CS 411 Lecture 7 Program Analysis as Non-Standard Denotational Semantics 15 Sep 2003
Lecturer: Greg Morrisett

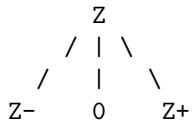
We can use a non-standard denotational semantics to reason about programs. The basic insight is that if we abstract from actual values, keep our abstractions suitably finite, and are always conservative, then we can compute conservative approximations of the flow of a given program.

For example, let us consider the abstract domain D defined as:

$$\mathsf{D} ::= \mathbb{Z} \mid \mathbb{Z}^+ \mid \mathbb{Z}^- \mid \mathsf{0}$$

where $\mathbb{Z}$ represents the set of all integers, $\mathbb{Z}^+$ represents the set of all positive integers, $\mathbb{Z}^-$ all negative integers, $\mathsf{0}$ represents the singleton set $\{0\}$, and $\emptyset$ represents the empty set of integers.

This is an abstraction of our concrete domain of integers, and it's only one abstraction that we could choose for a given analysis problem. It has the property that there is an "information ordering" in the sense $\mathbb{Z}$ is a superset of all of the other abstract domain elements. It has less information than something like $\mathbb{Z}^+$ or $\mathsf{0}$.



In principle, we can use any lattice to abstract the domain of values, but it helps to have a finite lattice or at least a lattice of finite height (no infinite ascending or descending chains.)

We can define a denotational semantics using this abstract domain as follows:

First, we use abstract stores which will be functions from variables to D instead of variables to integers. In other words, we're going to forget what specific integer value a variable holds and only remember whether it's zero, positive, negative, etc. If we don't know anything about the variable, then we'll have to assume that it's any possible integer (i.e., $\mathbb{Z}$).

$$S \in \mathsf{AbsStore} : \mathsf{Var} \to \mathsf{D}$$

Next, we provide an interpretation of expressions $\mathcal{E}'[\![\cdot]\!]$ that respects our abstraction:

$$\begin{aligned}
\mathcal{E}'[\![i]\!]S &= \begin{cases} 0 & \text{if } i = 0 \\ \mathbb{Z}^- & \text{if } i < 0 \\ \mathbb{Z}^+ & \text{if } i > 0 \end{cases} \\
\mathcal{E}'[\![x]\!]S &= S(x) \\
\mathcal{E}'[\![e_1 + e_2]\!]S &= \begin{cases} 0 & \text{if } \mathcal{E}'[\![e_1]\!]S = \mathcal{E}'[\![e_2]\!]S = 0 \\ \mathbb{Z}^+ & \text{if } (\mathcal{E}'[\![e_1]\!]S = \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^+) \vee \\ & \quad (\mathcal{E}'[\![e_1]\!]S = \mathbb{Z}^+ \wedge \mathcal{E}'[\![e_2]\!]S = 0) \vee \\ & \quad (\mathcal{E}'[\![e_1]\!]S = 0 \wedge \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^+) \\ \mathbb{Z}^- & \text{if } (\mathcal{E}'[\![e_1]\!]S = \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^-) \vee \\ & \quad (\mathcal{E}'[\![e_1]\!]S = \mathbb{Z}^- \wedge \mathcal{E}'[\![e_2]\!]S = 0) \vee \\ & \quad (\mathcal{E}'[\![e_1]\!]S = 0 \wedge \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^-) \\ \mathbb{Z} & \text{otherwise} \end{cases}
\end{aligned}$$

Note that evaluating an integer i returns either 0, $\mathbb{Z}^-$, or $\mathbb{Z}^+$ but not $\mathbb{Z}$. This reflects perfect information with respect to the abstraction. We're returning the most precise thing that we possibly can.

The variable case is just as in the standard denotational semantics – we lookup the value in the store. Of course, here, the store returns an abstract domain value instead of an integer.

Finally, we had to interpret the operation + in a way that's consistent with the domain. For instance, we can only say that we get a positive number out if we know that e_1 and e_2 yield non-negative numbers.

In general, the property that we want is that if we run the real semantics, whatever value we get out is contained in the set returned by the abstract semantics. Formally, we can say that an abstract store S is faithful to a concrete store s if for all variables x, $s(x)$ is an element of $S(x)$.

Then, we can say that $\mathcal{E}'[\![\cdot]\!]$ is faithful to $\mathcal{E}[\![\cdot]\!]$ if for expressions e, all stores s, and all abstract stores S that are faithful to s, $\mathcal{E}[\![e]\!]s \in \mathcal{E}'[\![e]\!]S$.

We can define a similar function $\mathcal{B}'[\![\cdot]\!]$ for boolean valued expressions. However, we'll need a new abstract domain for booleans:

$$\mathsf{D}_2 ::= \mathsf{True} \mid \mathsf{False} \mid \mathsf{DontKnow}$$

Here, DontKnow represents the set $\{\mathsf{true}, \mathsf{false}\}$ while True reprsents $\{\mathsf{true}\}$ and False $\{\mathsf{false}\}$. Then we can define $\mathcal{B}'[\![\cdot]\!]$ as follows:

$$\begin{aligned}
\mathcal{B}'[\![\mathsf{true}]\!] &= \mathsf{True} \\
\mathcal{B}'[\![\mathsf{false}]\!] &= \mathsf{False} \\
\mathcal{B}'[\![e_1 \leq e_2]\!] &= \begin{cases}
\mathsf{True} & \text{if } (\mathcal{E}'[\![e_1]\!]S = \mathbb{Z}^- \wedge \mathcal{E}'[\![e_2]\!]S \in \{0, \mathbb{Z}^+\}) \vee \\
& \quad (\mathcal{E}'[\![e_1]\!]S = 0 \wedge \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^+) \\
\mathsf{False} & \text{if } (\mathcal{E}'[\![e_1]\!]S = \mathbb{Z}^+ \wedge \mathcal{E}'[\![e_2]\!]S \in \{0, \mathbb{Z}^-\}) \vee \\
& \quad (\mathcal{E}'[\![e_1]\!]S = 0 \wedge \mathcal{E}'[\![e_2]\!]S = \mathbb{Z}^-) \\
\mathsf{DontKnow} & \text{otherwise}
\end{cases}
\end{aligned}$$

Next we need to define our abstract semantics for commands, $\mathcal{C}'[\![\cdot]\!]$. The first few cases work just as before:

$$\begin{aligned}
\mathcal{C}'[\![\mathsf{skip}]\!]S &= S \\
\mathcal{C}'[\![x := e]\!]S &= S[x \mapsto \mathcal{E}'[\![e]\!]S] \\
\mathcal{C}'[\![c_1; c_2]\!]S &= \mathcal{C}'[\![c_2]\!](\mathcal{C}'[\![c_1]\!]S)
\end{aligned}$$

We run into problems with if commands, because in general, we don't know which branch will be taken. In particular, if we have "if $e_1 \leq e_2$ then c_1 else c_2", and we only know that e_1 and e_2 are integers, then we don't know which branch will be selected. Thus, we must conservatively look at both branches to compute the possible output state, and we must somehow merge the information in the two output states to get a single abstract store.

$$\mathcal{C}'[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!]S = \begin{cases}
\mathcal{C}'[\![c_1]\!]S & \text{if } \mathcal{B}'[\![e]\!]S = \mathsf{True} \\
\mathcal{C}'[\![c_2]\!]S & \text{if } \mathcal{B}'[\![e]\!]S = \mathsf{False} \\
\mathsf{merge_stores}(\mathcal{C}'[\![c_1]\!]S, \mathcal{C}'[\![c_2]\!]S) & \text{otherwise}
\end{cases}$$

where we define $\mathsf{merge\_stores}(S_1, S_2)$ as:

$$\{(x, \mathsf{merge}(S_1(x), S_2(x))) \mid x \in \mathsf{Var}\}$$

and merge for two abstract domains as:

$$\begin{aligned}
\mathsf{merge}(\mathbb{Z}, _) &= \mathbb{Z} \\
\mathsf{merge}(_, \mathbb{Z}) &= \mathbb{Z} \\
\mathsf{merge}(X, X) &= X
\end{aligned}$$

In general, we calculate the union of the two sets and then find the smallest abstract domain element that is big enough to cover the union. Since $\mathbb{Z}$ contains everything, merging it with one of the other domains always yields $\mathbb{Z}$.

As an example, consider what we get out of analyzing:

$$\mathsf{if}\ x \leq 0\ \mathsf{then}\ x := x + 1\ \mathsf{else}\ \mathsf{skip}$$

If we assume on input that x is any negative integer (i.e., $S(x) = \mathbb{Z}^-$) then the analysis works as follows:

1. First, we need to compute $\mathcal{E}'[\![x]\!]S = S(x) = \mathbb{Z}^-$
2. Then we need to compute $\mathcal{E}'[\![0]\!]S = 0$
3. We know that all elements of $\mathbb{Z}^-$ are less than 0, so we only need to calculate $\mathcal{C}'[\![x := x + 1]\!]S$ and return that.
4. $\mathcal{C}'[\![x := x + 1]\!]S$ requires computing $\mathcal{E}'[\![x + 1]\!]S$. Since $\mathcal{E}'[\![x]\!]S = S(x) = \mathbb{Z}^-$, and $\mathcal{E}'[\![1]\!]S = \mathbb{Z}^+$, we can only conclude that the result is a $\mathbb{Z}$. Thus, $\mathcal{C}'[\![x := x + 1]\!]S = S[x \mapsto \mathbb{Z}]$.

Now let's analyze a different program:

$$\mathsf{if}\ x \leq 0\ \mathsf{then}\ x := 3\ \mathsf{else}\ x := -4$$

and assume on input that x is any integer (i.e., $S(x) = \mathbb{Z}$).

1. compute $\mathcal{E}'[\![x]\!]S = S(x) = \mathbb{Z}$.
2. compute $\mathcal{E}'[\![0]\!]S = 0$.
3. We can't tell which way the if goes since we don't know whether $x \leq 0$ (i.e., all elements of $\mathbb{Z}$ aren't less-than or equal to zero, nor are they all greater than 0.) So, we have to compute both possible outcomes and merge them.
4. Compute $\mathcal{C}'[\![x := 3]\!]S = S[x \mapsto \mathbb{Z}^+]$ since $\mathcal{E}'[\![3]\!]S = \mathbb{Z}^+$.
5. Compute $\mathcal{C}'[\![x := -4]\!]S = S[x \mapsto \mathbb{Z}^-]$ since $\mathcal{E}'[\![-4]\!]S = \mathbb{Z}^-$.
6. Merge the two output states $S[x \mapsto \mathbb{Z}^+]$ and $S[x \mapsto \mathbb{Z}^-]$ which results in $S[x \mapsto \mathbb{Z}]$ since in one case, x is positive and in another case it's negative. The most we can say is that after executing the if command, x is an integer.

So much for conditionals. What about while loops?

$$\mathcal{C}'[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]S = ???$$

Suppose we could guess an output S' for this. What properties should S' have? Well, it ought to be the case that S' is faithful to the original semantics. In particular, if we have a concrete store s and S is faithful to s (i.e., for all x, $s(x) \in S(x)$), and $\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]s = s'$, then it ought to be the case that s' is faithful to S'.

One thing we could guess for S' is the store that maps every variable to $\mathbb{Z}$. Let's call that store $\top$. $\top$ is certainly faithful. It also has the property that:

$$\mathcal{C}'[\![\mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}]\!]S \leq \top$$

where $S_1 \leq S_2$ means for all x, $S_1(x) \subseteq S_2(x)$. So, we could just use $\top$, but that's a little imprecise.

I claim that you can do something like the following:

```
fun loop S =
    let S' = merge_stores(S, C'[[c]]S)
    if S' = S then return S'
    else return loop S'
```

That is, start with the input state, compute $\mathcal{C}'[\![c]\!]S$ (the result of running the body of the while loop), and merge that with the input state to produce an S'. If S' is different than S (i.e., for some variable x, $S(x) \neq S'(x)$), then we try again but using S' as the input state.

Why would this work? First, note that:

1. $S_1 \leq \mathsf{merge\_stores}(S_1, S_2)$ and $S_2 \leq \mathsf{merge\_stores}(S_1, S_2)$ and
2. if $S_1 \leq S_2$, then $\mathsf{merge\_stores}(S, S_1) \leq \mathsf{merge\_stores}(S, S_2)$.

So, that tells us that after we go around the loop, S' is more abstract than S and is more abstract than $\mathcal{C}'[\![c]\!]S$. So, for instance, if $S'(x) = 0$, then it must be the case that $S(x) = 0$ and $\mathcal{C}'[\![c]\!]S(x) = 0$.

Second, suppose we've reached a point where $S' = S$. That is, $\mathsf{merge}(S, \mathcal{C}'[\![c]\!]S) = S$. Well, for one thing, it tells us that going around the loop again won't matter, because we'll always be getting the same S out. So we might as well stop.

Now I need to convince you of two things. First, using loop in this fashion will terminate (i.e., we'll eventually find a fixed point for the loop.) This is pretty easy to see – if we go around the loop, then we get something that is more abstract than we had on input. That means that for some variable, we went from 0,$\mathbb{Z}^+$, or $\mathbb{Z}^-$ to $\mathbb{Z}$. Now, the next time around the loop, that variable can't change. There can only be a finite number of variables that change (since the program can only mention a finite number of variables), so sooner or later, we'll stop. In this case, we should be able to stop within n iterations where n is the number of program variables. (If we had a deeper lattice, it might require more iterations.)

Technically, I need to convince you that this is a faithful approximation. I won't prove this formally, but it's pretty easy to see that for all finite unrollings of the while loop, we get out something that is a faithful approximation.

Homework 3

1. Given the following datatypes for IMP programs:

```
type var = string
datatype oper = Plus | Times | Minus | Divide
datatype exp = Var of var | Int of int | Oper of exp * oper * exp
datatype bexp = True | False | LessThanEq of exp * exp
datatype com = Skip | Assign of var * exp | Seq of com * com |
  If of bexp * com * com | While of bexp * com
```

   Write an analysis which conservatively determines whether or not the program can divide by zero. You should build a non-standard denotational semantics with an abstract domain that records whether an integer value can be negative, positive, zero, non-negative, non-positive, or simply an integer. That is, your lattice will have more elements than the example I gave above.

   You'll need to define suitable merge operations for abstract integers (and abstract booleans). You'll want to use abstract stores that are finite maps from variables to abstract integers (e.g., association lists or something from the SML library) and define a suitable merge_stores function.

2. Suppose we modified IMP so that we allowed assigning boolean values to variables, as well as integer values. For instance, we could write:

   $x :=$ true;
   $x := 1 + 42$;
   if x then $x := 42$ else $x :=$ true

   We want to allow variables to hold both integers and booleans, but we would like to warn the programmer if they ever use a variable in a way that is inconsistent with the way the variable was last assigned. In particular, if the variable was last assigned as an integer, but is used as a boolean, then this should generate a warning. Similarly, if the variable was last assigned a boolean but is then used as an integer, we should generate a warning.

   Write down an analysis (on paper – or as ML code) which generates these warnings.