Undecidability of Static Analysis

William Landi
Siemens Corporate Research Inc
755 College Rd East
Princeton, NJ 08540
wlandi@scr.siemens.com

Abstract

Static Analysis of programs is indispensable to any software tool, environment, or system that requires compile time information about the semantics of programs. With the emergence of languages like C and LISP, Static Analysis of programs with dynamic storage and recursive data structures has become a field of active research. Such analysis is difficult, and the Static Analysis community has recognized the need for simplifying assumptions and approximate solutions. However, even under the common simplifying assumptions, such analyses are harder than previously recognized. Two fundamental Static Analysis problems are May Alias and Must Alias. The former is not recursive (i.e., is undecidable) and the latter is not recursively enumerable (i.e., is uncomputable), even when all paths are executable in the program being analyzed for languages with if-statements, loops, dynamic storage, and recursive data structures.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Processors; F.1.1 [**Computation by Abstract Devices**]: Models of Computation– *bounded-action devices*; F.4.1 [**Math Logic and Formal Languages**]: Mathematical Logic– *computability theory*

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, data flow analysis, abstract interpretation, halting problem

From *acm Letters on Programming Languages and Systems*,
Vol. 1, No. 4, December 1992, Pages 323-337.

1 Introduction

Static Analysis is the processes of extracting semantic information about a program at compile time. One classical example is the *live variables* [4] problem; a variable x is *live* at a statement s iff on some execution x is used (accessed) after s is executed without being redefined. Other classical problems include reaching definitions, available expressions, and very busy expressions [4]. There are two main frameworks for doing Static Analysis: *Data Flow Analysis* [4] and *Abstract Interpretation* [3]. The framework is not relevant to this paper, as we show that two fundamental Static Analysis problems are harder than previously acknowledged, regardless of the framework used.

We view the solution to a Static Analysis problem as the set of "facts" that hold for a given program. Thus, for live variables the solution is $\{(x, s) |$ variable x is live at statement $s\}$. With that in mind, we review a few definitions:

- A set is *recursive* iff it can be accepted by a Turing machine that halts on all inputs.
- A set is *recursively enumerable* iff it can be accepted by a Turing machine which may or may not halt on all inputs.

Static Analysis originally concentrated on FORTRAN, and was predominately confined to a single procedure (*intraprocedural analysis*) [7, 9, 15]. However, even this simple form of Static Analysis is not recursive. The difficulty lies in conditionals. There are, in general, many paths through a procedure, but not all paths correspond to an execution. For example, consider

```
if (x > -1) y = 1;
if (x < 0) y = -1;
```

Execution of this fragment always executes exactly one *then* branch. It is impossible for both or neither *then* branches to be executed. Static Analysis is not recursive since determining which paths are executable is not recursive. To overcome this problem, Static Analysis is performed assuming that all paths through the program are executable [2]. This assumption is not always valid, but it is *safe* [2].[1] Also, it simplifies the problem and allows Static Analysis of FORTRAN procedures to be done fairly efficiently. Some approaches (for example [16]) categorize some paths as not executable. However, these techniques have limited applicability, and often must assume that paths are executable.

With a basis of a firm understanding of intraprocedural Static Analysis of FORTRAN, Static Analysis of entire programs (*interprocedural analysis*) was investigated. Myers [14] came up with the negative result that many interprocedural Static Analysis problems are $\mathcal{NP}$ complete. Practically, this means that interprocedural Static Analysis must make further approximations over intraprocedural analysis or take an exponential amount of time.

With the emergence of popular languages like C and LISP, the Static Analysis community has turned its attention to languages with pointers, dynamic storage, and recursive data structures. It is widely accepted that Static Analysis under these conditions is hard. The general feeling is that it is probably $\mathcal{NP}$ complete [11, 13, 12]; this is incorrect. Recently, the problem of finding aliases was shown to be $\mathcal{P}$-space hard [10]. Unfortunately, this is still an underestimate.

[1] The term *conservative* is used in [2] instead of *safe*.

An *alias* occurs at some point during execution of a program when two or more names exist for the same storage location. For example, the C statement "`p = &v`" creates an alias between `*p` and `v`. Aliases are associated with program points, indicating not only that `*p` and `v` refer to the same location during execution, but also where in the program they refer to the same location. *Aliasing*, statically finding aliases, is a fundamental problem of Static Analysis. Consider the problem of finding live variables for:

```
s1:  v = 1;
s2:  p = &v;
s3:  w = 2;
s4:  printf("%d",*p);
```

The variable `v` is live at s_3 only because `*p` is aliased to `v` when program point s_4 is executed. Aliasing also influences most interesting Static Analysis problems. Any problem that is influenced by aliasing is at least as hard as aliasing. There are two types of aliasing.

May Alias Find the aliases that occur during *some* execution of the program.
Must Alias Find the aliases that occur on *all* executions of the program.

Finding the aliases can mean determining the set of all aliases which hold at some associated program points, or determining whether x and y are names for the same location at a particular program point s. We use the latter meaning as, in general, the set of all aliases maybe infinite in size. We formally define May Alias as a boolean function:

> $may\text{-}alias_P(s, \langle x, y \rangle)$ is *true* iff there is an execution of program P to program point s (including the effects of executing s) on which x and y refer to the same location.

Must Alias is defined analogously. We show that, for languages with if-statements, loops, dynamic storage, and recursive data structures, Intraprocedural May Alias is not recursive (i.e., is undecidable) and Intraprocedural Must Alias is not recursively enumerable (i.e., is uncomputable) even when all paths in a program are executable by *reducing* ([5], p. 321-322) the halting problem into an alias problem. This is a different from the result of Kam and Ullman [8] that the MOP solution is undecidable for monotone frameworks.

2 Reduction of the Halting Problem to an Alias Problem

A *Deterministic Turing Machine* (DTM) [1] is a tuple (Q,T,I,δ,β,q_0,q_f) where:

- Q = $\{q_1, q_2, ..., q_{n_Q}\}$ is the set of states
- T = $\{\sigma_1, \sigma_2, ..., \sigma_{n_T}\}$ is the set of tape symbols
- I $\subset$ T is the set of input symbols
- δ: (Q $\times$ T) $\rightarrow$ (Q $\times$ T $\times$ {L,R,S}) is the transition function[2]

- $\beta \in T - I$ is the blank symbol
- $q_0 \in Q$ is the start state
- $q_f \in Q$ is the final state

We assume that δ is a total function and that the DTM will not move off the left end of the tape. In general, neither of these assumptions are true, but any Turing machine can be modified to conform to them.

In this section, we specify a machine *reduce* (Figure 1) which takes a DTM M and input string w and procedures a program C such that

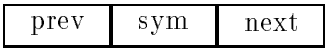
- *may-alias*$_C(s, \langle$**current_state,valid_simulation$\rangle)$ is *true* iff M halts on w.
- *must-alias*$_C(s, \langle$**current_state,not_valid$\rangle)$ is *true* iff M does not halt on w.
- all paths through C are executable

2.1 Representing an ID

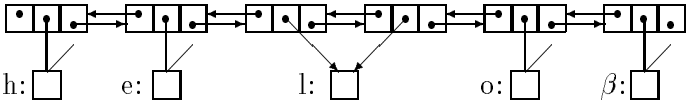
An *Instantaneous Description* (ID) is an encoding of the following information:

- contents of the DTM's tape
- current state of the DTM
- location of the tape head

An ID is usually represented by a string $x\underline{q_i}y \in T^*QT^*$ where the tape contains xy infinitely padded to the right with blanks, the current state is q_i, and the tape head is scanning the first character of y.[3] We encode this information in the *alias pattern* of a program execution. By alias pattern, we mean the relationship of names to each other.

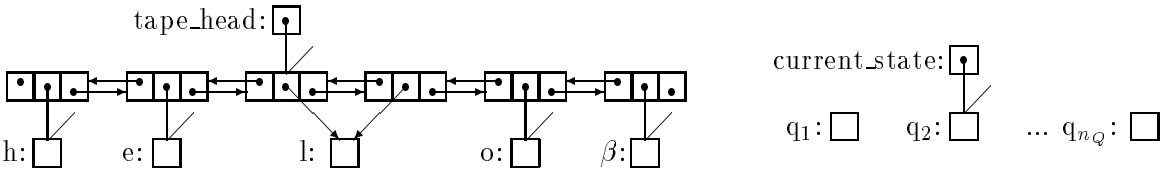
We use a doubly linked list to represent the tape of a DTM: 

. For each $\sigma_i \in T$ we create a variable σ_i. The "sym" field points to σ_i iff the tape location contains σ_i. Thus, the tape that contained "hello" padded to the right with blanks (β) is represented by the alias pattern:



[2] L moves tape head left, R moves tape head right, and S leaves the tape head where it is.
[3] q_i is underlined in $x\underline{q_i}y$ to make the state stand out from the tape string.

For each $q_i \in Q$ we create a variable $\mathbf{q}_i$ and there are two additional variables, **current_state** and **tape_head**. **Current_state** points to the current state of the machine, and **tape_head** indicates the tape head location by pointing into the list representing the tape. The ID = $he\underline{q_2}llo$ is represented by:



2.2 Programming Language

In order to perform the required reduction, we need to construct a program from a DTM. The program is in C, but it could be any language with if-statements, loops, dynamic storage, and recursive data structures. We use the address operator (&) but it is not fundamentally necessary to the proof. To specify a C program from a DTM, we need the meta-statements: `#for` and `#if`. The syntax and meaning of these are relatively straight forward and should be apparent from the following examples:

```
#for i = 1 to 3
   x_i = i;
#endfor
```
represents
```
x_1 = 1;
x_2 = 2;
x_3 = 3;
```

```
#for i = 1 to 3
   x_i = i;
   #if i is odd
      y_i = i;
   #endif
#endfor
```
represents
```
x_1 = 1;
y_1 = 1;
x_2 = 2;
x_3 = 3;
y_3 = 3;
```

Also, we use *next_bool* for reading program input. It returns the next boolean value from the input stream. If the end of the stream has been encountered, it returns 0.

2.3 Simulating a DTM

In Section 2.1, we showed how we represent an ID with aliases. In this section, we show how to simulate a DTM with the alias pattern of executions of a particular program. We now specify *reduce* (Figure 1) which constructs a program from a DTM $M = (Q,T,I,\delta,\beta,q_0,q_f)$ with initial input $w \in$

```
/* Given a DTM, M = (Q,T,I,δ,β,q0,qf) and w ∈ I* */

    Generate variable declaration and initialization                 Figure 2
    Generate code for creating the initial Instantaneous Description  Figure 3
    Generate code for simulating the transition function             Figure 4
    Generate code for validating the result                          Figure 4
s:
```

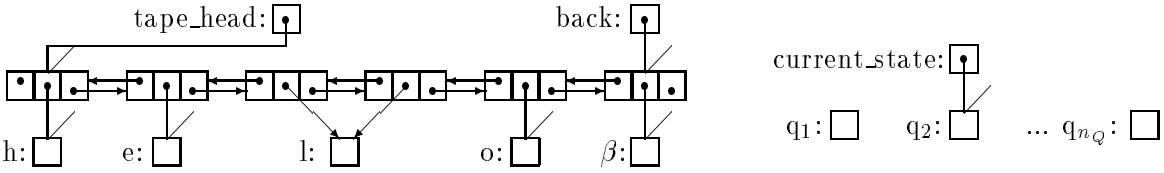
Figure 1: Outline of the machine *reduce*

I* such that ∗∗**current_state** is aliased to **valid_simulation** on some execution to program point s iff machine M halts on input w = $x_1 x_2 ... x_{n_w}$.

Lemma 2.1 *The code generated by Figure 3 creates the data structure that represents the initial configuration of M ($\underline{q_0} x_1 x_2 ... x_{n_w}$) in the manner described in Section 2.1.*

This is evident from inspection of the code. "`current_state = &q0;`" points **current_state** to $\mathbf{q}_0$. **tape_head** points to the first element of the linked list which corresponds to the tape head of M being on the beginning of the tape. Finally, NEXT_SYM(i) points the "sym" field of the i^{th} element of the linked list to the variable representing the i^{th} symbol on the tape (x_i). This is exactly what is required by Section 2.1 for the initial ID $\underline{q_0} x_1 x_2 ... x_{n_w}$. □

As an example, consider the case where T = {h,e,l,o,β}, w = "hello", and $q_0 = q_2$. The initial ID is $\underline{q_2} hello$ and the code generated by Figure 3 produces:



Notice that **back** points to the end of the linked list representing the tape. This allows us to add a new tape element to the end of the tape and, as seen later, is used to ensure that we never run off the right end of the tape.

The declaration and initialization of the variables of the program used to simulate a DTM are specified in Figure 2. All the variables except **yes**, **no**, **not_valid**, and **valid_simulation** have

```
/* Given a DTM, M = (Q,T,I,δ,β,q0,qf) and w ∈ I* */
        typedef int **state;
        typedef int **letter;
        struct tape {
           letter *sym;
           struct tape *next,*prev;
        } *back,*tape_head;
        state *current_state;
        int not_valid,valid_simulation;
        int *yes = &valid_simulation;
        int *no = &not_valid;
        #for i=1 to n_Q
           state q_i = &yes;
        #endfor
        #for i=1 to n_T
           letter σ_i = &yes;
        #endfor
```

Figure 2: Variable declaration and initialization

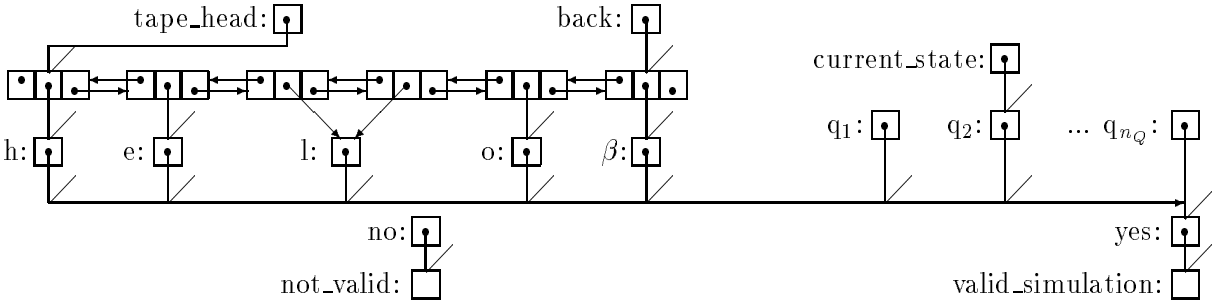
```
/* Given a DTM, M = (Q,T,I,δ,β,q0,qf) and w ∈ I* */
current_state = &q0;
back = malloc(sizeof(struct tape));
tape_head = back;
tape_head->prev = NULL;
tape_head->next = NULL;
tape_head->sym = &β;
/* initialize tape to w = x1x2...x_nw */
#for i = 1 to n_w
   back->next = malloc(sizeof(struct tape));   }
   back->sym = &x_i;                           }
   back->next->prev = back;                    } NEXT_SYM(i)
   back = back->next;                          }
   back->next = NULL;                          }
   back->sym = &β;                             }
#endfor
```

Figure 3: Initial Instantaneous Description (ID)

```
        /* Given a DTM, M = (Q,T,I,δ,β,q0,qf) and w ∈ I* */
      /* next_bool returns the next boolean value from the input stream. */
      back->next = malloc(sizeof(struct tape));
      back->next->prev = back;
      back = back->next;                                          ADD_TO_END
      back->next = NULL;
      back->sym = &β;
      #for i = 1 to nQ /* once for each state qi ∈ Q */
         #for j = 1 to nT /* once for each letter σj ∈ T */
            /* let δ(qi,σj) = (q'i,σ'j,d) */
            if (next_bool == 1) {
               qi = &no;
               σj = &no;
               **current_state = &not_valid;
               **(tape_head->sym) = &not_valid;
               current_state = &q'i;
               tape_head->sym = &σ'j;
               #if d = R                                           DELTA(i,j)
                  tape_head = tape_head->next;
               #endif
               #if d = L
                  tape_head = tape_head->prev;
               #endif
               qi = &yes;
               σj = &yes;
            } else
         #endfor
      #endfor
               yes = &not_valid; /* This is the else part of the last
                                  * if-then-else produced above. */
   }
   no = &not_valid; /* *no already is not_valid, but the
                     * assignment makes the proof simpler */
   #for i=1 to nQ                                                 CHECK_ANSWER
      qi = &no;
   #endfor
   qf = &yes;
s: /* **current_state is valid_simulation here iff M halts on w */
```

Figure 4: Representation of the transition function

already been explained in Section 2.1. Three of the four new variables are simply to record if the current path is a valid simulation of M. **Yes** points to **valid_simulation** iff the current path is a simulation of M and **yes** points to **not_valid** otherwise. The fourth variable (**no**) is just a "don't care" location for pointers that must refer to something other than **yes**. **No** always points to **not_valid**. Continuing our example, the initialization specified in Figure 2 followed by the code specified in Figure 3 for T = {h,e,l,o,β}, w = "hello", and $q_0 = q_2$, produces:



The remainder of the program created by *reduce* is mostly a while loop (Figure 4). Each pass through the body of the loop represents one application of the transition function (δ). The first part the the loop (ADD_TO_END) simply adds a new tape location, initially blank, to the right end of the tape. This ensures that whenever the simulated DTM moves right on the tape, a tape location is available to it. The remainder of the loop is a nested if-then-else-if

```
if (next_bool == 1) DELTA(1,1);
else if (next_bool == 1) DELTA(1,2);
...
else if (next_bool == 1) DELTA(nQ,nT);
else yes = &not_valid;
```

where DELTA(i,j) (explained below) is the code to implement $\delta(q_i, \sigma_j)$. The code

```
if (next_bool == 1) DELTA(1,1);
if (next_bool == 1) DELTA(1,2);
...
if (next_bool == 1) DELTA(nQ,nT);
```

would also be valid and does not require an artificially deep if-then-else-if nesting. However, in the former, a pass through the loop either is not a valid simulation or represents exactly *one* transition of the DTM M. In the later, a pass through the loop is an invalid simulation or represents a legal sequence of 0 to $n_Q * n_T$ transitions of M. The former is preferable because it makes the proof of correctness easier.

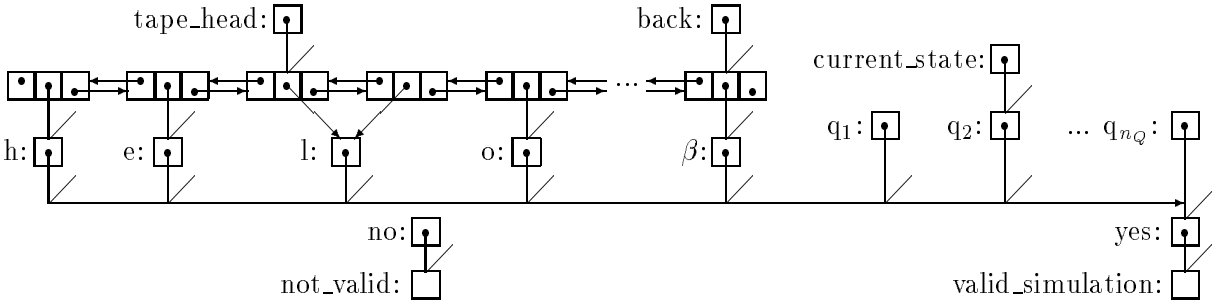
Lemma 2.2 *If* **yes** *points to* **not_valid** *before the loop specified in Figure 4 is executed then* **yes** *still points to* **not_valid** *after execution of the loop.*

Inspection of the code reveals that in the while loop only **&not_valid** can be assigned to **yes**. □

The next lemma basically states that if the execution is a valid simulation of M before executing the loop generated by Figure 4, then one path through the loop simulates the next transition of M and all other paths are not a valid simulation.

Lemma 2.3 *If before the loop specified in Figure 4 is executed,* **yes** *points to* **valid_simulation**, *the ID encoded by the alias pattern is* $x\underline{q_i}y$, *and* $x\underline{q_i}y \vdash_M x'\underline{q_j}y'$ *then on all but one path through the loop* **yes** *points to* **not_valid** *and on the remaining path,* **yes** *points to* **valid_simulation** *and the alias pattern represents the ID* $x'\underline{q_j}y'$.

We illustrate this proof with the example he$\underline{q_2}$llo $\vdash_M$ h$\underline{q_1}$eelo (therefore $\delta(q_2,l) = (q_1,e,L)$). For this example, the alias pattern before execution of the loop is:



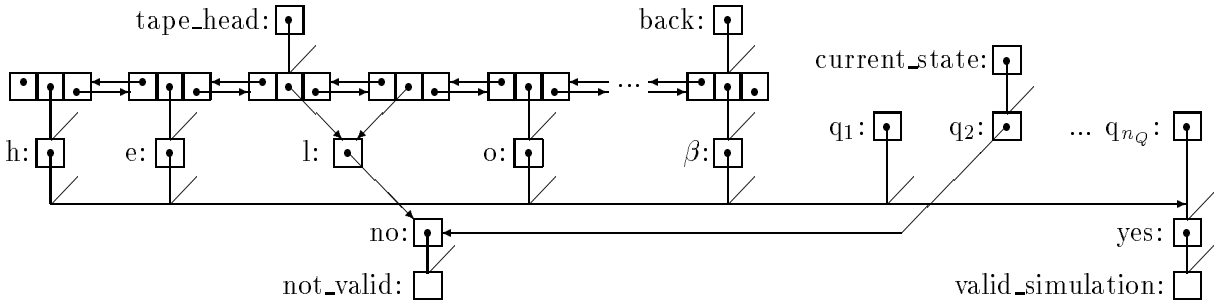
Before execution of the while loop, all $\sigma \in$ T and all $\mathbf{q} \in$ Q point to **yes**.[4] The first part of the loop, as stated earlier just expands the tape. The last *else*-clause simply points **yes** to **not_valid** signaling an invalid simulation. Now consider the code for DELTA(i,j) which represents the application of $\delta(\mathtt{q}_i, \sigma_j)$. Notice that we can only use this rule if M is in state $\mathtt{q}_i$ and the tape head is reading σ_j. Thus we would like to say (in pseudo-C)

$$\begin{array}{l}\texttt{if (*current\_state} \neq \mathtt{q}_i \textbf{ or } \texttt{*tape\_head->sym} \neq \sigma_j\texttt{)} \\ \qquad \text{yes} = \text{\&not\_valid } /* \textit{ i.e., not a valid simulation } */\end{array} \tag{1}$$

but this would create paths which are not executable in the program. The first four statements of DELTA(i,j) do exactly (1) without using a conditional. The statements "$\mathtt{q}_i$ `= &no;` σ_j `= &no`"

[4] We have not proven this here, but it follows from a simple inductive proof on number of iterations of the while loop.

point $\mathbf{q}_i$ and σ_j to **no**. All other states and alphabet symbols still point to **yes**. In our running example (where $q_i = q_2$ and σ_j = l) we have:



The statements, "`**current_state = &not_valid`" and "`**(tape_head->sym) = &not_valid`", make sure that the application of $\delta(q_i,\sigma_j)$ is applicable. There are two cases that can occur:

- The tape head is scanning σ_j (l) and M is in state q_i (q_2)

  This means that ***current_state** is **no** and ***(tape_head->sym)** is also **no**. Thus both of these statements effectively are "`no = &not_valid`" and the value of **yes** is unchanged and still points to **valid_simulation**. This is the path through the loop that is a valid simulation of M. For all i,j there is exactly one such path since M is a DTM.

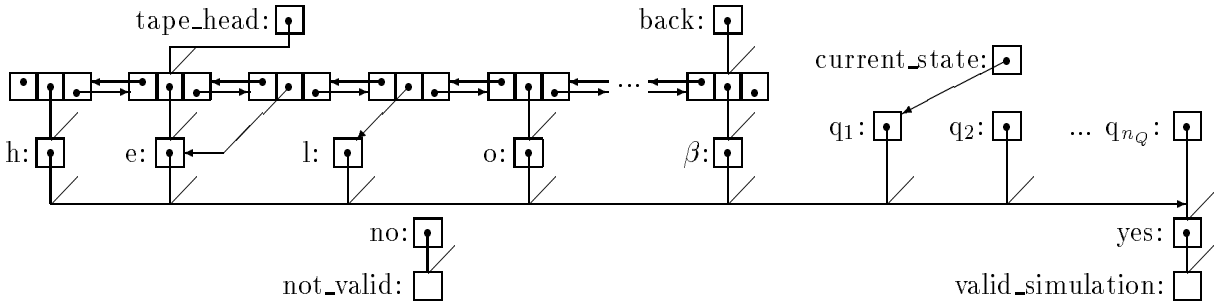
- Either the tape head is not scanning σ_j (l) or M is not in state q_i (q_2)

  In the first case, ***current_state** is **yes**. Thus "`**current_state = &not_valid`" causes **yes** to point to **not_valid** instead of **valid_simulation**. In the second case ***(tape_head->sym)** is **yes** and "`**(tape_head->sym) = &not_valid`" causes **yes** to point to **not_valid**. The lemma is satisfied regardless of the subsequent code because **yes** points to **not_valid**.

We now proceed assuming that M is scanning σ_j (l) and is in state q_i (q_2). Since $\delta(q_i,\sigma_j) = (q_i',\sigma_j',d)$ we want to shift to state q_i' ("`current_state = &`q_i'"), write σ_j' to the tape ("`tape_head->sym = &`σ_j'"), and move the tape head one unit in direction d:

```
#if d = R                                  #if d = L
   tape_head = tape_head->next;               tape_head = tape_head->prev;
#endif                                     #endif
```

Finally, the statements "q_i `= &yes;` σ_j `= &yes`" restore the condition that all state and alphabet variables point to **yes**. To finish our example where $\delta(q_2,l) = (q_1,e,L)$, the program store now is:



□

The rest of Figure 4 is addressed in the following lemma:

Lemma 2.4 *M halts on w iff on some path to s, in the program generated by reduce,* ∗∗**current_state** *is* **valid_simulation**.

Consider the program C generated by *reduce*. By an inductive argument it is easy to show that:

1. on all paths to the top of the while loop on which **yes** points to **valid_simulation**, if the alias pattern represents $x'\underline{q_{i'}}y'$ then $\underline{q_0}w \vdash_M^* x'\underline{q_{i'}}y'$.
2. if $\underline{q_0}w \vdash_M^* x\underline{q_i}y$ then there is at least one path[5] to the top of the while loop generated by Figure 4 on which **yes** points to **valid_simulation** and the alias pattern represents $x\underline{q_i}y$.

The proofs are by induction on the number of times the path has passed through the top of the loop and on number of steps M has taken to derive $x\underline{q_i}y$. In both proofs, the base case is shown by Lemma 2.1 and the induction step can be shown by appealing to Lemma 2.2 and Lemma 2.3. M halts on w iff $\underline{q_0}w \vdash_M^* x\underline{q_f}y$ (some x,y); thus M halts on w iff there exists a path to the top of the loop on which **yes** points to **valid_simulation** and **current_state** points to $\mathbf{q}_f$.

Consider the effects of CHECK_ANSWER in Figure 4.

- If **current_state** does not point to $\mathbf{q}_f$ then ****current_state** must be **not_valid**.
- If **yes** points to **not_valid** then ****current_state** must be **not_valid**.
- If **current_state** points to $\mathbf{q}_f$ and **yes** points to **valid_simulation** then ****current_state** must be **valid_simulation**.

This means that M halts on w iff on some path to s ∗∗**current_state** is **valid_simulation**. □

[5] exactly one path unless $x\underline{q_i}y \vdash_M^+ x\underline{q_i}y$.

3 May Alias is not recursive

Theorem 3.1 *Statically determining Intraprocedural May Alias for languages with if-statements, loops, dynamic storage, and recursive data structures is recursively enumerable but not recursive even when all paths through the program are executable.*

Consider the program C generated by *reduce*. All paths through C are executable. Consider any path P, let $c_1c_2...c_k$ be a sequence with a unique c_i for every conditional (i.e., *if* and *while* statements) on P in the order that they appear on P. Let c_i be 1 if the true branch of the corresponding conditional is taken on P, and let c_i be 0 otherwise. Clearly, $c_1c_2...c_k$ is an input that executes path P. By Lemma 2.4, M halts on w iff on some path to s in C $**$**current_state** is **valid_simulation**. Therefore, May Alias is not recursive as it can be used to solve the halting problem even for programs on which all paths are executable. May Alias is recursively enumerable since we can nondeterministically generate runs of a program and questions about aliasing during an execution can be answered by examining the symbol table and the program store. □

Theorem 3.2 *Statically determining Intraprocedural* **Must** *Alias for languages with if-statements, loops, dynamic storage, and recursive data structures is* **not recursively enumerable** *even when all paths through the program are executable.*

A quick look at the program produced by *reduce* (on which all paths are executable; see proof of Theorem 3.1) shows that **yes** either points to **not_valid** or **valid_simulation**, **no** always points to **not_valid**, **q** $\in$ Q always points to **yes** or **no**, and **current_state** always points to a state. This implies that $**$**current_state** is either **valid_simulation** or **not_valid**. Since we already showed that M halts on w iff on some path to s $**$**current_state** is **valid_simulation** (Lemma 2.4), it follows that M does not halt iff $**$**current_state** is **not_valid** on all paths to s. Thus, Must Alias can be used to solve the complement of the halting problem which is not recursively enumerable. This means that Must Alias is not recursively enumerable given the language requirements stated by the theorem. □

4 Conclusion

We have shown that intraprocedural May Alias is not recursive and intraprocedural Must Alias is not recursively enumerable even for programs on which all paths are executable given the language has dynamically allocated recursive data structures, loops, and if-statements. This is an extremely negative result, considering that a doubly linked list was the only dynamic data structure needed. Unfortunately, the proofs can be modified to use only a singly linked list. In Appendix A, we show how to modify *reduce* so that only a singly linked list is needed.

These results do not imply that Static Analysis is dead. They simply mean that, as has been known all along, some approximation must be done. What the new results do show is that, even if we are allowed to write exponential algorithms, Static Analysis would still have to be approximate. It probably also means that, in the presence of dynamically allocated recursive data structures, we have to accept approximations of lesser quality, which also take more time and space to compute than in FORTRAN. One final implication of our results is that while determining the structure of dynamic data structures (i.e., is it a tree? linked list? ...) is important, it is not sufficient to make Static Analysis recursive, because our proofs used only programs with linked lists.

Currently, Static Analysis is in the same situation as it was for FORTRAN 20 or so years ago. We are dealing with a problem which is not recursive, and need to come up with a good set of simplifying assumptions and algorithms that yield reasonably good approximations quickly and cheaply. In most work in Static Analysis, these assumptions are being introduced in an ad hoc manner and are often not even explicitly stated. In the future, it would be nice to have simplifying assumptions that would work for Static Analysis is general. One such assumption is that only polynomial length paths need be considered. This seems reasonable, as we already assume that programs "terminate normally" (for example, no division by 0), and non-polynomial executions are generally not tolerated. The nice result of this assumption is that it makes most Static Analyses $\mathcal{NP}$ complete (see Appendix B). On the other hand, it is difficult to see how this would influence the development of approximate algorithms.

Acknowledgments: We thank Rita Altucher, Bruce Ladendorf, Tom Marlowe, Michael Plat-off, and the reviewers for their comments on this material.

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1976.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1977), pp. 238–252.

[4] Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[5] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

[6] Horwitz, S., Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction* (June 1989), pp. 28–40.

[7] Kam, J. B., and Ullman, J. D. Global flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (1976), 158–171.

[8] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7* (1977), 305–317.

[9] Kildall, G. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Jan. 1973), pp. 194–206.

[10] Landi, W. *Interprocedural Aliasing in the Presence of Pointers.* PhD thesis, Rutgers University, Jan. 1992. LCSR-TR-174.

[11] Landi, W., and Ryder, B. G. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1991), pp. 93–103.

[12] Larus, J. R. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California Berkeley, May 1989.

[13] Larus, J. R., and Hilfinger, P. N. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (July 1988), pp. 21–34. SIGPLAN NOTICES, Vol. 23, No. 7.

[14] Myers, E. M. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1981), pp. 219–230.

[15] Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2*, 3 (1973), 191–213.

[16] Wegman, M., and Zadeck, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (Apr. 1991), 181–210.

A Modification to singly linked list

The machine *reduce* (Figure 1) is modified to construct a program with only a singly linked list and that still satisfy Theorem 3.1 and Theorem 3.2 as follows:

- Remove **prev** field from **struct tape**. Also remove all statements dealing with **prev**.
- Add the field "**int** $**$**not_at**" to type **struct tape**. Point **not_at** to **yes** whenever a new tape element is created.

- Add a variable "**struct tape** ∗**front**" which will always point to the first (leftmost) element of the linked list.
- Immediately before the while loop that simulates δ add the statement:

```
tape_head->not_at = &no
```

  In general, the **not_at** field of the linked list representing the tape points to **yes** except at **tape_head** where it points to **no**.
- Replace the code for when `d = R` in the loop with:

```
#if d = R
   tape_head->not_at = &yes;
   tape_head = tape_head->next ;
   tape_head->not_at = &no;
#endif
```

- Finally, replace the code for when `d = L` in the loop with:

```
#if d = L
   tape_head = front;
   while (next_bool == 1) {
      ADD_TO_END /* The code for ADD_TO_END here */
      tape_head = tape_head->next;
   }
   /* tape_head->next->not_at points to yes and this
    * assignment signals invalid simulation unless
    * tape_head is one LEFT of where it was */
   *(tape_head->next->not_at) = &not_valid;

   tape_head->next->not_at = &yes;
   tape_head->not_at = &no;
#endif
```

B Polynomial paths only

When only polynomial length paths are considered, May Alias and the complement of Must Alias are $\mathcal{NP}$ complete for programs on which all paths are executable given the language has dynamically allocated recursive data structures, loops, and if-statements. These problems are in $\mathcal{NP}$ because we can nondeterministically generate an execution of the program and then simulate the execution in time linear in the length of the execution (a similar argument is in [12]). From the program store

we can easily answer questions about May Alias and the complement of Must Alias. This argument works for Static Analysis problems which can be nondeterministically answered for a given execution in a polynomial amount of time, in the size of the program and length of the execution path, from the program, symbol table, and a possibly augmented program store. However, the augmented store must be polynomial in the size of the original store; for example, [6].

The fact that May Alias and the complement of Must Alias are both $\mathcal{NP}$ hard under this new assumption is evident because our original proofs of this claim [11, 10] do not contain any loops and only linear length paths exist. Since most Static Analyses are influenced by aliases, they are also $\mathcal{NP}$ hard.