Scribe: Krzysztof Ostrowski Lecturer: Andrew Myers

1 Introduction

Throughtout the course we learned about two prominent techniques of reasoning about program behavior and correctness: *dynamic semantics*, the most powerful, but also the most computationally intensive method used only in few specialized applications, and *type systems*, a leightweight method that exists in some form in almost all known languages, but that provides relatively little information about a program.

One notable deficiency of type systems is that they are *flow insensitive*, the typing information represent properties of parts of the program that do not depend on previously executed code. In contrast to that, in *data flow analysis* a program is represented as a graph, its parts as nodes, the flow of execution as edges with cycles corresponding to loops and fixed points. This representation allows for reasoning about things like which variables are never used, never read before being overwritten etc., which allows for code optimization.

Type systems are an example of a feature in support for *static program analysis*, reasoning about properties of programs that hold during all executions. Both dynamic semantics and static analysis are examples of formal methods. In practice, we usually rely on more ad-hoc methods such as *testing*, more feasible in practice than using theorem provers and more insightful than static analysis. Testing, however, has its own deficiencies. One problem is writing test cases: not only is it tedious, but we can rarely cover all the possible situations that may arise. Automatic this process leads to *model checking*, a technique for verification by intelligent state-space exploration, used e.g. for the analysis of distributed protocols for its effectiveness in finding counterexamples. Nevertheless, if the space of program parameteres is large enough, as is usually the case (often called *state explosion*), even this technique is very limited.

One answer to this problem is to go back to dynamic semantics, but instead of reasoning about concrete values or states, to reason about *abstractions*, sets of values or states, which leads to *abstract interpretation*, a technique of reasoning about programs through via formal semantics where states are replaced by sets of states and inference rules talk about sets of executions. In the following section we present an example of abstract interpretation technique in the context of a denotational semantics of IMP.

2 Sign Analysis

Let's imagine that for the purpose of our algorithm we only need to care about the signs of some variables, our abstraction will therefore need to capture the information about possible signs of numbers stored in them. We replace the set of values $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$ with a power set of $A = \{-, 0, +\}$, the set of possible signs. We will no longer be able to say that a given variable has a specific value (say, 63). Instead, we will be able to talk about the possible signs: for example, we can say that a value of a variable is nonnegative, which in our framework is represented as $\{0, +\}$. Consequently, the store, previously $\sigma : \mathbf{Var} \to \mathbb{Z}$, will now be a function into $\mathcal{P}(A)$, i.e. $\sigma : \mathbf{Var} \to \mathcal{P}(A) = \Sigma$.

In order to accomodate the new values, we also need to modify our arithmetic. Arguments and values of functions are now sets of signs. For example, consider a successor function, $f(n) = n + 1$. When applied to zero or a positive number it will always yield a positive number, whereas when applied to a negative number it will always return a negative number or zero. We can summarize this finding as:

$$\begin{array}{lcl} f(\{-\}) & = & \{-, 0\} \\ f(\{0\}) & = & \{+\} \\ f(\{+\}) & = & \{+\} \end{array}$$

Values of f for arbitrary sets of signs can be obtained from the above as union of values over their elements. Using this idea, the following table encodes the semantics of operator $\underline{+}$, a version of $+$ in our new framework.

	-	0	+
-	-	-	A
0	-	0	+
+	A	+	+

Using this table, we can conclude e.g. that $\{+, 0\} \underline{+} \{0\} = \{0, +\}$. Now, to formalize all this, we define the denotation of arithmetic expressions as $\underline{\mathcal{A}}[\![a]\!] : \Sigma \to \mathcal{P}(A)$, and the semantics of our only arithmetic operator as $\underline{\mathcal{A}}[\![a_1 + a_2]\!]\sigma = \underline{\mathcal{A}}[\![a_1]\!]\sigma \ \underline{+} \ \underline{\mathcal{A}}[\![a_2]\!]\sigma$, with $\underline{+}$ defined by the table above. The denotation of statements does not need to be change, it is still $\underline{\mathcal{C}}[\![c]\!]\sigma : \Sigma \to \Sigma$, denotations of simple statements are given below.

$$\begin{array}{lcl} \underline{\mathcal{C}}[\![\mathsf{skip}]\!]\sigma & = & \sigma \\ \underline{\mathcal{C}}[\![x := a]\!]\sigma & = & \sigma[x \mapsto \underline{\mathcal{A}}[\![a]\!]\sigma] \end{array}$$

We can define denotation of boolean expressions analogously. Note that, since we no longer know the exact values of variables, we need to reason about all the possible outcomes of comparisons, therefore apart from true and false, our boolean expressions could have undefined values, which we may conveniently capture by defining the set of boolean values as the powerset of $\{\mathsf{true}, \mathsf{false}\}$.

The fact that boolean values may be undefined slightly complicates the semantics of if..then..else since we need to account for both possibilities.

$$\underline{\mathcal{C}}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\sigma = (\mathsf{if}\ \mathsf{true} \in \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \mathcal{C}[\![c_1]\!]\sigma\ \mathsf{else}\ \lambda x.\emptyset) \sqcup (\mathsf{if}\ \mathsf{false} \in \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \mathcal{C}[\![c_2]\!]\sigma\ \mathsf{else}\ \lambda x.\emptyset)$$

What happens here is taking a supremum over states resulting from processing both branches, with $\lambda x.\emptyset$ representing the bottom of Σ (indeed, it is a $\perp$ of Σ since $\emptyset$ is a $\perp$ for the set of values). Note that in any case $\mathcal{B}[\![b]\!]$ must contain at least one of $\{\mathsf{true}, \mathsf{false}\}$. We can derive the denotation of while in a similar way.

$$\underline{\mathcal{C}}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]\sigma = fix(\lambda f.\mathsf{if}\ \mathsf{true} \in \underline{\mathcal{B}}[\![b]\!]\sigma\ \mathsf{then}\ f(\underline{\mathcal{C}}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma)$$

3 Applications

In practice, the usefullness of abstract interpretation depends on choosing the right abstraction. A too coarse-grained abstraction may fail to precisely identify problems with a program, whereare a too fine-grained asbtraction may be hard to reason about. One idea that has recently gained on popularity is *counterexample-driven refinement*, shown schematically on Figure 3. The idea is that we have a model checker that takes in a program and an abstraction and performs an abstract execution, in a way similar to what we did above, trying to identify problems with the program. If our abstraction turns out to be too weak to identify problems, a feedback loop is used to refine the abstraction. This process typically involves a model checker that, guided by the abstract execution, is trying to find a concrete execution of a program, to determine, for example, which variables of the program should no longer be represented abstractly. This process proceeds iteratively, gradually refining the abstraction as necessary to determine whether there is a problem with the program or not.

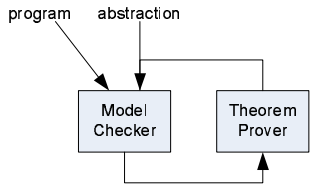

Figure 1: Counterexample-driven refinement.

A project SLAM at Microsoft (`http://research.microsoft.com/slam/`), one of the many systems using this technique, is used to verify correctness of Windows drivers.