Harvard School of Engineering and Applied Sciences — Computer Science 152

Abstract interpretation

Lecture 20 Thursday, April 8, 2010

---

1 Static program analyses

For the last few weeks, we have been considering type systems. The key advantage of type systems is that they provide a static semantics to a program, allowing us to reason about all possible executions of a program before actually running the program. However, in order to achieve this, type systems must approximate the runtime behavior of a program. Type systems are (typically) flow-insensitive: the type of a variable (i.e., the static information associated with the runtime value of a variable) is the same for the entire execution of the program, regardless of which control flow path the program takes.

Abstract interpretation, also called dataflow analysis, is another form of static semantics, allowing us to reason about the behavior of a program before executing it.

Like type systems, abstract interpretation can be used to prove statically that various facts hold true for all possible dynamic executions of a program. In this regard, they can provide greater assurance than testing. Testing can verify the behavior of a program on one, or several, possible execution, but it is difficult to use tests to provide assurance that *all* possible executions of a program will be acceptable.

2 Abstract interpretation

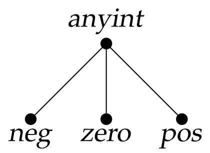
In abstract interpretation we define an *abstract domain*, an *abstraction* or approximation, of the real values manipulated by a program, and define a semantics for the language that uses the abstract domain. The key idea is that the abstract semantics for the language (which manipulates values from the abstract domain) is a faithful approximation of the concrete semantics for the language (which manipulates values from the concrete domain). By requiring the abstract domain to be “suitably finite” we can ensure that the abstract interpretation of a program is guaranteed to terminate.

We consider abstract interpretation in IMP. Moreover, we’ll introduce the key ideas by considering an analysis that computes the signs of variables.

We start by defining an abstract domain for integers, that captures their sign.

$$\mathbf{AbsInt} = \{pos, zero, neg, anyint\}$$

In contrast to the concrete domain **Int** of possible integer values, the abstract domain **AbsInt** is clearly finite. Furthermore, we can give a partial order to the elements of **AbsInt** as follows:



In this diagram (which is called a *Hasse diagram*), if two elements are connected by an edge, then the element higher up is a conservative approximation of the other element. We say that the element lower in the diagram is more precise the element higher up, and write “$x \sqsubseteq y$” to denote this . For example, *pos* is more precise than *anyint*, and indeed, knowing that an integer is positive is more precise than just knowing it is some integer.

The ordering is a *partial order*: it is reflexive, anti-symmetric, and transitive. Note that some elements of a partial order may be incomparable. Indeed, *pos* and *neg* are incomparable: neither is more precise than the other.

A partial order in which every pair of elements has a least upper bound is called a *join semi-lattice*. The least upper bound of a pair of elements a and b is called the *join* of a and b, and is written $a \sqcup b$.

When we said earlier that the abstraction needs to be "suitably finite", what we really meant is that the join semi-lattice of abstract values should have finite height; it may have an infinite number of elements, but the height needs to be finite.

Given the abstract values **AbsInt**, we define abstract stores **AbsStore** to be functions from variables to abstract values.

$$\sigma \in \textbf{AbsStore} = \textbf{Var} \to \textbf{AbsInt}$$

We can define a semantics for IMP that executes a program using the abstract store, and abstract values. This is abstract interpretation: the analysis forgets the concrete values of variables, and instead uses an approximation to their value; in this case, we are using the sign of integers.

Let's start defining this semantics, using denotational semantics.

2.1 Arithmetic expressions

For an arithmetic expression, the analysis uses an abstract store to derive a sign for that expression. The analysis tries to derive a precise sign for the expression. However, due to the lack of knowledge about concrete values of variables, in certain cases it may be conservative and say that the sign of the expression is unknown (represented as *anyint*). We express the analysis of arithmetic expressions using an abstract denotation $\mathcal{A}'[\![a]\!]$.

$$\mathcal{A}'[\![a]\!] : \textbf{AbsStore} \to \textbf{AbsInt}$$

$$\mathcal{A}'[\![n]\!]\sigma = \begin{cases} pos & \text{if } n > 0 \\ zero & \text{if } n = 0 \\ neg & \text{if } n < 0 \end{cases}$$

$$\mathcal{A}'[\![x]\!]\sigma = \sigma(x)$$

$$\mathcal{A}'[\![a_1 + a_2]\!] = \begin{cases} pos & \text{if } (\mathcal{A}'[\![a_1]\!]\sigma = pos \wedge \mathcal{A}'[\![a_2]\!]\sigma \in \{zero, pos\}) \\ & \qquad \vee(\mathcal{A}'[\![a_1]\!]\sigma = zero \wedge \mathcal{A}'[\![a_2]\!]\sigma = pos) \\ neg & \text{if } (\mathcal{A}'[\![a_1]\!]\sigma = neg \wedge \mathcal{A}'[\![a_2]\!]\sigma \in \{zero, neg\}) \\ & \qquad \vee(\mathcal{A}'[\![a_1]\!]\sigma = zero \wedge \mathcal{A}'[\![a_2]\!]\sigma = neg) \\ zero & \text{if } \mathcal{A}'[\![a_1]\!]\sigma = zero \wedge \mathcal{A}'[\![a_2]\!]\sigma = zero \\ anyint & \text{otherwise} \end{cases}$$

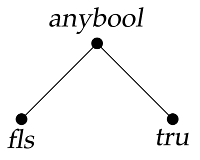
Note that all of these evaluations use just the information in the abstract store when reasoning about variables. The evaluation has no knowledge about the concrete values of variables. In the last case, the analysis cannot precisely determine the sign of the expression, and conservatively returns *anyint*.

2.2 Boolean expressions

We can also define a similar evaluation for boolean expressions. We use the following abstract domains for booleans.

$$\textbf{AbsBool} = tru, fls, anybool$$

This domain includes the value *anybool*, indicating that the precise truth value of an expression cannot be precisely determined. Again, we can define an ordering between these values and form a join semi-lattice domain:



$$\mathcal{B}'[\![b]\!] : \textbf{AbsStore} \rightarrow \textbf{AbsBool}$$

$$\begin{aligned}\mathcal{B}'[\![\textbf{true}]\!]\sigma &= tru\\ \mathcal{B}'[\![\textbf{false}]\!]\sigma &= fls\end{aligned}$$

The evaluation $\mathcal{B}'[\![a_1 < a_2]\!]\sigma$ must evaluate the signs of subexpressions a_1 and a_2 (using the denotation for arithmetic expressions) and try to determine the truth value of the comparison based on those signs.

$$\mathcal{B}'[\![a_1 < a_2]\!]\sigma = \begin{cases} tru & \text{if } (\mathcal{A}'[\![a_1]\!]\sigma = neg \wedge \mathcal{A}'[\![a_2]\!]\sigma \in \{zero, pos\} \\ & \quad \vee(\mathcal{A}'[\![a_1]\!]\sigma = zero \wedge \mathcal{A}'[\![a_2]\!]\sigma = pos) \\ fls & \text{if } (\mathcal{A}'[\![a_1]\!]\sigma = zero \wedge \mathcal{A}'[\![a_2]\!]\sigma \in \{zero, neg\} \\ & \quad \vee(\mathcal{A}'[\![a_1]\!]\sigma = pos \wedge \mathcal{A}'[\![a_2]\!]\sigma \in \{zero, neg\}) \\ anybool & \text{otherwise} \end{cases}$$

2.3 Commands

To finish up the analysis that determines the sign of variables, we must define how the analysis processes commands. For each command c, we want to execute c in some abstract store before c, and determine an abstract store after the command. At each point during the analysis of c, we will keep track of the current abstraction at that point, but we will have no information about the concrete store (i.e., the concrete integer values of variables). We express the analysis for a command c using an abstract denotation $\mathcal{C}'[\![c]\!]$.

$$\mathcal{C}'[\![c]\!] : \textbf{AbsStore} \rightarrow \textbf{AbsStore}$$

Note that $\mathcal{C}'[\![c]\!]$ is a total function, in contrast to the concrete semantics $\mathcal{C}[\![c]\!]$, which is a partial function. (Why?) The analysis of skip, assignment, and sequence is straightforward.

$$\begin{aligned}\mathcal{C}'[\![\textbf{skip}]\!]\sigma &= \sigma\\ \mathcal{C}'[\![x := a]\!]\sigma &= \sigma[x \mapsto \mathcal{A}'[\![a]\!]\sigma]\\ \mathcal{C}'[\![c_1; c_2]\!]\sigma &= \mathcal{C}'[\![c_2]\!](\mathcal{C}'[\![c_1]\!]\sigma)\end{aligned}$$

More interesting is the analysis of an **if** command. If we can determine which branch is being taken given the current sign information, then we only need to execute the appropriate branch. However, if we cannot precisely determine the truth value of the condition, then we must conservatively analyze each branch in turn and then combine the results:

$$\mathcal{C}'[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma = \begin{cases} \mathcal{C}'[\![c_1]\!]\sigma & \text{if } \mathcal{B}'[\![b]\!]\sigma = tru \\ \mathcal{C}'[\![c_2]\!]\sigma & \text{if } \mathcal{B}'[\![b]\!]\sigma = fls \\ (\mathcal{C}'[\![c_1]\!]\sigma) \sqcup (\mathcal{C}'[\![c_2]\!]\sigma) & \text{if } \mathcal{B}'[\![b]\!]\sigma = anybool \end{cases}$$

Note that the last case is the one that we usually expect in real programs – the case when we cannot statically tell which branch is going to be executed (usually because the test depends on some input to the program). The first two cases show that one branch is always taken; then, we can optimize the other branch away, and replace the **if** command with branch being taken.

The join operation over two abstract stores, just takes the join of each variable. That is, suppose that $\sigma'' = \sigma \sqcup \sigma'$; then for any variable x, we define $\sigma''(x) = \sigma(x) \sqcup \sigma'(x)$.

Note that the semantics for a conditional yields the most precise sign of a variable given its signs on the consequent and alternative of the conditional. If a variable x is positive on both branches, then x will map to *pos* in the resulting abstract store; but if x has unknown sign on one branch, or if it has different signs on the two branches, then x will map to *anyint* in the resulting abstract store.

2.4 Example programs

Let consider a few examples. Let's try the following sequence of assignments:

$$x := 4; y := -3; z := x + y$$

If we start with a concrete store where all variables are initialized to zero ($\sigma_0 = [x \mapsto 0, y \mapsto 0, z \mapsto 0]$), then the concrete execution of c yields:

$$\mathcal{C}[\![c]\!]\sigma = [x \mapsto 4, y \mapsto -3; z \mapsto 1]$$

Now if we start with abstract store $\sigma_1 = [x \mapsto zero, y \mapsto zero, z \mapsto zero]$ and we analyze the program using the sign abstraction, we get:

$$\mathcal{C}'[\![c]\!]\sigma_1 = [x \mapsto pos, y \mapsto neg; z \mapsto anyint]$$

Note that the analysis result $\mathcal{C}'[\![c]\!]\sigma$ is faithful to the concrete result $\mathcal{C}[\![c]\!]\sigma$: the sign of each variable correctly describe the actual value of that variable. However, the analysis may yield results that, despite being correct, are not very accurate; in this case for z. This is the price to pay for restricting ourselves to the abstract domain during the analysis.

Now consider a program with a conditional:

$$x := 4; y := 3; \textbf{if } x > y \textbf{ then } x := x + y \textbf{ else } y := y - x$$

If we start with the same concrete store σ_0, the program will take the true branch and yield:

$$\mathcal{C}[\![c]\!]\sigma_0 = [x \mapsto 7, y \mapsto 4]$$

Starting with an abstract store $\sigma_2 = [x \mapsto zero, y \mapsto zero]$, we get the following. The first two assignments will yield a store $\sigma_2' = [x \mapsto pos, y \mapsto pos]$. Based on these signs, the analysis cannot determine whether the test $x > y$ succeeds. Therefore, it analyzes each branch. On the true branch, the analysis can determine that $x + y$ is positive and maintain a positive sign for x:

$$\mathcal{C}'[\![x := x + y]\!]\sigma_2' = [x \mapsto pos, y \mapsto pos]$$

On the false branch, however, the analysis cannot determine the sign of $y - x$ since both x and y are positive, but their magnitudes are not known. Therefore, the analysis sets an unknown sign for y:

$$\mathcal{C}'[\![y := y - x]\!]\sigma' = [x \mapsto pos, y \mapsto anyint]$$

Finally, the analysis combines the results from the two branches to get the following final result:

$$\mathcal{C}'[\![c]\!]\sigma = [x \mapsto pos, y \mapsto anyint]$$

So the analysis could successfully determine that x is positive at the end of the program, regardless of the branch being taken; but it could not precisely determine the sign of y. As in the previous example, the analysis result correctly models the signs of the values computed in the actual execution of the program.

2.5 Analysis of while loops

Consider analysis of a while loop **while** b **do** c. In the concrete semantics, we would repeatedly evaluate the loop body c until the test b evaluates to **false**. Suppose we were executing the while loop with the abstract semantics. If, in the abstract semantics, we evaluate b to tru, then we know we need to execute the loop body. If, after executing the loop body zero or more times, b evaluates to the abstract value fls, then we know that we can stop executing the loop. But what if after executing the loop body zero or more times, b evaluates to *anybool*? That doesn't tell us whether we need to execute the loop body again or not!

To get an intuition for the semantics of a while loop, lets think about what properties we want of the abstract semantics. Intuitively, suppose we have some concrete store σ'_0, and we execute a program c, and it terminates with store σ'_1. And suppose that abstract store σ_0 is a faithful approximation of σ'_0, and σ_1 is the result of executing our abstract semantics. Then σ_1 should be a faithful approximation of σ'_1. The following commutative diagram shows this graphically.

$$\begin{tikzcd}
\text{Abstract semantics} & \sigma_0 \arrow[r, "\mathcal{C}'[\![c]\!]"] & \sigma_1 \\
\text{Concrete semantics} & \sigma'_0 \arrow[u, "\textit{faithful abstraction}"] \arrow[r, "\mathcal{C}[\![c]\!]"] & \sigma'_1 \arrow[u, "\textit{faithful abstraction}"']
\end{tikzcd}$$

Let's consider instantiating this with the while loop **while** b **do** c. For the concrete semantics to get from store σ'_0 to σ'_1, the loop body is executed zero or more times. So, starting with an abstract store σ_0, we want to find a σ_1 such that σ_1 is an abstraction of executing the loop body zero or more times.

To make sure that it is a suitable abstraction after executing the loop body zero times, we need to make sure that

$$\sigma_0 \sqsubseteq \sigma_1.$$

To make sure it is a suitable abstraction after executing the loop body once, we need to make sure that

$$\mathcal{C}'[\![c]\!]\sigma_0 \sqsubseteq \sigma_1.$$

Indeed, to make sure it is a suitable abstraction after executing the loop body n times, we need

$$\mathcal{C}'[\![c]\!]^n\sigma_0 \sqsubseteq \sigma_1.$$

Since, in a join semi-lattice, $a \sqcup b \sqsubseteq c$ if and only if $a \sqsubseteq c$ and $b \sqsubseteq c$, we can state our requirements succinctly:

$$\bigsqcup_{i\in\mathbb{N}} \mathcal{C}'[\![c]\!]^i\sigma_0 \quad \sqsubseteq \quad \sigma_1$$

Note that if σ_1 maps every variable to $anyint$, then this constraint is satisfied. But such an abstract state is useless: it tells us nothing about the sign of any variable. In fact, we want the most precise abstract store σ_1 that satisfies the requirements. That is, in fact, $\bigsqcup_{i\in\mathbb{N}} \mathcal{C}'[\![c]\!]^i\sigma_0$.

$$\mathcal{C}'[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \bigsqcup_{i\in\mathbb{N}} \mathcal{C}'[\![c]\!]^i\sigma$$

Note that $\sigma_1 = \bigsqcup_{i\in\mathbb{N}} \mathcal{C}'[\![c]\!]^i\sigma_0$ is a fixed point of $\mathcal{C}'[\![c]\!]$; that is $\mathcal{C}'[\![c]\!]\sigma_1 = \sigma_1$. We have an efficient way to find this fixed point:

```
$\mathcal{C}'[\![\textbf{while } b \textbf{ do } c]\!]\sigma =$ while (true)
    $\sigma' = \sigma \sqcup \mathcal{C}'[\![c]\!]\sigma$;
    if $\sigma' = \sigma$ then
        return $\sigma$
    else
        $\sigma = \sigma'$
```

This algorithm is guaranteed to terminate, since the abstract domain is of finite height. That is, each time through the loop, either we have reached a fixed point (and $\sigma' = \sigma$) or σ' is less precise than σ for at least one variable. But there are only a finite number of variables used in a program, and, with a finite-height domain, only a finite number of times any given variable can be made less precise. Thus, we are guaranteed to reach a fixed point eventually.