1 Introduction to Abstract Interpretation

At this point in the course, we have looked at several aspects of programming languages: *operational semantics*, *denotational semantics*, *axiomatic semantics*, and *static semantics* (type theory). Operational and denotational semantics characterize the dynamic execution of a program; axiomatic and static semantics include techniques 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 verifying correctness with respect to some specification.

The two most prominent static analysis techniques are *type systems* and *abstract interpretation*. The distinguishing features of these two approaches are:

- *Type systems*:
  - In type systems, programmers typically annotate the program with type information, or it may be inferred by the compiler. We can regard these type annotations as global invariants provided by the user explicitly or inferred by the compiler based on the programmer's use of symbols.
  - Types are *flow invariant*. This means that a variable or expression has the same type regardless of where it appears in its scope.
- *Abstract Interpretation*: Abstract interpretation differs from type analysis in that it is not flow invariant. It is related to dataflow analysis, but also provides a framework that allows one to formally prove the correctness of the 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 is a coarse approximation to the concrete information at a program point in all possible executions of the program.

  An example of abstract interpretation is *sign analysis*. Its goal is to statically compute the possible signs of each variable at each program point. Here the concrete information is the actual values of the variables during program execution, and the abstract information gives just the sign of variables.

  In contrast to type systems, abstract interpretation has the following features:
  - 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.
  - There are typically no program annotations (although there could be). The analysis typically infers the abstract information by itself, because it is unreasonable to ask the programmer to provide annotations at every point of every program. In particular, the compiler must often discover loop invariants; it is not easy to extract this information automatically.

The compiler statically determines the abstract information that we are interested in by "executing" the program in an 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 does not. We expect the compiling process, including static analysis, to terminate even if our program has an infinite loop.

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 greater cost.

2 Lattices

We 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 *least upper bound* (lub) or *supremum* $\sqcup X$ and a *greatest lower bound* (glb) or *infimum* $\sqcap X$.

A complete lattice is different from a CPO, which only requires a lub for nonempty chains.

2.1 Notation for Lattices

Regarded as binary operators, the lub and the glb are also referred to as *join* and *meet*, respectively. In this case, we use 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: $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$.

The intuition behind the partial ordering in the abstract lattice is that elements lower in the lattice are more precise. The most precise piece of abstract information is $\bot$ and the least precise is $\top$ (although in some treatments in the literature, it is the other way around).

2.2 Properties of 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.

2.3 Properties of 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 *ascending chain condition* (ACC). This is the opposite of well-foundedness. A lattice satisfies the ACC if there are no infinite ascending chains:

$$\text{If } x_n \sqsubseteq x_{n+1} \text{ for all } n, \text{ then } \exists n_0\ \forall n \geq n_0\ x_n = x_{n_0}.$$

We can also define the *height* of a lattice as the maximum number of distinct elements in a chain. A finite height implies the ACC.

3 Formal Framework

We are going to study abstract interpretation using the imperative language IMP. We will extend its syntax with labels for each atomic instruction and test:

$$c \quad ::= \quad [\mathsf{skip}]^\ell \quad | \quad [x := a]^\ell \quad | \quad c_0; c_1 \quad | \quad \mathsf{if}\ [b]^\ell\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \quad | \quad \mathsf{while}\ [b]^\ell\ \mathsf{do}\ c_0$$

where the $\ell \in \mathit{Labels}$ are labels. Different instructions are labeled with distinct labels. We denote by ℓ_{init} the label of the first atomic instruction of the program.

Let L_a be a lattice of abstract values (the "a" stands for "abstract"). The result of abstract interpretation is a function *Result* which assigns two elements of L_a to each atomic instruction, before and after the instruction:

$$\mathit{Result} : \mathit{Labels} \to L_a \times L_a$$

We denote by $\mathit{Result}(\bullet\,\ell)$ the result right before the instruction labeled by ℓ; and by $\mathit{Result}(\ell\,\bullet)$ the result right after ℓ.

We now want to determine how the abstract information changes when a command is executed. This is done by "executing" the command in the abstract domain. For this, we introduce a *transfer function* for each command c to map the abstract value prior to executing the command to the abstract value just after:

$$[\![c]\!] : L_a \to L_a.$$

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*.

$$\begin{aligned}
\mathit{Result}(\ell\,\bullet) &= [\![c]\!](\mathit{Result}(\bullet\,\ell)) && (1)\\
\mathit{Result}(\bullet\,l) &= \sqcup\{\mathit{Result}(\ell'\bullet) \mid \ell' \in \mathit{pred}(\ell)\},\ \ell \neq \ell_{\text{init}} && (2)\\
\mathit{Result}(\bullet\,\ell_{\text{init}}) &= i_0, && (3)
\end{aligned}$$

where $\mathit{pred}(\ell)$ is the set of 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. The datum i_0 is the boundary condition—the abstract information at the entry point in the program, representing what we know about the input values.

Informally, (1) says how the abstract value just after execution of an atomic command depends on the abstract value just before. The equation (2) says how to combine two or more branches of execution coming into an atomic instruction. Here the join operator is used to go up in the lattice, to a conservative, less precise result representing the best knowledge that we have at that point, given that execution might have flowed to that point from any one of the predecessors. Finally, (3) says where to start, since ℓ_{init} has no predecessors. This gives us a starting point from which to solve the constraint system.

In order to find the result, we must solve this system (1)–(3); that is, we must find a solution *Result* that satisfies the equations. Provided the soundness criteria of Section 5 below are satisfied, any solution will be a sound approximation, but the least solution will be the most precise. Note that the system be recursive if the program contains a while loop.

We can solve the system using an iterative algorithm called a *worklist algorithm*, which repeatedly inspects each rule in the system and updates *Result* accordingly. We start with R that labels all program points $\bot$ except for $R(\bullet\ell_{\text{init}}) = i_0$. We put ℓ_{init} on a worklist, which can be a queue or a stack. Then we repeatedly remove the next ℓ from the worklist, apply the associated transfer function to the current $R(\bullet\ell)$ to get a new $R(\ell\bullet)$, then for every successor ℓ', update $R(\bullet\ell')$ by taking the join of $R(\ell\bullet)$ with the current value of $R(\bullet\ell')$. For any value that changes (it can only go up in the lattice order if $[\![c]\!]$ is monotone), we put ℓ' on the worklist. We continue in this fashion until there are no more changes, which must happen after a finite time if the lattice satisfies the ACC.

A variant of Kleene algebra can also be used: a matrix can be formed whose rows and columns are indexed by program points whose entries are the transfer functions, and the star of the matrix can be taken to compute the least solution.

To build an abstract interpretation algorithm, one must define the following: the abstract lattice domain L_a, the transfer functions $[\![c]\!]$ for each atomic command c, and the initial dataflow information i_0. To ensure the termination of the worklist algorithm that solves the constraints, the following conditions must be satisfied:

- The lattice must satisfy the ACC
- The transfer functions $[\![c]\!]$ must be monotone for all atomic commands c.

The intuition behind these requirements is that we will only go up in the lattice by monotonicity, therefore the algorithm will terminate due to the ACC.

4 Example: Sign Analysis

In this example, we will statically compute the possible signs of each variable at each point in a given program. The set of possible signs is:

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

The set 2^{Sign} of subsets of *Sign* is partially ordered by set inclusion $\subseteq$. This models the possible signs for each variable at a given point in the program. Our lattice of abstract values is the set of functions

$$L_a \;=\; \mathit{Var} \to (2^{\mathit{Sign}}, \subseteq)$$

giving a set of possible signs for each variable, ordered pointwise.

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 functions $[\![c]\!] : L_a \to L_a$. 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:

$$
\begin{aligned}
[\![b]\!]s &\triangleq s, \text{ where } s \in L_a \\
[\![x := a]\!]s &\triangleq s[\mathit{sign}(a, s)/x] \\
\\
\mathit{sign}(n, s) &\triangleq \begin{cases} \{+\}, & \text{if } n > 0, \\ \{0\}, & \text{if } n = 0, \\ \{-\}, & \text{if } n < 0 \end{cases} \\
\mathit{sign}(x, s) &\triangleq s(x), \text{ where } x \text{ is a variable} \\
\mathit{sign}(a_1 \oplus a_2, s) &\triangleq \mathit{sign}(a_1, s) \oplus_a \mathit{sign}(a_2, s) \\
s_1 \oplus_a s_2 &\triangleq \bigcup_{\substack{x \in s_1 \\ y \in s_2}} x \oplus_a y
\end{aligned}
$$

and the functions $\oplus_a$ are defined individually in tables; for example,

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

Intuitively, the table entry for $+$ and $-$ is $\{-, 0, +\}$ because the sum of a positive and a negative number could be either positive, negative, or 0, and that is the best knowledge we have.

By using the functions defined above, we can statically compute our best knowledge of the signs that each variable could have at each point in the program.

5 Soundness

5.1 Concrete vs. Abstract Domains

Let L_c be the lattice of concrete values corresponding to the abstract values L_a. For the sign analysis example, we could take

$$L_c \;=\; \mathit{Var} \to (2^{\mathbb{Z}}, \subseteq).$$

We can define an abstraction function $\alpha : L_c \to L_a$, which takes concrete values and returns their signs: for $s_c : \mathit{Var} \to 2^{\mathbb{Z}}$,

$$
\begin{aligned}
\alpha(s_c) &\triangleq \lambda x \in \mathit{Var}.\, \{\mathit{sign}(n) \mid n \in s_c(x)\} \\
\mathit{sign}(n) &\triangleq \begin{cases} +, & \text{if } n > 0, \\ 0, & \text{if } n = 0, \\ -, & \text{if } n < 0. \end{cases}
\end{aligned}
$$

We can also define a concretization function $\gamma : L_a \to L_c$, which returns the possible concrete values given our abstract (sign) value. For $s_a : \mathit{Var} \to 2^{\mathit{Sign}}$,

$$\gamma(s_a) \;\triangleq\; \lambda x \in \mathit{Var}.\, \{n \in \mathbb{Z} \mid \mathit{sign}(n) \in s_a(x)\}.$$

These two functions have the following properties:

- α, γ are both monotone,
- $\forall x \in L_c \;\; x \sqsubseteq \gamma(\alpha(x))$,
- $\forall y \in L_a \;\; y \sqsupseteq \alpha(\gamma(y))$.

5.2 Soundness Condition

For soundness, we want to verify that the abstract information correctly reflects the concrete information at each point in the program. For example, in the sign analysis example, we wish to verify that the set of possible signs of the values of a variable at any point of the program at any time during execution when control is at that point is a subset of the set of signs given by the abstract interpretation at that point.

To do this, first we define a function $\beta(\sigma) = \alpha(\lambda x \in \mathit{Var}.\, \{\sigma(x)\})$. For soundness, we must show:

$$\forall c, \sigma, i_a \;\; \beta(\sigma) \sqsubseteq i_a \;\Rightarrow\; \beta(\mathcal{C}[\![c]\!]\sigma) \sqsubseteq [\![c]\!] i_a.$$

Alternatively, we could also show:

$$\beta(\mathcal{C}[\![c]\!]\sigma) \;\sqsubseteq\; [\![c]\!](\beta(\sigma)).$$