of bind */
Appendix D: Alias_at_exit_implies

alias_at_exit_implies(exit, AA, ⟨x, y⟩)
{ if both x and y contain non_visible
  { let AA = ⟨a₁, a₂⟩
    for each call node (call) that invokes the procedure containing exit
      for each ⟨assumed₁, nv₁⟩ in BBT(call, ⟨non_visible, a₁⟩)
        for each ⟨assumed₂, nv₂⟩ in BBT(call, ⟨non_visible, a₂⟩)
          approximate_when_both_non_visible(call, ⟨x, y⟩, nv₁, nv₂, assumed₁, assumed₂)
  } else
  { if neither x nor y contains non_visible (assume x)
    for each call node (call) that invokes the procedure containing exit
      for each assumed in BBT(call, AA)
        let return be the return node that corresponds to call
        if the object names x and y at call are visible in the called procedure
          /* e.g., two aliased locals in a recursive procedure */
            if may_hold([[return, assumed], ⟨x, y⟩]) is false
              { set may_hold([[return, assumed], ⟨x, y⟩]) to true
                add (return, assumed, ⟨x, y⟩) to worklist
              }
    if x or y contains non_visible (assume x)
    for each call node (call) that invokes the procedure containing exit
      for each ⟨assumed, nv⟩ in BBT(call, AA)
        let return to the return node that corresponds to call
        if the object name y at call is visible in the called procedure
          { apply_trans(non_visible, x, nv)
            for each ṖA in alias_consequences(⟨nv, y⟩)
              if may_hold([[return, assumed], ṖA]) is false
                { set may_hold([[return, assumed], ṖA]) to true
                  add (return, assumed, ṖA) to worklist
                }
          }
    }
  }
}
Appendix E: Any\_other\_alias\_implies

```c
any\_other\_alias\_implies(node, assumed\_alias, \langle y, z \rangle)
{ for each successor, succ, of node
  { if succ is not an assignment to a pointer
    { if may\_hold([\langle succ, assumed\_alias \rangle, \langle y, z \rangle]) is false
      { set may\_hold([\langle succ, assumed\_alias \rangle, \langle y, z \rangle]) to true
        add (succ, assumed\_alias, \langle y, z \rangle) to the worklist
      }
    } else /* let node be the assignment ‘‘p = q’’ */
    { if p is not a prefix of y and p is not a prefix of z /* Case 1 */
      if may\_hold([\langle succ, assumed\_alias \rangle, \langle y, z \rangle]) is false
      { set may\_hold([\langle succ, assumed\_alias \rangle, \langle y, z \rangle]) to true
        add (succ, assumed\_alias, \langle y, z \rangle) to the worklist
      }
    }
    if is\_prefix\_with\_deref(q, y) /* Case 2 */
    { if !is\_prefix(p, z)
      { let \( p' = p \)
        apply\_trans(q, y, p')
        if may\_hold([\langle succ, assumed\_alias \rangle, \langle p', z \rangle]) is false
        { set may\_hold([\langle succ, assumed\_alias \rangle, \langle p', z \rangle]) to true
          add (succ, assumed\_alias, \langle p', z \rangle) to the worklist
          for each \( AA, \langle p, v \rangle \) such that
            may\_hold([\langle node, AA \rangle, \langle p, v \rangle]) is true
            if \( v \) contains non\_visible or !is\_prefix(v, z)
            { apply\_trans(q, y, v)
              safely\_make\_alias(succ, v, AA, z, assumed\_alias)
            }
          }
        }
      }
    }
  }
}
```
if $z = p$ /* Case 3 */
    { for each $(a, b)$ in alias_consequences($(p, y))$ or
        (provided !is_prefix(y, q) and !is_prefix(p, q))
            alias_consequences($(p, y))
            { if $may\_hold([\langle suc, assumed\_alias \rangle, (a, b)])$ is false
                { set $may\_hold([\langle suc, assumed\_alias \rangle, (a, b)])$ to true
                    add $(\langle suc, assumed\_alias, (a, b) \rangle)$ to the worklist
                }
            }
            for each $(\langle A, A, (a, b) \rangle)$ such that $may\_hold([\langle node, A, A \rangle, (a, b)])$ and
                is_prefix_with_deref(q, a) and !is_prefix(p, b)
                if y contains non_visible or !is_prefix(y, b)
                    { let $y' = y$
                        apply_trans(q, a, y')
                        safely_make_alias(succ, y', assumed_alias, b, A)
                    }
            }
    } /* node is an assignment to a pointer */
} /* end for each successor */
References


