CS611 Lecture Leture 37 **Abstract Interpretation** 1 December 2003

Scribe: Dan Williams Lecturer: Radu Rugina

1 Introduction to Abstract Interpretation

At this point in the course, we have looked at two aspects of programming languages: dynamic semantics and static program analysis.

1. *Dynamic Semantics* characterizes the dynamic execution of a program. Examples include operational and denotational semantics.

2. *Static Program Analysis* is a general term that refers to the collection of techniques and methods that allow us to reason statically at compile time about the program and extract information that is guaranteed to hold during all executions. This information can then be used for optimization and correctness.

There are two static analysis techniques which are most prominent:

1. *Type Systems*: If the compiler successfully type checks a program then it is guaranteed that there will not be any errors during execution. There are two aspects of type systems to keep in mind as we move on to abstract interpretation:

   (a) In type systems, programmers annotate the program with type information. We can regard these *type annotations* as global invariants provided by the user.

   (b) Types are *flow insensitive*. A variable or expression has the same type regardless of where it appears in the flow of the program. This is because the type annotations represent global properties.

2. *Abstract Interpretation*: We can think of abstract interpretation being similar to dataflow analysis, but also providing a framework that allows to formally prove the correctness of an analysis.

   The idea behind abstract interpretation (and dataflow analysis) is as follows. The execution of a program computes a piece of concrete information. The goal of abstract interpretation is to statically compute a piece of abstract information that characterizes the concrete information in all possible executions of the program.

   A good example of abstract interpretation is Sign Analysis, whose goal is to statically compute the possible signs of each variable at each program point. In this example, the concrete information represents the values of variables during program execution, i.e, the state. The abstract information models just the sign of variables.

   In contrast to type systems, abstract interpretation has the following features:

   (a) It is *flow sensitive*. It computes abstract information at each point in the program. The information at different points in the program may be different.

   (b) There are *no program annotations*. The analysis must compute the abstract info by itself, because it is unreasonable to ask the user to provide annotations at every point of every program. In particular, the compiler must find out the loop invariants; it is not easy to extract this information automatically.

How does the compiler statically figure out the abstract information that we are interested in? It "executes" the program in the abstract domain, hence the name Abstract Interpretation. Two differences compared to the concrete execution are the following:

1. The analysis must follow all possible paths through program (dynamic execution only follows one path).

2. The static analysis must terminate, even if the program doesn't. We expect the compiling process, including static analysis, to terminate even if our program has an infinite loop.

To summarize this introduction to abstract interpretation, we make the following comments. Type systems are a lightweight form of static analysis, which give some form of correctness (no errors) without much work. Abstract interpretation, on the other hand, is a heavyweight form of static analysis, giving detailed information at each point in the program. As a result, it provides a stronger sense of correctness and also enables optimizations, but at a larger cost.

2 Formal Framework

We are going to study abstract interpretation using an imperative language, in fact IMP. We will extend its syntax with *labels* to model program points:

$$c ::= [\mathbf{skip}]^l \ \mid \ [x := a]^l \ \mid \ c_0; c_1 \ \mid \ \mathbf{if}\, [b]^l\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \ \mid \ \mathbf{while}\, [b]^l\, \mathbf{do}\, c_0$$

where the labels $l \in \mathit{Labels}$ model program points. There is a special label l_{init}, representing the entry point in the program.

2.1 Lattices

We will formalize both the concrete and the abstract domain using lattices.
A complete lattice is a pair $(L, \sqsubseteq)$ such that:

- $\sqsubseteq$ is a partial order
- Any subset $X \subseteq L$ has lub $\sqcup X$ and glb $\sqcap X$.

A complete lattice is different than a cpo, in that a cpo only requires a lub for ω-chains.

2.2 Notation for Lattices

Regarded as binary operators, the lub and the glb are also referred to as join and meet. In this case, we use the infix notation:

- Join of two elements: $x \sqcup y$
- Meet of two elements: $x \sqcap y$

From its definition, a complete lattice is guaranteed to have a top and a bottom element:

- Top element: $\top = \sqcup L$
- Bottom element: $\bot = \sqcap L$

In many cases in the literature, lattices are sometimes denoted as tuples to emphasize all these operators and values: $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$

The intuition behind the partial ordering in the abstract lattice is that elements higher in the lattice are more precise. The most precise piece of abstract information is $\top$ and the least precise is $\bot$. Note that some books put it the other way around.

2.3 Properties about Operators

The operators $\sqcap, \sqcup$ and the partial order $\sqsubseteq$ satisfy the following properties:

- $x \sqcap y = x$ iff $x \sqsubseteq y$
- $x \sqcup y = y$ iff $x \sqsubseteq y$
- $\sqcap$ and $\sqcup$ are idempotent, commutative and associative.

2.4 Properties about Lattices

A property that not all lattices have, but which will be important in providing a guarantee that the static analysis will terminate is the *descending chain condition* (DCC). A lattice satisfies DCC is any descending chain stabilizes:

$$\{x_i\}_n \text{ s.th. } x_i \sqsupseteq x_{i+1} \quad \Rightarrow \quad \exists n_0 \text{ s.th. } \forall n \geq n_0.\ x_n = x_{n_0}$$

We can also define the *height* of a lattice as the maximum number of distinct elements in a chain. A finite height implies the descending chain condition (DCC).

3 Back to Abstract Interpretation

We formalize the result of abstract interpretation using a function *Result* which assigns two elements in the abstract lattice to each program point, before and after the point:

$$\mathit{Result} : \mathit{Labels} \rightarrow (L_a \times L_a)$$

We denote by $\mathit{Result}(\bullet\, l)$ the result right before the program point represented by l; and by $\mathit{Result}(l\, \bullet)$ the result right after l.

We next want to determine how the abstract information changes when a command is executed; in other words, how to "execute" a command in the abstract domain. For this, we introduce a *transfer function* for each command c, to map a piece of abstract information into another piece of abstract information:

$$[\![c]\!] : L_a \rightarrow L_a$$

We now formulate the problem as a constraint problem. To compute the *Result* function, we build the following constraint system (the constraints in the system are also know as *dataflow equations*):

$$\begin{aligned}
\mathit{Result}(l\, \bullet) &= [\![c]\!]\ \mathit{Result}(\bullet\, l) \\
\mathit{Result}(\bullet\, l) &= \sqcap\ \{ \mathit{Result}(l'\bullet) \mid l' \in\ \mathit{pred}(l)\ \},\ \text{where } l \neq l_{init} \\
\mathit{Result}(\bullet\, l_{init}) &= i_0
\end{aligned}$$

where *pred(l)* is the set immediate predecessors of l. Hence, *pred* describes the flow of control in the program and can be computed from the nested structure of sequencing commands, if commands, and while loops. Also, i_0 is the *boundary condition* – the abstract information at the entry point in the program.

We next describe informally the rules in the system. First, we have a constraint to compute the result after execution of c. Next, in order to combine two branches of execution, we need to use the meet operator to go down in the lattice, to a conservative, less precise result. Thus we must add the second constraint rule. However, this only holds if l has predecessors, when $l \neq l_{init}$. So finally, we must add an initial value to our labels, which give a starting point in order to solve the constraint system.

In order to find the result, we must solve this system. The result is the greatest fixed point of this system (the most precise solution). Can the system be recursive? YES! For programs with while loops, the system is recursive. We can solve the system using an iterative algorithm, which repeatedly inspects each rule in the system and updates the information in *Result* accordingly.

To build an abstract interpretation analysis algorithm, one must define the following: the abstract lattice domain, the transfer functions for each command, and the initial dataflow information. To ensure the termination of the iterative algorithm that solves the constraints, the following conditions must be satisfied:

- The lattice must satisfy the DCC condition
- The transfer functions $[\![c]\!]$ are monotonic for all commands c.

The intuition behind these requirements is that we will only go down in the lattice by monotonicity, and then will terminate due to the DCC.

4 Example: Sign Analysis

In this example, we will take a program and statically compute the signs of each variable in the program, at each point. We build the abstraction on top of the set of possible signs:

$$Sign = \{-, 0, +\}$$

We then define the lattice as a set of elements in Sign, with the partial order as set inclusion:

$$L_{a_1} = (\mathcal{P}(Sign), \supseteq)$$

This models the possible signs for a single variable, at a given point. The set of all variables is given by the lattice domain: $L_a : Var \to L_{a_1}$.

Now we have defined the lattice, we need to define how the program executes in the abstract domain. For this, we must define the transfer function $[\![c]\!] : L_a \to L_a$. We just need to define this function for skip, assignments, and test conditions. The other commands (sequences, if, while) are just control flow constructs and their effect is captured in the *pred* function. The transfer functions are:

$$\begin{array}{rcll}
[\![b]\!]s & = & s & \text{where } s \in L_a \\
[\![x := a]\!]s & = & s[x \to sign(a, s)] & \\
 & & & \\
sign(n, s) & = & sgn(n) & \text{where } n \text{ is a number and } sgn(n) \text{ is its sign} \\
sign(x, s) & = & s(x) & \text{where } x \text{ is a variable and } s(x) \text{ is the set of signs for } x \\
sign(a_0 \oplus a_1, s) & = & sign(a_0, s) \oplus_{as} sign(a_1, s) & \\
 & & & \\
s_1 \oplus_{as} s_2 & = & \bigcup_{\substack{x \in s_1 \\ y \in s_2}} x \oplus_a y &
\end{array}$$

We can define the $\oplus_a$ functions individually in tables:

$+_a$	$-$	0	$+$
$-$	$\{-\}$	$\{-\}$	$\{-, 0, +\}$
0	$\{-\}$	$\{0\}$	$\{+\}$
$+$	$\{-, 0, +\}$	$\{+\}$	$\{+\}$

$\times_a$	$-$	0	$+$
$-$	$\{+\}$	$\{0\}$	$\{-\}$
0	$\{0\}$	$\{0\}$	$\{0\}$
$+$	$\{-\}$	$\{0\}$	$\{+\}$

By using the functions defined above, we can statically compute the signs that each variable may have throughout the program. One thing we haven't seen is why this is correct. Next lecture we will discuss how one can formally prove that this analysis is correct.