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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a free and/or specific permission.
©1992 ACM 1057-4514/92/1200-0323$01.50

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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*.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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:

> $\textit{may-alias}_P(s, \langle x, y \rangle)$ 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 $(Q,T,I,\delta,\beta,q_0,q_f)$ 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]

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

- $\beta \in$ 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

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

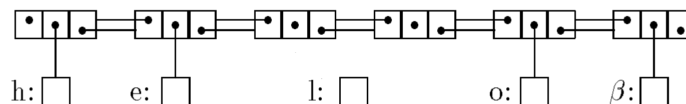
2.1 Representing an ID

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 $x\underline{q_i}y \in \mathrm{T}^*\mathrm{Q}\mathrm{T}^*$ 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: | prev | sym | next | . 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:

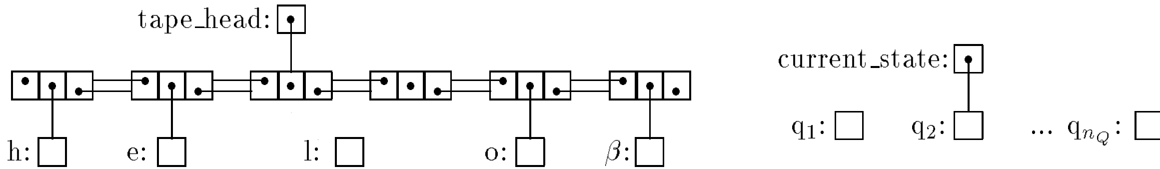


[2] L moves tape head left, R moves tape head right, and S leaves the tape head where it is.

[3] q_i is underlined in $x\underline{q_i}y$ to make the state stand out from the tape string.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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 = heq_2llo 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                       x1 = 1;
   xi = i;           } represents {   x2 = 2;
#endfor                               x3 = 3;
```

```
#for i = 1 to 3                       x1 = 1;
   xi = i;                            y1 = 1;
   #if i is odd      } represents {   x2 = 2;
      yi = i;                         x3 = 3;
   #endif                             y3 = 3;
#endfor
```

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 $\in$

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.