Static Program Analysis

[Based on Tom Reps' lecture notes.]

Abstract

This lecture introduces the area of static program analysis. We introduce the topics to be reviewed in this unit, and say a bit about abstract interpretation.

1 Introduction to Static Program Analysis

Static program analysis is known by various terms, including static analysis, dataflow analysis, abstract interpretation, state-space exploration, model checking, and static bug finding. All of these terms are rough synonyms.

1.1 Brief Background

Static analysis tries to answer questions about a program's behavior without running the program on specific inputs. Many questions can be of interest, including

- Can variable x equal value v at label L?
- Must variable x equal value v at label L?
- What are the upper bounds on the value of variable x at label L?
- Is x initialized before it is first used?
- What is the sign of x at L?
- What is the parity of x at L?
- Can there be any overflows in the computation performed at L?
- Can the value of x be read before it is overwritten (during execution actions performed after reaching L)?
- Does $x = y$ hold at L?
- The questions answered by BTA/taint-analysis
- Can pointer p be NULL at label L?
- What variable(s)/heap-blocks can pointer p point to at L?
- What program points could have assigned to x the value read from x at L?
- Do pointers p and q point to disjoint heap-allocated data structures at L?

Traditionally static analysis was done to optimize program performance. However, with improving hardware predictors, static analysis yielded diminishing returns for increasing performance. More recently, say after 1998, static analysis has pivoted toward software engineering, program verification, and security where it can provide information about possible bugs and security vulnerabilities.

1.2 A Difficulty: Undecidability of Static Program Analysis

Unfortunately, all static-program-analysis problems of interest are undecidable. We can often prove such results by a reduction from the halting problem. For example, imagine a program analyzer *DecideConstant* that can decide the problem of testing the following property:

> Given program P, is it the case that on all runs of P, variable x has the same value at the end of execution?

We will depict *DecideConstant* as follows:

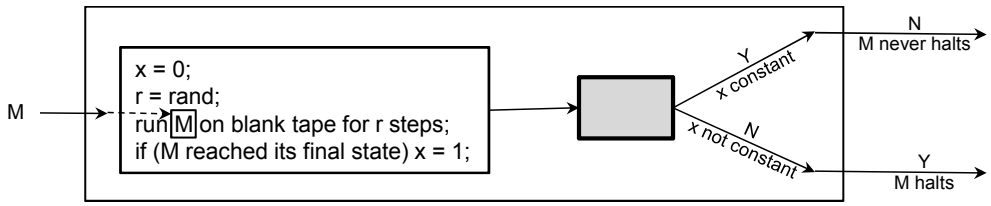
We now show that if *DecideConstant* exists, we can use it to create an algorithm that decides whether a given Turing machine halts when run on blank tape—which is known to be impossible.

Consider what happens if P is the following program, where we would like to know whether Turing machine M halts:

```
x = 0;
r = rand;
run M on blank tape for r steps
if (M reached its final state) x = 1;
```

(Without loss of generality, we are assuming that M's initial state is not a final state.)

As depicted below, if we feed this program P into *DecideConstant* and get back "Y," then we know that no matter what the value of r is, when P finishes x has the value 0. In other words, no matter how many steps M runs, it can never reach its final state, and hence it never halts on blank tape. Thus, the algorithm can return "N," meaning "M never halts on blank tape." If *DecideConstant* returns "N," then sometimes (the constructed) P finishes with $x = 0$ and sometimes with $x = 1$, and hence after some number of steps M halts on blank tape, so the algorithm can return "Y," meaning "M halts on blank tape." Such an algorithm is depicted below:



This construction contradicts the known fact that the question of whether a given Turing-machine halts on blank tape is undecidable, which means that our putative analyzer *DecideConstant* cannot exist.

To argue the undecidability of other static-analysis problems, one either uses a similar argument (i.e., reduction from the halting problem), or in some cases, reduction from Post's Correspondence Problem. For examples of the latter, see [1, 3, 2].

1.3 Sidestepping Undecidability: One-Sided Answers

One might think that the undecidability issue would put the kibosh on the whole idea of static program analysis. However, the field of static program analysis has an interesting approach to sidestepping the undecidability issue, namely, to only build analyzers that return "one-sided" answers.

At first glance, the one-sided nature of the answers may be concealed, because typically an analyzer returns a Boolean result, TRUE or FALSE, which doesn't sound "one-sided." There are really two kinds of such analyzers:

1. One kind behaves as follows: whenever TRUE is returned, the property is true; however, when FALSE is returned, the property can be either false or true.
2. Another kind behaves as follows: whenever FALSE is returned, the property is false; however, when TRUE is returned the property can be either false or true.

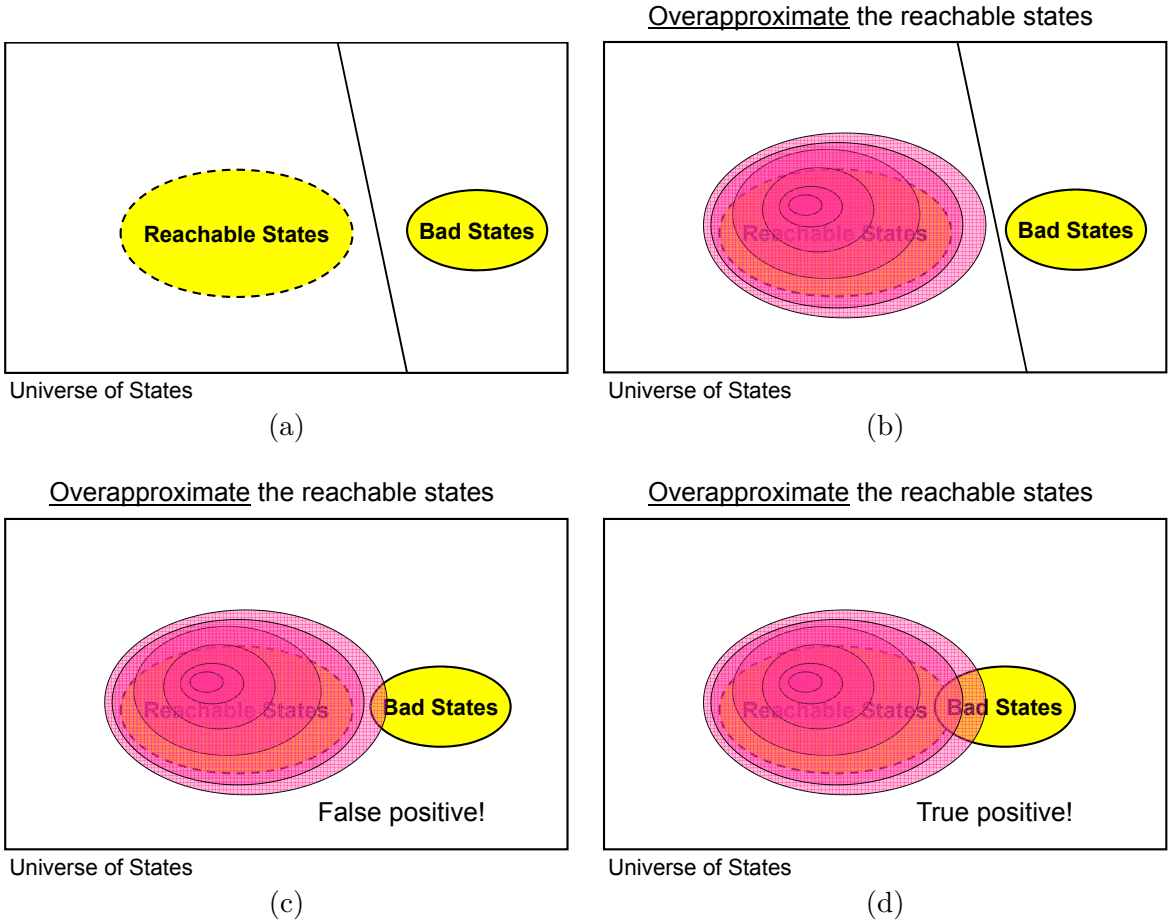


Figure 1: Abstract depiction of how a static program analyzer carries out state-space exploration. (a) For a correct program, there is a separation between the set of reachable states and the set of bad states. (b) A correct program, with an analysis that is capable of demonstrating a separation between the set of *apparently* reachable states and the set of bad states. (c) The analysis is too imprecise; an overlap between the set of apparently reachable states and the set of bad states would be reported, which is actually a false positive because there is no overlap between the set of reachable states and the set of bad states. (d) Another case in which a overlap is reported. In this case, there is an overlap between the reachable states and the set of bad states. Note that a client of the analysis would not know whether an overlap report falls into case (c) or case (d).

The description of the analyzer should make it clear which of TRUE or FALSE is a definite answer (and which is the indefinite answer.

Alternatively—and much less common—an analyzer can return one of three values: TRUE, FALSE, or UNKNOWN [4].

- Whenever TRUE is returned, the property is true.
- Whenever FALSE is returned, the property is false.
- Whenever UNKNOWN is returned, the property can be either true or false.

1.4 State-Space Exploration

The computational state of a program can be represented as a pair (*pp*, *store*), where *pp* is a program point (i.e., label) and *store* is a map from variables to their values. The universe of states for a

given program is the Cartesian product, $\{pp\} \times \{store\}$. A subset of this universe is the reachable states of the program. (Note that, in general, this subset cannot be identified exactly, otherwise we could solve the Halting Problem.)

Static analysis does not try to find the exact reachable states of the program. Instead, it tries to find an *over-approximation* (superset) of the reachable states and, we hope, prove that there exists a separation between this over-approximation and a given set of bad states. However, if the analysis over-approximates too much, the analyzer loses too much accuracy and we may not be able to prove the separation.

1.5 Topics to be covered

- Abstract Interpretation
- Meanings of descriptors
- Over-approximations
- Dataflow analysis
- Intra-procedural Analysis
- Inter-procedural Analysis
- Call-strings
  - approximations of stack configurations, using strings of bounded length
- Function summaries
  - no approximation with respect to the stack
  - in effect, call-strings with unbounded-length strings ("call-strings ∞")
- Graph reachability
  - related to function summaries
- Pushdown Systems
  - related to both call strings and function summaries
  - no approximation with respect to the stack
- Weighted Pushdown Systems

2 Abstract Interpretation

- Abstraction $\Rightarrow$ Replace values with descriptors of sets of values
- Interpretation $\Rightarrow$ Run the program over these descriptors.

The descriptors depend on the problem of interest, but a useful example is the domain of "environments of intervals." Each variable's value is represented by an interval $[a, b]$, and an environment of intervals is a map from variables to intervals, e.g.,

$$\begin{aligned} x &\mapsto [0, 3] \\ y &\mapsto [5, 10] \end{aligned}$$

Operations in the concrete domain—in which the original program is executed—have over-approximating analogs in the abstract domain.

$$(x = x + 1) \Rightarrow (X = X +^\sharp 1^\sharp) \tag{1}$$

For instance, if the pre-states to statement (1) are described by the interval environment given above, the post-states are described by the following interval environment:

$$\begin{aligned} x &\mapsto [1, 4] \\ y &\mapsto [5, 10] \end{aligned}$$

2.1 Concretization Function γ

Each family of descriptors has an operation γ, which takes a descriptor and returns the set of states in the concrete domain that the descriptor describes. For example,

$$\gamma((x \mapsto [0,3], y \mapsto [5,10])) = \{(0,5), (1,5)..(3,5), (0,6), (1,6)...(3,10)\}$$

2.2 Abstraction Function α

Suppose that at some program point, the only states possible are $(x \mapsto 0, y \mapsto 5)$ and $(x \mapsto 3, y \mapsto 10)$. By abstracting into the interval-environments domain, there is an unavoidable loss of precision. In this case, the most precise descriptor that includes both $(x \mapsto 0, y \mapsto 5)$ and $(x \mapsto 3, y \mapsto 10)$ is $(x \mapsto [0,3], y \mapsto [5,10])$; however, $\gamma((x \mapsto [0,3], y \mapsto [5,10]))$ includes far more than just the two states $(x \mapsto 0, y \mapsto 5)$ and $(x \mapsto 3, y \mapsto 10)$.

If α represents the abstraction function and C represents a set of states in the concrete domain, the loss of precision can be stated precisely in either of the following two ways:

- $\gamma(\alpha(C)) \supseteq C$, or
- $(\gamma \circ \alpha)(C) \supseteq C$.

This inequalities express the over-approximation inherent in abstract interpretation.

2.3 Correctness ("Soundness") of Abstract Interpretation

The essence of the correctness ("soundness") argument that one uses in abstract interpretation can be called *lock-step over-approximation*. That is, each operation in the abstract domain is in lock-step with an operation in the concrete domain.

$$\text{concrete function } f \Longrightarrow \text{abstract domain function } f^\sharp$$

Moreover, we want each $f^\sharp$ to over-approximate f in some fashion. To make this idea precise, in the concrete domain, we need to lift the function f to operate on a set of values (instead of singleton values) to form $\tilde{f}$.

The key idea for a soundness argument is expressed by the equation:

$$\tilde{f}(\gamma(a_1), \gamma(a_2)...\gamma(a_k)) \subseteq \gamma(f^\sharp(a_1, a_2..a_k)) \tag{2}$$

In words, applying $\tilde{f}$ to a set of concretized values $\gamma(a)$ will always be a subset of the concretization of applying $f^\sharp(a)$ in the abstract domain.

2.4 Abstract Containment

Let us define an ordering relation in the abstract domain, denoted by $\sqsubseteq$. For example,

$$(x \mapsto [0,3], y \mapsto [5,10]) \sqsubseteq (x \mapsto [-2,3], y \mapsto [4,20])$$

Both intervals on the right-hand side are larger—and thus represent supersets of—the intervals on the left-hand side. Soundness of an abstract interpretation can also be expressed as:

$$\alpha(\tilde{f}(c_1, c_2..c_k)) \sqsubseteq f^\sharp(\alpha(c_1), \alpha(c_2)...\alpha(c_k)). \tag{3}$$

(Eqns. (2) and (3) are really two ways of saying the same thing.)

2.5 Creating a Sound Abstract Interpreter

To create a sound abstract interpreter that uses a given abstract domain, we need to have some way to create the abstract analogs of

1. each constant that can be denoted in the programming language
2. each primitive operation in the programming language
3. each user-defined function in every program to be analyzed.

Task (1) is related to defining the α function; to create the abstract analog of a constant k, apply α: $k^\sharp = \alpha(k)$. By an "abstract analog" of an operation/function, we mean an abstract operation/function that satisfies Eqns. (2) and (3).

The effort that has to go into task (2) is bounded—the language has a fixed number of primitive operations—and task (2) only has to be done once for a given abstract domain. However, unless we have some automatic approach, the effort that would have to go into task (3) depends on the size of a user's program, which is not of *a priori* bounded size.

2.5.1 Syntax-Directed Replacement

Fortunately, Eqn. (3) enables a simple, compositional approach to be used—namely, syntax-directed replacement of the concrete constants and concrete primitive operations by their abstract analogs. For instance, consider the following function:

$$f(x, y) = x * y + 1$$

To see why Eqn. (3) enables a compositional approach to be applied, first hoist f to $\tilde{f}$, i.e.,

$$\tilde{f}(X, Y) = X \mathbin{\tilde{*}} Y \mathbin{\tilde{+}} \{1\}.$$

Then, by Eqn. (3), we have

$$\begin{array}{rcl} \alpha(\tilde{f}(X, Y)) & = & \alpha(X \mathbin{\tilde{*}} Y \mathbin{\tilde{+}} \{1\}) \\ & \sqsubseteq & \alpha(X \mathbin{\tilde{*}} Y) +^\sharp \{1\}^\sharp \\ & \sqsubseteq & \alpha(X) *^\sharp \alpha(Y) +^\sharp \{1\}^\sharp \end{array}$$

Consequently, one way to ensure that we have a sound $f^\sharp$ is to define $f^\sharp(A, B)$ by

$$f^\sharp(A, B) \stackrel{\text{def}}{=} A *^\sharp B +^\sharp \{1\}^\sharp.$$

2.5.2 Challenges for Abstract Interpretation of Functional Languages

The runtime environment for an abstract interpreter must be different from that of a concrete interpreter. Note that the two interpreters are doing different things: the concrete interpreter works with *singleton values*; the abstract interpreter works with (descriptors of) *sets of values*.

Consider some of the issues that come up, even if we are dealing with a pure functional language.

- *The need to execute "both ways" at a branch*: The expression in a conditional expression will be evaluated in the abstract domain to produce a value in the domain of *AbstractBoolean*s, which is often 3-valued: $\{\text{TRUE}, \text{FALSE}, \text{UNKNOWN}\}$. In the case of UNKNOWN, we will have to consider both branches of the conditional (and take the join of the two results).
- How does the abstract interpreter detect termination of the analysis process? (We could be analyzing a recursively defined function.)

2.5.3 Example

Consider the following concrete arithmetic operations:

$+$	0	1	2	3	...
0	0	1	2	3	...
1	1	2	3	4	...
2	2	3	4	5	...
3	3	4	5	6	...
⋮	⋮	⋮	⋮	⋮	⋱

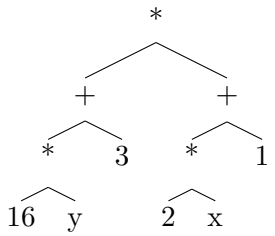
$\times$	0	1	2	3	...
0	0	0	0	0	...
1	0	1	2	3	...
2	0	2	4	6	...
3	0	3	6	9	...
⋮	⋮	⋮	⋮	⋮	⋱

Now consider the abstraction of sets of numbers to the *parity abstract domain*. There are three values: E, O, and ?, standing for "even," "odd," and "unknown." The abstract operations $+^\sharp$ and $\times^\sharp$ are defined as follows:

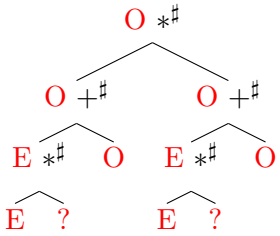
$+^\sharp$	?	E	O
?	?	?	?
E	?	E	O
O	?	O	E

$\times^\sharp$	?	E	O
?	?	E	?
E	E	E	E
O	?	E	O

Now consider the following function: $g(x, y) = (16 * y + 3) * (2 * x + 1)$. The abstract syntax tree (AST) for the defining right-hand-side expression is as follows:



Consider an evaluation over the AST using the parity abstract domain, where abstract values at leaves and internal nodes are shown as Red annotations:



2.5.4 Drawbacks of Syntax-Directed Replacement

Although the syntax-directed-replacement approach is simple and compositional, it can be quite myopic as it focuses solely on what happens at a single production in the abstract syntax tree. The approach can lead to a loss of precision by not accounting for correlations between operations an far-apart positions in the abstract syntax tree.

To illustrate the issue, consider the function

$$h(x) \stackrel{\text{def}}{=} x + (-x).$$

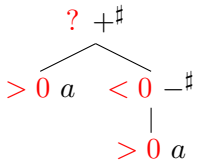
Obviously, $h(x)$ always returns 0. Suppose that we apply the syntax-directed method described in §2.5.1.

$$\begin{array}{rcl} \tilde{h}(X) & = & X \mathbin{\tilde{+}} (\tilde{-}X) \\ h^\sharp(X) & \stackrel{\text{def}}{=} & X +^\sharp (-^\sharp X) \end{array} \tag{4}$$

Now suppose that we evaluate over the *sign abstract domain*, which consists of six values: $\{< 0, 0, > 0, \leq 0, \geq 0, ?\}$. In particular, the abstract unary-minus operation is defined as follows:

a	$-^\sharp a$
$?$	$?$
≥ 0	≤ 0
≤ 0	≥ 0
> 0	< 0
0	0
< 0	> 0

Consider evaluating $h^\sharp(a)$ with the abstract value > 0 for the value of a. (Again, abstract values at leaves and internal nodes of the AST of the right-hand-side expression are shown as Red annotations.)



In other words, we get no information from the abstract interpreter, whereas the concrete value is always 0, and therefore the most-precise abstract answer would have been 0 (because $\alpha(\{0\}) = 0$).

References

[1] G. Ramalingam. The undecidability of aliasing. *Trans. on Prog. Lang. and Syst.*, 16(5):1467–1471, 1994.

[2] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *Trans. on Prog. Lang. and Syst.*, 22(2):416–430, 2000.

[3] T. Reps. Undecidability of context-sensitive data-dependence analysis. *Trans. on Prog. Lang. and Syst.*, 22(1):162–186, January 2000.

[4] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.