Automatic Software Verification Spring Semester, 2015

Lecture 8: May 5, 2015

Lecturer: Prof. Mooly Sagiv *Scribe: Oleg Zlydenko and Ofir Geri*

1 Introduction

This lecture provides theoretical background required for static program analysis, and presents the idea of chaotic iterations. Abstract interpretation was introduced in the seminal paper by Cousot and Cousot [1]. The main idea is to map each of the program states to an abstract state. An abstract state may represent more than one concrete state. Using the abstract states, it will be easier to analyze how running the program changes the program state. If the resulting abstract state corresponds only to concrete states that are valid (i.e., satisfy the required properties), we can be sure that the program is valid and has no bugs. However, since the abstract states only approximate the running of the program, it is possible that our analysis will report bugs that do not exist. We say that an analysis that cannot miss any bugs is *sound*, but if the analysis can report non-existent bugs, we say that it is *not complete*.

2 Mathematical Background

We start with the definition of *partially ordered sets* (posets). We will use posets to describe the abstract states of the program.

Definition 2.1. A partial order $\sqsubseteq$ over a set L is a binary relation, $\sqsubseteq: L \times L \to \{true, false\}$, such that:

1. $\sqsubseteq$ is reflexive: For every $l \in L$, $l \sqsubseteq l$.
2. $\sqsubseteq$ is transitive: For every $l_1, l_2, l_3 \in L$, if $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$, then $l_1 \sqsubseteq l_3$.
3. $\sqsubseteq$ is anti-symmetric: For every $l_1, l_2 \in L$, if $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_1$, then $l_1 = l_2$.

Definition 2.2. A *partially ordered set* (poset) is a tuple $(L, \sqsubseteq)$ where $\sqsubseteq$ is a partial order over L.

Intuitively, a member l of a poset represents the abstract state of the program. Each $l \in L$ may represent more than one concrete state of the program, and given two $l_1, l_2 \in L$, we say that $l_1 \sqsubseteq l_2$ if l_1 is more precise than l_2 (that is, l_1 represents a smaller set of concrete states).

We present a few examples of posets:

- The set of natural numbers N with the $\leq$ operator .
- The powerset $P(S)$ of a set S with the $\subseteq$ or $\supseteq$ operator.
- Odd/even: Suppose we are only in interested in determining whether an integer variable is odd or even. We can use two elements $\{odd, even\}$ to describe the state of a variable. We add two more elements: $\top$ that represents any value of the variable (if the state of the variable is $\top$, it can be either odd or even), and $\bot$, which means that the variable cannot take any value (for example, if it is the result of division by zero). The resulting poset is $\{odd, even, \top, \bot\}$ with $\bot \sqsubseteq odd \sqsubseteq \top$ and $\bot \sqsubseteq even \sqsubseteq \top$.
- Constant propagation: Finding which variables have constant values after running a program, and what these values are. We will discuss this example further throughout the lecture.

We introduce some additional notation. For any $l_1, l_2 \in L$, we say that $l_1 \sqsupseteq l_2$ if and only if $l_2 \sqsubseteq l_1$. Also, we say that $l_1 \sqsubset l_2$ if $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. Similarly, $l_1 \sqsupset l_2$ if and only if $l_2 \sqsubset l_1$.

2.1 Upper and Lower Bounds

Considering the fact that the elements of a poset represent abstract states, and we use the relation $\sqsubseteq$ to denote which abstract states are more precise, it makes sense to define the lower and upper bounds of a set of abstract states. Since each abstract state represents a set of concrete states, intuitively the upper bound of a set of abstract states would be a state that contains all the possible concrete states for that set of abstract states. Conversely, the lower bound would include only concrete states that belong to each of the abstract states in the set. We continue with formal definitions.

Definition 2.3. A subset $L' \subseteq L$ has a lower bound $l \in L$ if for every $l' \in L'$, $l \sqsubseteq l'$.

Definition 2.4. A subset $L' \subseteq L$ has an upper bound $u \in L$ if for every $l' \in L'$, $l' \sqsubseteq u$.

A subset $L' \subseteq L$ of states may have more than one lower or upper bound. For example, for the two sets $\{2,3\}, \{2,3,4\} \subseteq \{1,2,3,4\}$, both $\{2,3,4\}$ and $\{1,2,3,4\}$ serve as upper bounds, and both $\{2\}$ and $\{2,3\}$ serve as lower bounds. This motivates the following definitions.

Definition 2.5. The greatest lower bound of a subset $L' \subseteq L$ is a lower bound l_0 such that $l \sqsubseteq l_0$ for every lower bound l of L'. We denote the greatest lower bound of L' by $\sqcap L'$ (the operator $\sqcap$ is called "meet").

Definition 2.6. The lowest upper bound of a subset $L' \subseteq L$ is an upper bound u_0 such that $u_0 \sqsubseteq u$ for every upper bound u of L'. We denote the lowest upper bound of L' by $\sqcup L'$ (the operator $\sqcup$ is called "join").

Note that if the greatest lower bound exists, it is unique (and the same holds for the lowest upper bound). Sometimes we abuse notation and write $a \sqcap b$ and $a \sqcup b$ instead of $\sqcap\{a,b\}$ and $\sqcup\{a,b\}$, respectively.

2.2 Complete Lattices

In order to describe abstract states of a program, we wish to use posets that are complete lattices.

Definition 2.7. A poset $(L, \sqsubseteq)$ is a *complete lattice* if every subset $L' \subseteq L$ has a greatest lower bound and a lowest upper bound (with respect to $\sqsubseteq$). We denote a complete lattice $(L, \sqsubseteq)$ by the tuple $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, where $\bot = \sqcup\emptyset = \sqcap L$ and $\top = \sqcup L = \sqcap\emptyset$.

Many of the examples that we have shown for posets are actually complete lattices: The set of natural numbers N with the less than or equal to operator, the powerset $P(S)$ with the $\subseteq$ and $\supseteq$ operators, and constant propagation.

Lemma 2.8. *For every poset $(L, \sqsubseteq)$, the following conditions are equivalent:*

- *L is a complete lattice.*
- *Every subset of L has a least upper bound.*
- *Every subset of L has a greatest lower bound.*

Proof. From the definition it follows that if L is a complete lattice, every subset $L' \subseteq L$ has a least upper bound and greatest lower bound. Now, we assume that every $X \sqsubseteq L$ has a least upper bound $\sqcup X$, and we will show how to construct a greatest lower bound $\sqcap X$. The second direction can be shown in a similar fashion, thus completing the proof.

Choose some $X \subseteq L$ and consider the set S of the lower bounds of X, $S = \{z \in L | \forall x \in X, z \sqsubseteq x\}$. We can now use the least upper bound operator $b = \sqcup S$. We will show that b is a greatest lower bound of X.

First, notice that every item in X is an upper bound of S (by definition of S). Since b is the least upper bound of X, it means that for every $x \in X$ we have $b \sqsubseteq x$, thus b is a lower bound of X.

And since b is bigger than any item in S, which is the set of lower bounds of X, it means that b is the greatest lower bound of X. □

2.2.1 The Cartesian Product of Complete Lattices

Given two complete lattices L_1 and L_2, we can define a new relation over the set $L_1 \times L_2$: $(x_1, x_2) \sqsubseteq (y_1, y_2)$ if and only if $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$.

Claim 2.9. *Given two complete lattices, L_1 and L_2, the cartesian product $(L_1 \times L_2, \sqsubseteq)$ is also a complete lattice.*

Example 1. Let $(L_1, \sqsubseteq_1)$ be the complete lattice of natural numbers ($L_1 = \mathbb{N}, \sqsubseteq_1 = \leq$), and $(L_2, \sqsubseteq_2)$ be the complete lattice induced by a power set ($L_2 = P(\{0,1,2\}), \sqsubseteq_2 = \subseteq$). We define $L = L_1 \times L_2$, and $(x_1, x_2) \sqsubseteq (y_1, y_2)$ if and only if $x_1 \leq y_1$ and $x_2 \subseteq y_2$. We show that this is a complete lattice.

First, we show that $\sqsubseteq$ is a partial order.

- *Reflexive*: Let $(x_1, x_2) \in L$. Since $x_1 \leq x_1$ and $x_2 \subseteq x_2$, $(x_1, x_2) \sqsubseteq (x_1, x_2)$.
- *Transitive*: Let $(x_1, y_1), (x_2, y_2), (x_3, y_3) \in L$, be such that $(x_1, y_1) \sqsubseteq (x_2, y_2) \sqsubseteq (x_3, y_3)$. This means that $x_1 \leq x_2 \leq x_3$, hence $x_1 \leq x_3$. Similarly, $y_1 \subseteq y_2 \subseteq y_3$, hence $y_1 \subseteq y_3$. Therefore, $(x_1, y_1) \sqsubseteq (x_3, y_3)$.
- *Anti-symmetric*: Let $(x_1, y_1), (x_2, y_2) \in L$ be such that $(x_1, y_1) \sqsubseteq (x_2, y_2) \sqsubseteq (x_1, y_1)$. This means that $x_1 \leq x_2 \leq x_1$, hence $x_1 = x_2$. Similarly, $y_1 \subseteq y_2 \subseteq y_1$, hence $y_1 = y_2$. Therefore, $(x_1, y_1) = (x_2, y_2)$.

Now we show that $(L, \sqsubseteq)$ is a complete lattice. Following Lemma 2.8 it is enough to show that every set $X \subseteq L$ has a least upper bound.

Denote $X_1 = \{l_1 \in L_1 | \exists l_2 \in L_2 \; s.t. \; (l_1, l_2) \in X\}$ and $X_2 = \{l_2 \in L_2 | \exists l_1 \in L_1 \; s.t. \; (l_1, l_2) \in X\}$. Both of them have least upper bounds in their respective lattices $x_1 \in L_1$ and $x_2 \in L_2$. We will show that $x = (x_1, x_2) \in L$ is the least upper bound of X. First, it is an upper bound, since for every $(y_1, y_2) \in X$ we have $y_1 \leq x_1$ and $y_2 \subseteq x_2$. Second, x is minimal, because if we have another upper bound for X, $z = (z_1, z_2) \in L$, then z_1 is an upper bound of X_1, and from minimality of x_1 we have $x_1 \leq z_1$. Similarly, we get $x_2 \subseteq z_2$, showing that $x \sqsubseteq z$.

Note that in the above proofs we do not use any particular properties of the lattices, but just the fact that L_1, L_2 are complete lattices.

Another example of operations on complete lattices is finite maps. Given a complete lattice $(L_1, \sqsubseteq_1)$ and a finite set V, we define the poset $L = (V \to L_1, \sqsubseteq)$, where $e_1 \sqsubseteq e_2$ if for every $v \in V$, $e_1(v) \sqsubseteq e_2(v)$. We claim that L is also a complete lattice.

2.3 Chains

We now consider a sequence of elements, which is formalized by the notion of *chains*.

Definition 2.10. A subset $Y \subseteq L$ in a poset is a *chain* if every two elements in Y are ordered, i.e., for every $l_1, l_2 \in Y$, either $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$.

Definition 2.11. A chain $l_1, l_2, l_3, \ldots$ is:

1. Increasing if $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \ldots$
2. Strictly increasing if $l_1 \sqsubset l_2 \sqsubset l_3 \sqsubset \ldots$
3. Decreasing if $l_1 \sqsupseteq l_2 \sqsupseteq l_3 \sqsupseteq \ldots$
4. Strictly decreasing if $l_1 \sqsupset l_2 \sqsupset l_3 \sqsupset \ldots$

In future examples, we will consider the special case of finite chains.

Definition 2.12. A poset L has a finite height if every chain $Y \subseteq L$ is finite.

Lemma 2.13. *A poset $(L, \sqsubseteq)$ has a finite height if and only if every strictly decreasing or strictly increasing chain is finite.*

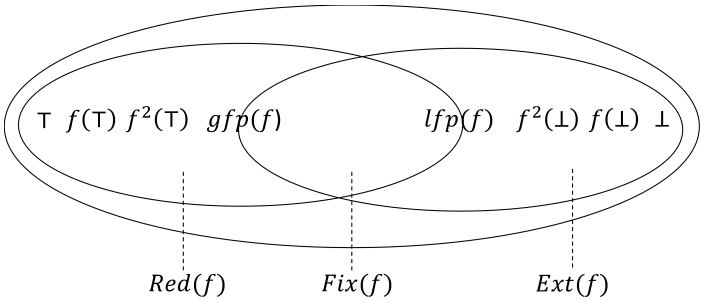

Figure 1: The relation between $Fix(f)$, $Red(f)$, and $Ext(f)$

Proof. First, assume that a poset $(L, \sqsubseteq)$ has a finite height. It follows from the definition that every chain (including strictly decreasing and strictly increasing ones) is finite.

Now, notice that every finite strictly increasing chain is the reverse of a finite strictly decreasing chain (and vice versa). Therefore it is sufficient to show that if we have an infinite chain $S \subseteq L$, we can construct either an infinite strictly increasing chain or an infinite strictly decreasing chain, which we denote by T.

Order the items of S in some order $l_1, l_2, ...$ If there is some index n_0 such that $l_{n_0} \sqsubset l_j$ for every $j > n_0$, put l_{n_0} in our new chain T, and continue the search with $S' = \{l_{n_0+1}, l_{n_0+2}, ...\}$. Since S is infinite, if this process never terminates, T will be an infinite strictly increasing chain.

If, however, the process terminates, it means that we have an infinite chain $l'_1, l'_2, ...$ and there is no index n'_0 such that $l'_{n'_0} \sqsubset l'_j$ for every $j > n'_0$. So we can construct an infinite strictly decreasing chain in the following fashion. Choose some index m_1. According to our assumption, there has to be some index $m_2 > m_1$ such that $l'_{m_2} \sqsubset l'_{m_1}$. Again, using the same argument, there has to be some index $m_3 > m_2$ such that $l'_{m_2} \sqsubset l'_{m_3}$. By repeating this argument an infinite number of times, we get an infinite strictly decreasing chain $l'_{m_1} \sqsupset l'_{m_2} \sqsupset ....$ □

2.4 Monotone Functions and Fixed Points

In our analysis, we will define program instructions as functions that affect the abstract states. Using monotone functions will provide us a way to compute an abstract state that upper bounds all the possible program states.

Definition 2.14. Let $(L, \sqsubseteq)$ be a poset. A function $f : L \to L$ is monotone if for every $l_1, l_2 \in L$ such that $l_1 \sqsubseteq l_2$, $f(l_1) \sqsubseteq f(l_2)$.

We provide an example of a monotone function. Consider the poset $(P(\{1, 2, 3\}), \subseteq)$. The function $f(X) = X \cup \{1\}$ is monotone: If $X \subseteq Y$, then $f(X) = X \cup \{1\} \subseteq Y \cup \{1\} = f(Y)$. However, the complement function is not monotone: If $A \subset B$, then $\bar{A} \supset \bar{B}$.

Definition 2.15. Let $f : L \to L$ be a monotone function, where $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is complete lattice. We define the following sets:

1. $Fix(f) = \{l \in L | f(l) = l\}$ (the set of the fixed points of f)
2. $Red(f) = \{l \in L | f(l) \sqsubseteq l\}$
3. $Ext(f) = \{l \in L | l \sqsubseteq f(l)\}$

The definitions are illustrated by Figure 1. Suppose $l \in Red(f)$ and we apply f on l and $f(l)$. From monotonicity, we get that $f(f(l)) \sqsubseteq f(l)$, hence $f(l) \in Red(f)$ as well. We continue this way until we reach a fixed point. The same applies for $Ext(f)$. We are interested in finding the least and greatest fixed points of f, which we denote by $lfp(f)$ and $gfp(f)$, respectively.

Theorem 2.16. *(Tarski 1955) If f is monotone, then $lfp(f) = \sqcap Fix(f) = \sqcap Red(f) \in Fix(f)$ and $gfp(f) = \sqcup Fix(f) = \sqcup Ext(f) \in Fix(f)$.*

Proof. We show that $lfp(f) = \sqcap Fix(f) = \sqcap Red(f) \in Fix(f)$. The proof of the second part is similar. First we show that $\sqcap Red(f) \in Fix(f)$. Denote $x_0 = \sqcap Red(f)$. By definition, $x_0 \sqsubseteq x$ for every $x \in Red(f)$. Since f is monotone, we get that $f(x_0) \sqsubseteq f(x) \sqsubseteq x$ for every $x \in Red(f)$. Therefore, $f(x_0)$ is a lower bound of $Red(f)$, and we get $f(x_0) \sqsubseteq x_0$ since x_0 is the greatest lower bound. This implies that $x_0 \in Red(f)$, and after applying f on both sides of $f(x_0) \sqsubseteq x_0$, we get that $f(x_0) \in Red(f)$ as well. Since x_0 is a lower bound, we get that $x_0 \sqsubseteq f(x_0)$, or $f(x_0) = x_0 \in Fix(f)$.

We now have to show that $x_0 = \sqcap Fix(f)$. Since $x_0 \in Fix(f)$, we have $\sqcap Fix(f) \sqsubseteq x_0$. Recall that $Fix(f) \subseteq Red(f)$, thus $x_0 = \sqcap Red(f) \sqsubseteq Fix(f)$. Finally, we get that $x_0 = \sqcap Fix(f)$, as promised. □

For the case of posets with finite height, we can use this theorem to compute the least fixed point, since there are no infinite ascending chains. The algorithm is presented as Algorithm 1.

Algorithm 1 LFP computation according to Tarski's theorem

```
x := ⊥
while f(x) ≠ x do
  x := f(x)
end while
```

3 Chaotic Iterations

Given a lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ with finite strictly increasing chains (of maximum length m), and a monotone function $\underline{f} : L^n \to L^n$, we would like to compute the least fixed point $lfp(\underline{f})$. This can be easily done with Algorithm 2. Essentially, we are executing Algorithm 1 on every coordinate of $\underline{x} \in L^n$, and as before, the algorithm is guaranteed to terminate.

Algorithm 2 Naive LFP Computation

```
x := (⊥, ⊥, ..., ⊥)
while f(x) ≠ x do
  x := f(x)
end while
```

Algorithm 3, named *Chaotic Iterations*, computes $lfp(\underline{f})$ in (potentially) less time than the naive approach of Algorithm 2. The initialization is the same as before: a vector of $\bot$'s. In this algorithm, during each iteration we compute f_i for only one index i according to the *worklist* (WL). At the end of the iteration, we update the WL with the indices of the vector that are affected by x_i.

For example, if each index represents a node in the Control Flow Graph (CFG) of a program, and we updated the value of one node v, we need to update the value of the successor of v. However, if the value of v does not change during the iteration, there is no need to update the successor again (unless it is already in the WL).

Algorithm 3 Generic Chaotic Iterations

```
x = {⊥, ⊥, ..., ⊥}
WL = {1, 2, ..., n}
while WL ≠ ∅ do
  select and remove i ∈ WL
  new := f_i(x)
  if new ≠ x[i] then
    x[i] := new
    add all the indices that directly depend on i to WL
  end if
end while
```

Remark. Note that the order of the items in the WL strong affects the runtime of the algorithm. Some orders can be (heuristically or empirically) better.

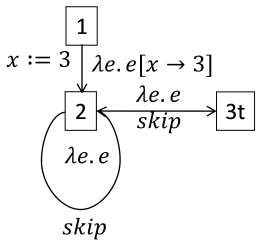

Figure 2: The program studied in Example 2

3.1 Specialized Chaotic Iterations

We wish to apply chaotic iterations to static analysis of programs. Consider a lattice L and the CFG of our program $CFG = (V, E, s)$ (s is the program entry point). L represents some information we want to find about the program (for example — Constant Propagation — which variables are constant and what values they have).

We will do this by computing the values $df_{entry}[v] \in L$ for every program location v in V, which hold a (conservative) value of L that holds on the entry to that location. Assume that for every instruction e in E we can define a monotone function $f(e) : L \to L$, which represents how the state changes on the execution of the instruction. Then, the values of df_{entry} are a solution of the following system of equations:

$$S = \begin{cases} df_{entry}[s] = \iota \\ df_{entry}[v] = \bigsqcup\{f(u,v)(df_{entry}[u]) | (u,v) \in E\} \end{cases}$$

The character ι represents the abstraction of the initial state. For every program location we go only over the possible predecessors, and use the $\sqcup$ operator, in order to find an overapproximation of the value.

Note that choosing $\forall v \in V : df_{entry}[v] = \top$ always solves S. Instead, we look for $lfp(S)$. If the trivial solution is indeed the lfp of S, it means that our abstraction was too imprecise.

We can also rewrite S and use vectors to represent the states. Denote $F_S : L^n \to L^n$. The equivalent system ($lfp(S) = lfp(F_S)$) is:

$$\begin{cases} F_S(\underline{x})[s] = \iota \\ F_S(\underline{x})[v] = \bigsqcup\{f(u,v)(\underline{x}[u]) | (u,v) \in E\} \end{cases}$$

Example 2. Consider the program depicted in Figure 2. Assume that the initial value of x is 0. We get the following system of equations:

$$df[1] = [x \mapsto 0] df[2] = df[1][x \mapsto 3] \sqcup DF(2) df[3] = df[2]$$

The most precise solution that we can get is:

$$df[1] = [x \mapsto 0] df[2] = [x \mapsto 3] df[3] = [x \mapsto 3]$$

However, the following is also a possible solution to the equation set:

$$df[1] = [x \mapsto \top] df[2] = [x \mapsto 3] df[3] = [x \mapsto 3]$$

The algorithm that solves S (or, similarly, F_S), is easily derived from Algorithm 3, and is presented as Algorithm 4.

We show an example for constant propagation using chaotic iterations.

Example 3. Consider the program presented in Figure 3. The algorithm runs as follows:

1. $[x \mapsto ?, y \mapsto ?, z \mapsto ?]$, with $WL = \{2, 3, 4, 5, 6, 7\}$
2. $[x \mapsto ?, y \mapsto ?, z \mapsto 3]$, with $WL = \{3, 4, 5, 6, 7\}$

Algorithm 4 Specialized Chaotic Iterations

Require: Graph $G(V, E)$, node s, lattice L, $L\ \iota$, $f : E \to (L \to L)$
Ensure:
  for each v in V **do**
    $df_{entry}[v] := \bot$
  end for
  $df_{entry}[s] := \iota$
  $WL := \{s\}$
  while $WL \neq \emptyset$ **do**
    select and remove $u \in WL$
    for each v such that $(u, v) \in E$ **do**
      $temp := f(e)(df_{entry}[u])$
      $new := df_{entry}[v] \sqcup temp$
      if $new \neq df_{entry}[v]$ **then**
        $df_{entry}[v] := new$
        $WL := WL \cup \{v\}$
      end if
    end for
  end while

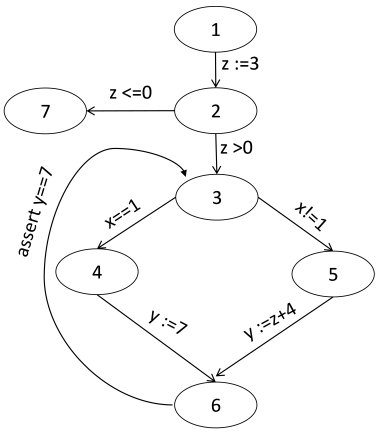

Figure 3: The program studied in Example 3

3. $[x \mapsto ?, y \mapsto ?, z \mapsto 3]$, with $WL = \{4, 5, 6, 7\}$
4. $[x \mapsto 1, y \mapsto 7, z \mapsto 3]$, with $WL = \{5, 6, 7\}$
5. $[x \mapsto ?, y \mapsto 7, z \mapsto 3]$, with $WL = \{6, 7\}$
6. $[x \mapsto ?, y \mapsto 7, z \mapsto 3]$, with $WL = \{7\}$

3.1.1 Complexity of Chaotic Iterations

Let n be the number of CFG nodes, k the maximum out degree of any node, h the height of the lattice L, and c the maximum cost of (a) applying f, (b) calculating $\bigsqcup$, (c) comparison of elements in L. Each of the n nodes can be updated at most h times. Each update adds at most k new nodes to the worklist, so in total there are at most $n \cdot h \cdot k$ visits to any node. Each visit (that is not necessarily an update) takes c time. Thus, the runtime complexity of the algorithm is $O(nkhc)$.

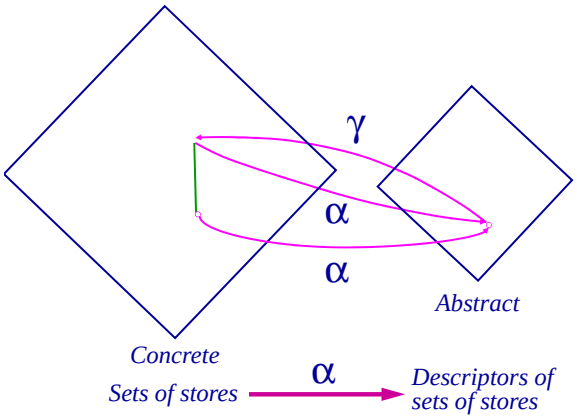

Figure 4: Galois Connection

3.1.2 Soundness and Completeness

Since we are updating conservatively (using monotone function and the operator $\sqcup$), the computed least fixed point represents all possibly occurring runtime states. This means that the described algorithm is *sound*. However, for the same reasons, the algorithm is not *complete*. It is possible that the resulting abstract state will represent impossible concrete states.

4 Abstract Interpretation

Abstract Interpretation was introduced by Cousot and Cousot [1] and this technique lies in the foundations of static program analysis. It provides a mathematical framework that allows us to prove that a static analysis (e.g., constant propagation) is locally sound, and sometimes helps us understand where precision is lost. The technique is not limited to a chosen abstraction or programming style.

4.1 Galois Connections

We begin by defining two lattices C and A, that represent the *concrete* and *abstract* states of the program respectively, and two functions $\alpha : C \to A$ and $\gamma : A \to C$, which are the *abstraction* and *concretization* functions, respectively. The abstraction function maps concrete states (or sets of states) to their abstract representation. The concretization function maps an abstract representation to the full set of concrete states that it represents — as depicted in Figure 4.

Definition 4.1. The pair of functions (α, γ) forms a *Galois connection* if:

- α and γ are monotone
- $\forall a \in A : \alpha(\gamma(a)) \sqsubseteq a$
- $\forall c \in C : c \sqsubseteq \gamma(\alpha(c))$

Alternately, the functions form a Galois connection if $\forall c \in C, \forall a \in A : \alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma(a)$.

Lemma 4.2. *The functions α and γ that form a Galois connection uniquely determine each other. That is, if (α, γ_1) and (α, γ_2) are both Galois Connections, then $\gamma_1 = \gamma_2$ (and analogously when the roles of α and γ are reversed).*

Proof. Suppose that (α, γ_1) and (α, γ_2) are both Galois connections. From the second definition we get, $\forall c \in C, \forall a \in A : \alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma_1(a) \iff c \sqsubseteq \gamma_2(a)$. For every $a \in A$, $\gamma_1(a) \sqsubseteq \gamma_1(a)$. By the definition of Galois connection, we get that for every $a \in A$, $\gamma_1(a) \sqsubseteq \gamma_2(a)$. Symmetrically, we can show that $\gamma_2(a) \sqsubseteq \gamma_1(a)$. Combining both inequalities we get $\gamma_1(a) = \gamma_2(a)$, as promised. □

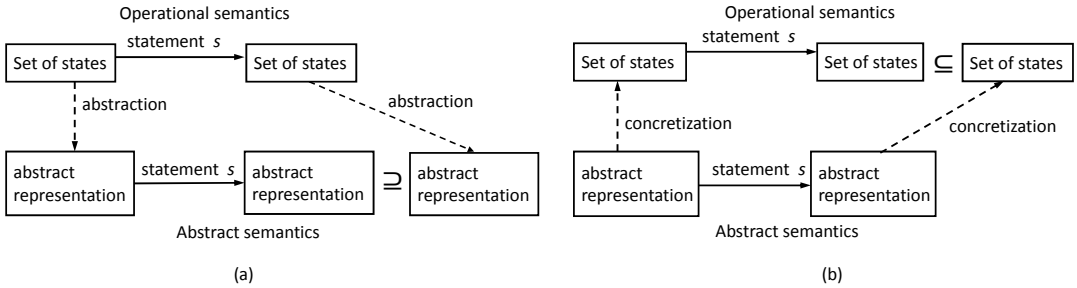

Figure 5: Sound Abstract Interpretation

Example 4. We return to the example of constant propagation. Our concrete states are $C = [Var_* \to \mathbb{Z}]$ (that is, mappings between the program variables to integers). Our abstract states are $A = [Var_* \to \mathbb{Z} \cup \{\bot, \top\}]$ (the variables can also be $\bot$ for "no integer" or $\top$ for "any integer").

The abstraction of an individual state is a function $\beta_{CP} : C \to A$, $\beta_{CP}(\sigma) = \sigma$. The abstraction of a set of states is a function $\alpha_{CP} : P(C) \to A$, $\alpha_{CP}(CS) = \sqcup\{\beta_{CP}(\sigma) | \sigma \in CS\}$. The concretization of an abstract state is a function $\gamma_{CP} : A \to P(C)$, $\gamma_{CP}(df) = \{\sigma | \beta_{CP}(\sigma) \sqsubseteq df\}$. So, if an abstract variable is mapped to $\top$, its concretization is $\mathbb{Z}$, and the concretization of $\bot$ is $\emptyset$.

These functions provide a sound but not complete representation of variables with constant values, in the sense that

$$\alpha_{CP}(Reach(v)) \sqsubseteq df(v)$$

and

$$Reach(v) \sqsubseteq \gamma_{CP}(df(v)),$$

but these are not necessarily equalities ($Reach(v)$ represents all the possible variable values that are accessible from state v). For example, consider $\alpha_{CP}(\{[x \to 5, y \to 7], [x \to 6, y \to 7]\}) = [x \to \top, y \to 7]$, and $\gamma_{CP}([x \to \top, y \to 7] = \{[x \to 1, y \to 7], [x \to 2, y \to 7], ...\}$.

Since we know exactly how α_{CP} and γ_{CP} behave, we can easily verify that:

- α_{CP} is monotone.
- γ_{CP} is monotone.
- $\forall df \in A : \alpha_{CP}(\gamma_{CP}(df)) \sqsubseteq df$.
- $\forall c \in P(C) : c \sqsubseteq \gamma_{CP}(\alpha_{CP}(c))$.

Therefore, the functions $(\alpha_{CP}, \gamma_{CP})$ form a Galois connection.

Figure 5 illustrates how Galois connections are used for sound abstract interpretation.

4.1.1 Upper Closures

Definition 4.3. An *upper closure* is a function $\uparrow$: $P(\Sigma) \to P(\Sigma)$ such that:

- $\uparrow$ is monotone, i.e. for every $X \subseteq Y$, $\uparrow X \subseteq \uparrow Y$.
- $\uparrow$ is extensive, i.e. $\uparrow X \supseteq X$.
- $\uparrow$ is a closure, i.e. $\uparrow (\uparrow X) = \uparrow X$.

Every Galois connection defines an upper closure $\uparrow$: $P(C) \to P(C)$, with: $\uparrow a = \gamma(\alpha(a))$. This function is monotonic, since it is the composition of two monotonic functions. It is extensive, by the definition of Galois connections ($\forall c \in P(C)$, $c \sqsubseteq \gamma(\alpha(c))$), and it is a closure, since $c \sqsubseteq \gamma(\alpha(c)) \sqsubseteq \gamma(\alpha(\gamma(\alpha(c)))) \sqsubseteq \gamma(\alpha(c))$ (the last inequality holds because γ is monotonic, and $\alpha(\gamma(\alpha(c))) \sqsubseteq \alpha(c)$).

5 Collecting Semantics

Another approach for describing the possible states of the program is using *collecting interpretation*. The simple idea is to collect all the possible states we may get in all the possible executions, as opposed to using the join operator. This technique can be used to collect either concrete or abstract states. As we see in the following two examples, it could provide a more precise analysis than what we get when using the join operator (Example 5), or it could never finish the execution and time out (Example 6).

Example 5. Consider the following program:

```
// We start with {[x: 0, y: 0, z: 0]}
z = 3 // {[x: 0, y: 0, z: 3]}
x = 1 // {[x: 1, y: 0, z: 3]}
while (x > 0) { // {[x: 1, y: 0, z: 3], [x: 3, y: 7, z: 3]}
    if (x == 1)
        y = 7 // {[x: 1, y: 7, z: 3]}
    else
        y = z + 4 // {[x: 3, y: 7, z: 3]}
    x = 3 // {[x: 3, y: 7, z: 3]}
    print y // {[x: 3, y: 7, z: 3]}
}
```

The comment after each line lists all the collected states. This example shows how using collecting interpretation, we may get a very precise set of final states.

Example 6. Consider the following program that just contains a simple while loop:

```
i = 0
while (true)
    i = i + 1
```

For this program, we will iteratively collect many states. We start with $\{[i \mapsto 0]\}$, then add another state and get $\{[i \mapsto 0], [i \mapsto 1]\}$, and this goes on until the verification times out. Conversely, if we used abstract interpretation, the verification would terminate almost immediately and return $[i \mapsto \top]$.

5.1 "Computing" Collecting Interpretation

Informally, we would like to generate a set of monotone functions (as in chaotic iterations) that describe the effects that different lines of code have on our (concrete or abstract) set of states. Each location in the program should be updated if one of its predecessors changed. We then want to find a minimal solution — *the collecting interpretation*. Note, however, that the solution may not be computable (Example 6).

The variables of our equations are $CS_{entry}(l)$ and $CS_{exit}(l)$ for every program location l (the entry and exit of that location). We write equations for elementary statements:

- $[skip]$: $CS_{exit}(l) = CS_{entry}(l)$.
- $[b]$ (the condition of some *if* or *while* statement): $CS_{exit}(l) = \{s \in CS_{entry}(l) : [\![b]\!]s = True\}$.
- $[x := a]$: $CS_{exit}(l) = \{s[x \mapsto [\![a]\!]s] : s \in CS_{entry}(l)\}$.

For control flow constructs we define $CS_{entry}(l) = \bigcup CS_{entry}(l')$, for all l' that immediately precede l in the CFG. The equation for the entry to the first location is $CS_{entry}(l_0) = \{s : s \in Var_* \to \mathbb{Z}\}$, i.e., we assume nothing about the input to the program.

This system of equations can be solved by directly using the chaotic iterations algorithm. In order to see that, it is sufficient to rewrite our updates for the elementary statements as one function that operates on the collecting interpretation state of the entire program. This function is obviously monotone: it only changes one variable from $CS_{entry}(l)$ to $CS_{exit}(l)$ at any time, and each of the updates we have written is monotone. Therefore, a least fixed point solution to the system of equations exists. However, we may never converge to it, if the height of the collecting interpretation lattice is infinite.

6 Soundness and Completeness

An abstract (or collecting) interpretation is *sound*, if $\alpha(lfp(f)) \sqsubseteq lfp(f^\sharp)$ or $lfp(f) \sqsubseteq \gamma(lfp(f^\sharp))$ (f is a function on the concrete domain and $f^\sharp$ is a function on the abstract domain). These two conditions are equivalent as (α, γ) is a Galois connection. We say that the interpretation is *sound and complete* if $\alpha(lfp(f)) = lfp(f^\sharp)$ or $lfp(f) = \gamma(lfp(f^\sharp))$. The first version states that performing operations in the abstract domain is the same as performing the appropriate operations in the concrete domain, and then using the abstraction function. The second version states that performing operations in the abstract domain and then using the concretization function is the same as performing the appropriate operations in the concrete domain.

Completeness of this sort is very hard to achieve (as illustrated in Figure 5). Since abstract states represent sets of concrete states, it makes sense that we will lose some precision. Hence, when we use concretization, we may also get non-achievable concrete states. Therefore, we normally settle for *soundness*.

Claim 6.1. *Let (α, γ) be functions that form a Galois connection from C to A. Let $f : C \to C$ and $f^\sharp : A \to A$ be monotone functions. The abstraction is* sound *if one of the following conditions holds:*

1. $\forall a \in A : f(\gamma(a)) \sqsubseteq \gamma(f^\sharp(a))$
2. $\forall c \in C : \alpha(f(c)) \sqsubseteq f^\sharp(\alpha(c))$
3. $\forall a \in A : \alpha(f(\gamma(a))) \sqsubseteq f^\sharp(a)$

Proof. We first show that the first condition implies soundness. Assume that $\forall a \in A$, $f(\gamma(a)) \sqsubseteq \gamma(f^\sharp(a))$, and choose $a = lfp(f^\sharp)$. It follows that $f(\gamma(lfp(f^\sharp))) \sqsubseteq \gamma(f^\sharp(lfp(f^\sharp))) = \gamma(lfp(f^\sharp))$. Hence, $\gamma(lfp(f^\sharp)) \in Red(f)$, and by Tarski's theorem, we get $lfp(f) \sqsubseteq \gamma(lfp(f^\sharp))$.

We now show that the other conditions imply the first condition. For the second condition, assume that $\forall c \in C$, $\alpha(f(c)) \sqsubseteq f^\sharp(\alpha(c))$. Let $a \in A$. We apply the condition on $c = \gamma(a)$, and get that $\alpha(f(\gamma(a))) \sqsubseteq f^\sharp(\alpha(\gamma(a)))$. Since $f^\sharp$ is monotone, and $\alpha(\gamma(a)) \sqsubseteq a$, we get that $\alpha(f(\gamma(a))) \sqsubseteq f^\sharp(a)$. We apply γ on both sides, and get that $f(\gamma(a)) \sqsubseteq \gamma(\alpha(f(\gamma(a)))) \sqsubseteq \gamma(f^\sharp(a))$. This holds for every $a \in A$, and that is exactly the first condition.

For the third condition, assume that $\forall a \in A$, $\alpha(f(\gamma(a))) \sqsubseteq f^\sharp(a)$. After applying γ on both sides, and using the properties of Galois connections, we get $f(\gamma(a)) \sqsubseteq \gamma(\alpha(f(\gamma(a)))) \sqsubseteq \gamma(f^\sharp(a))$. This is exactly the first condition, and soundness follows. □

6.1 Proving Soundness

In order to prove that an abstraction we have chosen is sound, we should use the following outline:

1. Define an "appropriate" structural operational semantics.
2. Define "collecting" structural operational semantics.
3. Establish a Galois connection between collecting states and reaching definitions.
4. Show that the abstract interpretation of every atomic statement is sound with respect to the collecting semantics. This is called *local correctness*.
5. Conclude that the analysis is sound. This is called *global correctness*.

References

[1] Patrick Cousot and Radhia Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1977, pp. 238–252.