Supplementary Note for Static Analysis: An Abstract Interpretation Perspective

Kwangkeun Yi

The 9th Summer School on Formal Techniques, May 2019

1 Order Relations and Ordered Sets

We assume a set E is given. Intuitively, an *order relation* over E is a mathematical object that specifies for which pairs x, y of elements of E we can say that x is "smaller" than y for some given notion of "smaller". Formally, an order relation is a binary relation $\preceq$ over E that is reflexive ($\preceq$ is *reflexive* if and only if $\forall x \in E,\ x \preceq x$), transitive ($\preceq$ is *transitive* if and only if $\forall x, y, z \in E,\ x \preceq y$ and $y \preceq z \implies x \preceq z$), and anti-symmetric ($\preceq$ is *anti-symmetric* if and only if $\forall x, y \in E,\ x \preceq y$ and $y \preceq x \implies x = y$).

An order relation is *total* when any pair of elements can be compared (i.e., if $\forall x, y \in E,\ x \preceq y \vee y \preceq x$).

In the following, we assume $\preceq$ defines an order relation over E. If there exists an element $x_0 \in E$ that is smaller than any other element of E for $\preceq$, we call x_0 the *infimum*; to distinguish it, we usually denote it by $\bot$ (read "bottom"). Similarly, the *supremum*, if it exists, is the element that is greater than any other element; it is usually denoted by $\top$ (read "top").

Let x and y be two elements of E. When it exists, we call *least upper bound* of x and y the element $x \sqcup y$ that is greater than x and y and is smaller than any other element with that property. While the existence of the least upper bound does not always hold, when it exists it is unique. In the case of the powerset, the least upper bound is simply the set union, and the greatest lower bound is the set intersection.

We say that an ordered set E is a *lattice* when it has an infimum and a supremum, and when each pair of elements has both a least upper bound and a greatest lower bound. Moreover, when any subset of E has both a least upper bound and a greatest lower bound, it is called a *complete lattice*. As an example, the powerset of any set is a complete lattice.

Additionally, we say that E is a *complete partial order* (for short, CPO) if it has an infimum, and is such that any chain of elements of E has a least upper bound in E.

Last, let us assume two ordered sets E and F (for simplicity we denote the order relations over both sets by $\preceq$) and a function $f : E \to F$. We say that f is *monotone* if and only if, for all $x, y \in E$ such that $x \preceq y$, we have $f(x) \preceq f(y)$. A stronger property is *continuity*: assuming that E and F are CPOs, we say that f is continuous if and only if the image of any chain G of E by f has a least upper bound, that is such that $\sqcup\{f(x) \mid x \in G\} = f(\sqcup G)$. It is very easy to show that, when a function is continuous, it is also monotone. We say that f is *extensive* if and only if, for all $x \in E$, we have $x \preceq f(x)$.

2 Operators over Ordered Structures and Fixpoints

We call *fixpoint* of f an element x such that $f(x) = x$. Moreover, when it exists, the *least-fixpoint* of f is the fixpoint of f that is smaller for $\preceq$ than any other fixpoint of f; it is denoted by $\mathbf{lfp}f$.

Theorem 1 (Kleene's fixpoint theorem) *If f is continuous and E is a CPO with infimum $\bot$, then f has a least fixpoint, that can be expressed as follows:*

$$\mathbf{lfp} f = \bigcup_{n \in \mathbb{N}} f^n(\bot)$$

Proof of Kleene's fixpoint theorem. First, we need to justify the existence of the least upper bound in the right hand side of the equality. As $\bot$ is the infimum of E, we have $\bot \preceq f(\bot)$. Since f is continuous, it is also monotone, thus we can prove by induction that, for all $n \in \mathbb{N}$, we have $f^n(\bot) \preceq f^{n+1}(\bot)$. Therefore $\{f^n(\bot) \mid n \in \mathbb{N}\}$ forms a chain. As E is a CPO, it has a least upper bound, that we denote by X.

Since the set of iterates of f from $\bot$ is a chain, we can also apply to it the continuity of f. This allows us to derive that $\{f^{n+1}(\bot) \mid n \in \mathbb{N}\}$ has a least upper bound, that is equal to $f(X)$:

$$f(X) = f(\bigcup_{n \in \mathbb{N}} f^n(\bot)) = \bigcup_{n \in \mathbb{N}} f^{n+1}(\bot) = \bot \cup \bigcup_{n \in \mathbb{N}} f^{n+1}(\bot) = \bigcup_{n \in \mathbb{N}} f^n(\bot) = X$$

We have proved that X is a fixpoint for f. To conclude, we simply need to prove that it is the least one, that is any fixpoint of f is greater than it. Let us assume X' is another fixpoint. Then we can prove by induction over n that $f^n(\bot) \preceq X'$, which is easy: it holds at rank 0 by definition of the infimum, and the property at rank n implies the property at rank $n+1$ thanks to the monotonicity, and to the fact that X' is a fixpoint. Thus, by definition of the least upper bound:

$$\bigcup_{n \in \mathbb{N}} f^n(\bot) \preceq X'$$

This concludes the proof.

3 Properties of Galois-connections

Theorem 2 (Properties of Galois connection) *Let $(\mathbb{C}, \subseteq)$ and $(\mathbb{A}, \sqsubseteq)$ be a concrete domain and an abstract domain with an abstraction function α and a concretization function γ that form a Galois-connection, that is:*

$$\forall c \in \mathbb{C},\ \forall a \in \mathbb{A}, \qquad \alpha(c) \sqsubseteq a \qquad \iff \qquad c \subseteq \gamma(a)$$

Then the following properties hold:

- $\forall c \in \mathbb{C},\ c \subseteq \gamma(\alpha(c))$ *(or equivalently,* $\mathrm{id} \subseteq \gamma \circ \alpha$*),*
- $\forall a \in \mathbb{A},\ \alpha(\gamma(a)) \sqsubseteq a$ *(or equivalently,* $\alpha \circ \gamma \sqsubseteq \mathrm{id}$*),*
- *γ and α are monotone, and*
- *if both $\mathbb{C}$ and $\mathbb{A}$ are CPOs (any chain has a least upper bound), then α is continuous.*

Proof. We assume the existence of the Galois-connection and prove items one by one:

($\mathrm{id} \sqsubseteq \gamma \circ \alpha$) Let c be a concrete element. By the reflexivity of the order relation $\sqsubseteq$, we have $\alpha(c) \sqsubseteq \alpha(c)$. Thus, by the definition of Galois connections, $c \subseteq \gamma(\alpha(c))$.

($\alpha \circ \gamma \sqsubseteq \mathrm{id}$) Let a be an abstract element. By the reflexivity of the order relation $\subseteq$, we have $\gamma(a) \subseteq \gamma(a)$. Thus, by the definition of Galois connections, $\alpha(\gamma(a)) \sqsubseteq a$.

(γ is monotone.) Let us assume c_0, c_1 are concrete elements such that $c_0 \subseteq c_1$. Then, because $\mathrm{id} \subseteq \gamma \circ \alpha$, $c_0 \subseteq c_1 \subseteq \gamma(\alpha(c_1))$. Hence, by the definition of Galois connections, $\alpha(c_0) \sqsubseteq \alpha(c_1)$.

(α is monotone.) Let us assume that a_0, a_1 are abstract elements such that $a_0 \sqsubseteq a_1$. Then, because $\alpha \circ \gamma \sqsubseteq \mathrm{id}$, $\alpha(\gamma(a_0)) \sqsubseteq a_1 \sqsubseteq a_1$. Hence, by the definition of Galois connections, $\gamma(a_0) \subseteq \gamma(a_1)$.

(α is continuous.) Let S be a chain in $\mathbb{C}$. Since $\mathbb{C}$ is a CPO, the least upper bound $\bigsqcup_{c \in S}$ exists in $\mathbb{C}$. Since α is monotone, for all element x in the chain S, we have $\alpha(x) \sqsubseteq \alpha(\bigcup_{c \in S} c)$ and thus $\bigsqcup_{c \in S} \alpha(c) \sqsubseteq \alpha(\bigcup_{c \in S} c)$.

The function $\gamma \circ \alpha$ is monotone so it transforms a chain into a chain, thus $\{\gamma(\alpha(c)) \mid c \in S\}$ is a chain and has its least upper bound in CPO $\mathbb{C}$. Moreover, $\mathrm{id} \subseteq \gamma \circ \alpha$, thus:

$$\bigcup_{c \in S} c \subseteq \bigcup_{c \in S} \gamma(\alpha(c))$$

The function γ is monotone, and for all element c in the chain S, we have $\alpha(c) \sqsubseteq \bigsqcup_{c \in S} \alpha(c)$ and thus $\gamma(\alpha(c)) \subseteq \gamma(\bigsqcup_{c \in S} \alpha(c))$. Therefore:

$$\bigcup_{c \in S} \gamma(\alpha(c)) \subseteq \gamma(\bigsqcup_{c \in S} \alpha(c)),$$

As we compose the two above inequalities, we get:

$$\bigcup_{c \in S} c \subseteq \gamma(\bigsqcup_{c \in S} \alpha(c)),$$

By the definition of Galois connections, this entails that $\alpha(\bigcup_{c \in S} c) \sqsubseteq \bigsqcup_{c \in S} \alpha(c)$. Hence, $\alpha(\bigcup_{c \in S} c) = \bigsqcup_{c \in S} \alpha(c)$. This concludes the proof.

4 Transitional-style Static Analysis Framework

4.1 Notations

- An element of $A \to B$, which is a map from A to B, is interchangeably an element in $\wp(A \times B)$. For example, an element in $\mathbb{L} \to \mathbb{M}^\sharp$ of the abstract states is interchangeably an element in $\wp(\mathbb{L} \times \mathbb{M}^\sharp)$, a set (so called *graph*) of pairs of labels and abstract memories. Note that in the above examples of this subsection we already used this graph notation to represent the abstract state function.

- A relation $f \subseteq A \times B$ is interchangeably a function $f \in A \to \wp(B)$ defined as

  $$f(a) = \{b \mid (a, b) \in f\}.$$

  For example, the concrete one-step transition relation $\hookrightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is interchangeably a function $\hookrightarrow \in \mathbb{S} \to \wp(\mathbb{S})$.

- For function $f : A \to B$, we write $\wp(f)$ for its powerset version defined as:

  $$\begin{array}{l} \wp(f) : \wp(A) \to \wp(B) \\ \wp(f)(X) = \{f(x) \mid x \in X\} \end{array}$$

- For function $f : A \to \wp(B)$, we write $\breve{\wp}(f)$ as a shorthand for $\cup \circ \wp(f)$

  $$\begin{array}{l} \breve{\wp}(f) : \wp(A) \to \wp(B) \\ \breve{\wp}(f)(X) = \bigcup\{f(x) \mid x \in X\}. \end{array}$$

  For example, powerset-lifted function $\mathit{Step} : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ of relation $\hookrightarrow$

  $$\mathit{Step}(X) = \{s' \mid s \hookrightarrow s', s \in X\}$$

  is equivalently, by regarding $\hookrightarrow$ as a function of $\mathbb{S} \to \wp(\mathbb{S})$,

  $$\mathit{Step} = \breve{\wp}(\hookrightarrow).$$

- For functions $f : A \to B$ and $g : A' \to B'$, we write (f, g) for

  $$\begin{array}{l} (f, g) : A \times A' \to B \times B' \\ (f, g)(a, a') = (f(a), g(a')). \end{array}$$

4.2 Framework

1. Let $\mathbb{M}$ to be the set of memory states that can occur during program executions. Let $\mathbb{L}$ be the finite and fixed set of labels of a given program.

2. Let a concrete semantics be the $\mathbf{lfp}F$ where

$$\begin{array}{rll} \text{concrete domain} & \wp(\mathbb{S}) & = \wp(\mathbb{L}\times\mathbb{M}) \\ \text{concrete semantic function} & F:\wp(\mathbb{S})\to\wp(\mathbb{S}) & \\ & F(X) & = I\cup \mathit{Step}(X) \\ & \mathit{Step} & = \breve{\wp}(\hookrightarrow) \\ & \hookrightarrow & \subseteq (\mathbb{L}\times\mathbb{M})\times(\mathbb{L}\times\mathbb{M}) \end{array}$$

   Let $\hookrightarrow$ be the one-step transition relation over $\mathbb{L}\times\mathbb{M}$.

3. Let its abstract domain and abstract semantic function be

$$\begin{array}{rll} \text{abstract domain} & \mathbb{S}^\sharp & = \mathbb{L}\to\mathbb{M}^\sharp \\ \text{abstract semantic function} & F^\sharp:\mathbb{S}^\sharp\to\mathbb{S}^\sharp & \\ & F^\sharp(X^\sharp) & = \alpha(I)\cup^\sharp \mathit{Step}^\sharp(X^\sharp) \\ & \mathit{Step}^\sharp & = \wp(\mathrm{id},\sqcup_M)\circ\pi\circ\breve{\wp}(\hookrightarrow^\sharp) \\ & \hookrightarrow^\sharp & \subseteq (\mathbb{L}\times\mathbb{M}^\sharp)\times(\mathbb{L}\times\mathbb{M}^\sharp) \end{array}$$

   The $\hookrightarrow^\sharp$ is the one-step abstract transition relation over $\mathbb{L}\times\mathbb{M}^\sharp$. Function π partitions a set $\subseteq \mathbb{L}\times\mathbb{M}^\sharp$ by the labels in $\mathbb{L}$ returning an element in $\mathbb{L}\to\wp(\mathbb{M}^\sharp)$ represented as a set $\subseteq \mathbb{L}\times\wp(\mathbb{M}^\sharp)$.

4. Let the abstract domains $\mathbb{S}^\sharp$ and $\mathbb{M}^\sharp$ be CPOs, and form a Galois-connection respectively with $\wp(\mathbb{S})$ and $\wp(\mathbb{M})$:

$$(\wp(\mathbb{S}),\subseteq)\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}(\mathbb{S}^\sharp,\sqsubseteq)\quad\text{and}\quad(\wp(\mathbb{M}),\subseteq)\underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}}(\mathbb{M}^\sharp,\sqsubseteq_M)$$

   where the partial order $\sqsubseteq$ of $\mathbb{S}^\sharp$ is label-wise $\sqsubseteq_M$:

$$a^\sharp\sqsubseteq b^\sharp \quad\text{iff}\quad \forall l\in\mathbb{L}: a^\sharp(l)\sqsubseteq_M b^\sharp(l).$$

5. Let the abstract one-step transition $\hookrightarrow^\sharp$ and abstract union $\cup^\sharp$ satisfy:

$$\begin{array}{rl} \breve{\wp}(\hookrightarrow)\circ\gamma & \subseteq \gamma\circ\breve{\wp}(\hookrightarrow^\sharp) \\ \cup\circ(\gamma,\gamma) & \subseteq \gamma\circ\cup^\sharp \end{array}$$

Theorem 3 (Correct static analysis by $F^\sharp$) *If $\mathbb{S}^\sharp$ is of finite-height (every chain in $\mathbb{S}^\sharp$ is finite) and $F^\sharp$ is monotone or extensive, then*

$$\bigsqcup_{i\geq 0} {F^\sharp}^i(\bot)$$

is finitely computable and over-approximates $\mathbf{lfp}F$*:*

$$\mathbf{lfp}F\ \subseteq\ \gamma(\bigsqcup_{i\geq 0}{F^\sharp}^i(\bot))\quad \textit{or equivalently}\quad \alpha(\mathbf{lfp}F)\ \sqsubseteq\ \bigsqcup_{i\geq 0}{F^\sharp}^i(\bot).$$

Proof. Proof structure is as follows:

$$\begin{array}{rll} \text{the conditions} & \Longrightarrow & F\circ\gamma\subseteq\gamma\circ F^\sharp & (1)\\ & \Longrightarrow & \forall n\geq 0: F^n\bot\subseteq\gamma({F^\sharp}^n(\bot)) & (2)\\ & \Longrightarrow & \mathbf{lfp}F\ \subseteq\ \gamma(\bigsqcup_{i\geq 0}{F^\sharp}^i(\bot)). & (3)\end{array}$$

First, note that $\bigsqcup_i {F^\sharp}^i(\bot)$ exists in CPO $\mathbb{S}^\sharp$, because that $F^\sharp$ is monotone or extensive implies the sequence $\bot, {F^\sharp}^i\bot, {F^\sharp}^2\bot, \cdots$ is a chain.

(Proof of 1) From the condition $\breve{\wp}(\hookrightarrow) \circ \gamma \ \subseteq \ \gamma \circ \breve{\wp}(\hookrightarrow^\sharp)$, follows $\mathit{Step} \circ \gamma \ \subseteq \ \gamma \circ \mathit{Step}^\sharp$. Recall that $\mathit{Step} = \breve{\wp}(\hookrightarrow)$ and $\mathit{Step}^\sharp = (\mathrm{id}, \sqcup_M) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp)$. Note that $\breve{\wp}(\hookrightarrow^\sharp) \sqsubseteq ((\mathrm{id}, \sqcup_M) \circ \pi) \circ \breve{\wp}(\hookrightarrow^\sharp)$, because the partial order $\sqsubseteq$ is label-wise and the $(\mathrm{id}, \sqcup_M) \circ \pi$ operation partitions the result set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$ of $\breve{\wp}(\hookrightarrow^\sharp)$ by the labels and returns the least upper bound ($\sqcup_M$) of abstract memories in each partition. Hence,

$$\begin{array}{rcll} \mathit{Step} \circ \gamma \ = \ \breve{\wp}(\hookrightarrow) \circ \gamma & \subseteq & \gamma \circ \breve{\wp}(\hookrightarrow^\sharp) & \text{by the condition of } \hookrightarrow^\sharp \\ & \subseteq & \gamma \circ (\mathrm{id}, \sqcup_M) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp) & \text{by the monotonicity of } \gamma \\ & = & \gamma \circ \mathit{Step}^\sharp. & \end{array}$$

Then it follows that $F \circ \gamma \subseteq \gamma \circ F^\sharp$, because

$$\begin{array}{rcll} (\gamma \circ F^\sharp)X & = & \gamma(\alpha I \cup^\sharp \mathit{Step}^\sharp(X)) & \\ & \supseteq & (\gamma \circ \alpha)I \ \cup \ (\gamma \circ \mathit{Step}^\sharp)X & \text{by the condition of } \cup^\sharp \\ & \supseteq & I \ \cup \ (\mathit{Step} \circ \gamma)X & \text{by } \mathit{Step} \circ \gamma \subseteq \gamma \circ \mathit{Step}^\sharp \text{ and } \mathrm{id} \subseteq \gamma \circ \alpha \\ & = & (F \circ \gamma)X. & \end{array}$$

(Proof of 2) From $F \circ \gamma \subseteq \gamma \circ F^\sharp$ it follows that

$$\forall n \geq 0 : F^n\bot \subseteq \gamma({F^\sharp}^n\bot)$$

by induction on n. Basis case $n = 0$ holds since $F^0(\bot) = \bot$. For inductive cases, assume $F^k(\bot) \subseteq \gamma({F^\sharp}^k(\bot))$. Then since F is monotone (since F is continuous),

$$\begin{array}{rcll} F(F^k\bot) & \subseteq & F(\gamma({F^\sharp}^k\bot) & \text{by induction hypothesis} \\ & \subseteq & \gamma(F^\sharp({F^\sharp}^k\bot)). & \text{by } F \circ \gamma \subseteq \gamma \circ F^\sharp \end{array}$$

(Proof of 3) From $\forall n \geq 0 : F^n\bot \subseteq \gamma({F^\sharp}^n\bot)$, we prove the final goal

$$\bigsqcup_{i \geq 0} F^i\bot \subseteq \gamma(\bigsqcup_{i \geq 0} {F^\sharp}^i\bot).$$

Note that $\bigsqcup_{i \geq 0} F^i(\bot)$ exists because sequence $(F^i(\bot))_{i \geq 0}$ is a chain in CPO $\wp(\mathbb{S})$, and $\bigsqcup_{i \geq 0} \gamma({F^\sharp}^i\bot)$ also exists in CPO $\mathbb{S}^\sharp$ becasue that $({F^\sharp}^i\bot)_{i \geq 0}$ is a chain and γ is monotone imply that $(\gamma({F^\sharp}^i\bot))_{i \geq 0}$ is a chain. Thus, from $\forall n \geq 0 : F^n\bot \subseteq \gamma({F^\sharp}^n\bot)$ it follows that

$$\begin{array}{rcll} \bigsqcup_{i \geq 0} F^i\bot & \subseteq & \bigsqcup_{i \geq 0} \gamma({F^\sharp}^i\bot) & \\ & \subseteq & \gamma(\bigsqcup_{i \geq 0} {F^\sharp}^i\bot). & \text{by the monotonicity of } \gamma \end{array}$$

This concludes the proof.

Definition 1 (Widening operator) *A* widening *operator over an abstract domain $\mathbb{A}$ is a binary operator $\triangledown$, such that:*

1. *For all abstract elements a_0, a_1, we have*

$$\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \triangledown a_1)$$

2. *For all sequence $(a_n)_{n \in \mathbb{N}}$ of abstract elements, the sequence $(a'_n)_{n \in \mathbb{N}}$ defined below is ultimately stationary:*

$$\left\{ \begin{array}{rcl} a'_0 & = & a_0 \\ a'_{n+1} & = & a'_n \triangledown a_n \end{array} \right.$$

Theorem 4 (Correct static analysis by $F^\sharp$ and widening operator $\bigtriangledown$) *Let $\bigtriangledown$ be a widening operator. Then the following chain $Y_0 \sqsubseteq Y_1 \sqsubseteq \cdots$*

$$Y_0 = \bot \qquad Y_{i+1} = Y_i \bigtriangledown F^\sharp(Y_i)$$

is finite and its last element Y_{lim} over-approximates $\mathbf{lfp}F$:

$$\mathbf{lfp}F \subseteq \gamma(Y_{\mathrm{lim}}) \quad \textit{or equivalently} \quad \alpha(\mathbf{lfp}F) \sqsubseteq Y_{\mathrm{lim}}.$$

Proof. First, the sequence $(Y_i)_{i\geq 0}$ is a chain. The widening operator's first condition $\gamma(a \bigtriangledown b) \supseteq \gamma(a) \cup \gamma(b)$ means

$$\begin{array}{lll} a \bigtriangledown b & \sqsupseteq & \alpha(\gamma(a) \cup \gamma(b)) \quad \text{by Galois-connection} \\ & \sqsupseteq & \alpha(\gamma(a)) \sqcup \alpha(\gamma(b)) \quad \text{by the monotonicity of } \alpha \\ & \sqsupseteq & a \sqcup b \quad \text{by id} \sqsubseteq \alpha \circ \gamma \end{array}$$

hence $a \bigtriangledown b \sqsupseteq a$ and $a \bigtriangledown b \sqsupseteq b$. Thus $\forall k \geq 0 : Y_k \sqsubseteq Y_{k+1} = Y_k \bigtriangledown F^\sharp(Y_k)$.

Second, by the second condition of the widening operator $\bigtriangledown$, the chain $(Y_i)_{i\geq 0}$ is finitely stationary. Let the last element be Y_{lim}.

We prove by induction that

$$\forall n \geq 0 : \bigcup_{i=0}^{n} F^k(\bot) \subseteq \gamma(Y_k), \tag{4}$$

which implies, because Y_{lim} is the biggest element of $(Y_i)_i$,

$$\forall n \geq 0 : \bigcup_{i=0}^{n} F^k(\bot) \subseteq \gamma(Y_{\text{lim}}),$$

hence

$$\bigcup_{i\geq 0} F^k(\bot) \subseteq \gamma(Y_{\text{lim}}).$$

(Proof of 4) Base case $n = 0$ is obvious. For inductive cases, assume $\bigcup_{i=0}^{k} F^i(\bot) \subseteq \gamma(Y_k)$.

$$\begin{array}{lll} \bigcup_{i=0}^{k+1} F^i(\bot) & = & \bigcup_{i=0}^{k} F^i(\bot) \cup F^{k+1}(\bot) \\ & \subseteq & \gamma(Y_k) \cup F^{k+1}(\bot) \quad \text{by induction hypothesis} \\ & = & \gamma(Y_k) \cup F(F^k(\bot)) \\ & \subseteq & \gamma(Y_k) \cup F(\bigcup_{i=0}^{k} F^i(\bot)) \quad \text{by the monotonicity of } F \\ & \subseteq & \gamma(Y_k) \cup F(\gamma(Y_k)) \quad \text{by induction hypothesis and the monotonicity of } F \\ & \subseteq & \gamma(Y_k) \cup \gamma(F^\sharp(Y_k)) \quad \text{by } F \circ \gamma \subseteq \gamma \circ F^\sharp \\ & \subseteq & \gamma(Y_k \bigtriangledown F^\sharp(Y_k)) \quad \text{by the definition of } \bigtriangledown \\ & = & \gamma(Y_{k+1}). \quad \text{by the definition of } Y_{k+1} \end{array}$$

This concludes the proof.

4.3 Use Example of the Transitional-style Static Analysis

Theorem 5 (Soundness of $\hookrightarrow^\sharp$) *Consider the concrete one-step transition relation and the abstract transition relation of the example slides. If the semantic operators satisfy the following soundness properties:*

$$\begin{array}{rcl} \wp(\mathit{eval}_E) \circ \gamma_M & \subseteq & \gamma_V \circ \mathit{eval}_E^\sharp \\ \wp(\mathit{update}_{\mathtt{x}}) \circ \times \circ (\gamma_M, \gamma_V) & \subseteq & \gamma_M \circ \mathit{update}_{\mathtt{x}}^\sharp \\ \wp(\mathit{filter}_B) \circ \gamma_M & \subseteq & \gamma_M \circ \mathit{filter}_B^\sharp \\ \wp(\mathit{filter}_{\neg B}) \circ \gamma_M & \subseteq & \gamma_M \circ \mathit{filter}_{\neg B}^\sharp \end{array}$$

then $\breve{\wp}(\hookrightarrow) \circ \gamma \sqsubseteq \gamma \circ \breve{\wp}(\hookrightarrow^\sharp)$. *(The $\times$ is the Cartesian product operator of two sets.)*

Proof. Without loss of generality, let us consider a singleton set $\{(l, m^\sharp)\} \in \mathbb{S}^\sharp$ and prove for the set the conclusion holds. We proceed by case analysis for the command at the l label.

(Case of "$\texttt{if}\ B\ C_1\ C_2$")

$$
\begin{array}{ll}
 & (\breve{\wp}(\hookrightarrow) \circ \gamma)\{(l, M^\sharp)\} \\
= & \breve{\wp}(\hookrightarrow)\{(l, m) \mid m \in \gamma_M(M^\sharp)\} \\
= & \{(\texttt{nextTrue}(l), m_1) \mid m_1 \in \wp(\mathit{filter}_B)(\gamma_M(M^\sharp))\} \cup \{(\texttt{nextFalse}(l), m_2) \mid m_2 \in \wp(\mathit{filter}_{\neg B})(\gamma_M(M^\sharp))\} \\
\subseteq & \{(\texttt{nextTrue}(l), m_1) \mid m_1 \in \gamma_M(\mathit{filter}^\sharp{}_B(M^\sharp))\} \cup \{(\texttt{nextFalse}(l), m_2) \mid m_2 \in \gamma_M(\mathit{filter}^\sharp{}_{\neg B}(M^\sharp))\} \\
 & \text{by the conditions of } \mathit{filter}^\sharp_B \text{ and } \mathit{filter}^\sharp_{\neg B} \\
= & \gamma\{(\texttt{nextTrue}(l), \mathit{filter}^\sharp{}_B(M^\sharp)), (\texttt{nextFalse}(l), \mathit{filter}^\sharp{}_{\neg B}(M^\sharp))\} \\
= & (\gamma \circ \breve{\wp}(\hookrightarrow^\sharp))\{(l, M^\sharp)\}.
\end{array}
$$

(Case of "$\texttt{x} := E$")

$$
\begin{array}{lll}
 & (\breve{\wp}(\hookrightarrow) \circ \gamma)\{(l, M^\sharp)\} & \\
= & \breve{\wp}(\hookrightarrow)\{(l, m) \mid m \in \gamma_M(M^\sharp)\} & \\
= & \{(\texttt{next}(l), (\mathit{update}_{\texttt{x}} \circ (\mathrm{id}, \mathit{eval}_E))m) \mid m \in \gamma_M(M^\sharp)\} & \text{where notation } (f, g)x \ = \ (f(x), g(x)) \\
\subseteq & \{(\texttt{next}(l), m) \mid m \in (\wp(\mathit{update}_{\texttt{x}}) \circ \times \circ (\mathrm{id}, \wp(\mathit{eval}_E)) \circ \gamma_M)M^\sharp\} & \\
= & \{(\texttt{next}(l), m) \mid m \in (\wp(\mathit{update}_{\texttt{x}}) \circ \times \circ (\gamma_M, \wp(\mathit{eval}_E) \circ \gamma_M))M^\sharp\} & \\
\subseteq & \{(\texttt{next}(l), m) \mid m \in (\wp(\mathit{update}_{\texttt{x}}) \circ \times \circ (\gamma_M, \gamma_V \circ \mathit{eval}^\sharp{}_E))M^\sharp\} & \text{by the condition of } \mathit{eval}^\sharp{}_E \\
= & \{(\texttt{next}(l), m) \mid m \in (\wp(\mathit{update}_{\texttt{x}}) \circ \times \circ (\gamma_M, \gamma_V) \circ (\mathrm{id}, \mathit{eval}^\sharp{}_E))M^\sharp\} & \\
\subseteq & \{(\texttt{next}(l), m) \mid m \in (\gamma_M \circ \mathit{update}^\sharp{}_{\texttt{x}} \circ (\mathrm{id}, \mathit{eval}^\sharp{}_E))M^\sharp\} & \text{by the condition of } \mathit{update}^\sharp{}_{\texttt{x}} \\
= & \gamma\{(\texttt{next}(l), (\mathit{update}^\sharp{}_{\texttt{x}} \circ (\mathrm{id}, \mathit{eval}^\sharp{}_E))M^\sharp)\} & \\
= & \gamma\{(\texttt{next}(l), \mathit{update}^\sharp{}_{\texttt{x}}(M^\sharp, \mathit{eval}^\sharp{}_E(M^\sharp)))\} & \\
= & (\gamma \circ \breve{\wp}(\hookrightarrow^\sharp))\{(l, M^\sharp)\}. &
\end{array}
$$

Other cases similarly holds. This concludes the proof.

Comments. The underlying structure of the proof is fairly simple. Note that the $\hookrightarrow$ and $\hookrightarrow^\sharp$ functions are compositions of semantic operators, both of which are homomorphic to each other. Only difference is the operators involved. The $\hookrightarrow$ uses concrete operators and the $\hookrightarrow^\sharp$ uses their abstract correspondents. The above proof of each case is basically a replay in a different guise of the proof that if every pair of concrete and its abstract operators satisfies the soundness property then a homomorphic pair of their compositions preserve the soundness property.

The soundness preservation over composition is a common property that most of the abstract interpretations enjoy.

1 Program Analysis

Goal of the chapter: Before we dive into the way static analysis tools operate, we need to define their scope and describe the kind of questions they can help solve. This is the purpose of this Chapter. First, we discuss in Section 1.1 the importance of understanding the behavior of programs by semantic reasoning, and we show applications in Section 1.2. Section 1.3 sets up the main concepts of static analysis and shows the intrinsic limitations of automatic program reasoning techniques that are based on semantics. Section 1.4 classifies the main approaches to semantic-based program reasoning and clarifies the position of static analysis in this landscape.

Recommended reading: **[S]**, **[D]**, **[U]**.

1.1 Understanding Software Behavior

In every engineering discipline, a fundamental question is: Will our design work in reality as we intended? We ask and answer that question when we design mechanical machines, electrical circuits, or chemical processes. The answer comes from analyzing our designs using our knowledge about nature that will carry out the designs. For example, using Newtonian mechanics, Maxwell equations, Navier-Stokes equations, or thermo-dynamic equations, we analyze our design to predict its actual behavior. When we design a bridge, for example, we analyze how nature runs the design, namely how various forces (gravitation, wind, vibration, etc.) are applied to the bridge and whether the bridge structure is strong enough to stand against them.

The same question applies to computer software. We want to ensure that our software will work as intended. The intention ranges widely. For general reliability, we want to ensure that the software will not crash with abrupt termination. If the software interacts with the outside world, we want to ensure it will not be deceived to violate the host computer's security. For specific functionalities, we want to check if the software will realize its functional goal. If the software is to control cars we want to ensure it will not drive them to an accident. If the software is to learn our preference, we want to ensure it will not

degrade as we teach more. If the software transforms the medical images of our bodies, we want to ensure it will not introduce bogus pixels. If the software is to bookkeep the ledgers for crypto currency, we want to ensure it will not allow double spending. If the software translates program text, we want to ensure the source's meaning is not lost in translation.

There is, however, one difference. For computer software, it is not nature that will carry out the software. It is the computer itself.

The computer will carry out a software, namely execute the software according to the meanings of the software's source language. Software's run-time behavior is solely defined by the meanings of the software's source language. The computer is just an undiscerning tool that blindly executes the software exactly as it is written. Any execution behavior that deviates from our intention is because the software is mistakenly written to behave that way.

Hence, to answer the question for computer software, we need knowledge by which we can somehow analyze the meanings of software source language. Such knowledge corresponds to what natural sciences have accumulated about nature. We need knowledge that computer science has accumulated about handling the meanings of software source languages.

We call a formal definition of a software's run-time behavior, which is determined by its source language's meanings, *semantics*:

Definition 1.1 (Semantics and Semantic Properties) *The* semantics *of a program is a (generally formal —although we do not make it so in this chapter) description of its run-time behaviors. We call* semantic property *any property about the run-time behavior (semantics) of a program.*

Hence, *checking if a software will run as we intended* is equivalent to *checking if this software satisfies a semantic property of interest.*

In the following, we call a technique that aims at checking that a program satisfies a semantic property *program analysis*, and we refer to an implementation of program analysis as a *program analysis tool*.

Figure 1.1 illustrates the correspondence between program analysis and design analysis of other engineering disciplines.

1.2 Program Analysis Applications and Challenge

Program analysis can be applied wherever understanding program semantics is important or beneficial. First, software developers (both humans and machines) may be the biggest beneficiaries. Software developers can use program analysis for quality assurance, to locate errors of any kind in their software. Software maintainers can use program analysis to understand legacy software that they maintain. System security gatekeepers can use program analysis to proactively screen out programs whose semantics can be malicious.

	Computing area	Other engineering areas
Object	software	machine/building/circuit/chemical process design
Execution subject	computer runs it	nature runs it
Our question	will it work as intended?	will it work as intended?
Our knowledge	program analysis	Newtonian mechanics, Maxwell equations, Navier-Stokes equations, thermo-dynamic equations, and so on.

Figure 1.1
Program analysis addresses a basic question common in every engineering area

Software that handles programs as data can use program analysis for their performance improvement too. Language processors such as translators or compilers need program analysis to translate the input programs into optimized ones. Interpreters, virtual machines, or query processors need program analysis for optimized execution of the input programs. Automatic program synthesizers can use program analysis to check and tune what it synthesizes. Mobile operating systems need to understand application's semantics in order to minimize the energy consumption of the application. Automatic tutoring systems for teaching programming can use program analysis to hint to students a direction to amend their faulty programs.

Use of program analysis is not limited to professional software or their developers. As programming becomes a way of living in a highly-connected digitized environment, citizen programmers can benefit from program analysis too to sanity-check their daily program snippets.

The target of program analysis is not limited to executable software either. Once the object's source language has semantics, program analysis can circumscribe its semantics to provide useful information. For example, program analysis of high-level system configuration scripts can provide information about any existing conflicting requests.

Though the benefits of program analysis are obvious, building a cost-effective program analysis is not trivial since computer programs are complex and often very large. For example, the number of lines of smartphone applications frequently reaches over half million, not to mention larger software such as web browsers or operating systems whose source sizes are over ten-million lines. Semantic-wise, the situation is much worse because a program execution is highly dynamic. Programs usually need to react to inputs from the external, uncontrolled environments. The number of inputs, not to mention the number of program states, that can arise in all possible use cases, is so huge that software developers are likely to fail to handle some corner cases. The number can be easily greater than the number of atoms in the universe, for example. The space of the inputs keep exploding as

we want our software to do more things. Also, constraints that could keep software simple and small quickly diminish because of the ever growing capacity of computer hardware.

Given that software is in charge of almost all infrastructures in our personal, social, and global life, the need for a cost-effective program analysis technology has grown bigger than ever before. We have already been experiencing a sequence of appalling accidents whose causes are identified as mistakes in software. Such accidents have occurred in almost all sectors, including space, medical, military, electric power transmission, telecommunication, security, transportation, business, and administration. To name a few software accidents, the large-scale Twitter outage (2016), the fMRI software error (2016) that invalidates 15 years of brain research, the Heartbleed bug (2014) in the popular OpenSSL cryptographic library that allows attackers to read the memory of any server that uses certain instances of OpenSSL, the stack overflow issues that we can explain the Toyota sudden unintended acceleration (2004-2014), the Northeast blackout (2003), the explosion of the Ariane 5, Vol 501 (1996) that took 10 years and $7 billion to build, and Missile Patriot, Dahran (1991) that failed to intercept an incoming Scud missile.

Though building an error-free software may be far-fetched at least with a reasonable cost for large-scale software, cost-effective ways to reduce as many errors as possible is always in high demand.

Static analysis, which is the focus of this book, is one kind of program analysis. We will conclude this chapter by characterizing the static analysis in comparison with other program analysis techniques.

1.3 Concepts in Program Analysis

In the rest of this chapter, we intend to characterize static program analysis and compare it with other program analysis techniques. Towards this goal, we provide keys to understand how each program analysis technique operates and to assess the strengths and weaknesses of each of them. This characterization will give basic intuitions of static analysis' strength and limitations.

1.3.1 What to Analyze

The first question to answer to characterize program analysis techniques is *what programs* they analyze in order to determine *what properties*.

Target programs. An obvious characterization of the target programs to analyze is the programming languages that the programs are written in, however it is not the only one.

- **Domain-specific analyses**: certain analyses are aimed at specific families of programs. This specialization is a pragmatic way to achieve a cost-effective program analysis. It is because each family has a particular set of characteristics (such as program idioms) on which a program analysis can focus.

For example, consider the C programming language. Though the language is widely used to write software including operating systems, embedded controllers, and all sorts of utilities, each family of programs has a special character. Embedded software is often safety-critical (thus needs thorough verification) but rarely uses the most complex features of the C language (such as recursion, dynamic memory allocation, non-local jumps `setjump/longjump`), which makes analyzing such programs typically easier than analyzing general applications. Device drivers usually rely on low-level operations that are harder to reason about (e.g., low-level access to sophisticated data-structures) but are often of moderate size (a few thousand lines of code).

- **Non-domain-specific analyses**: some analyses are designed without focus on a particular family of programs of the target language. Such analyses are usually those incorporated inside compilers, interpreters, or general-purpose programming environments. Such analyses collect information (e.g., constants variables, common errors such as buffer-overruns) about the input program in order to help compilers, interpreters, or programmers for an optimized or safe execution of the program.
  Non-domain-specific analyses risk being less precise and cost-effective than domain-specific ones in order to have an overall acceptable performance for a wide range of programs.

Besides the language and family of programs that are considered, the way input programs are handled may also vary and has an impact on how the analysis works. An obvious option is to directly handle source program just like a compiler would, but some analyses may input different descriptions of programs instead. We can distinguish two classes of techniques:

- **Program-level analyses** are run on the source code of programs (for instance written in C or in Java) or on executable program binaries, and typically involve a frontend similar to a compiler's that constructs the syntax trees of programs from the program source or compiled files;
- **Model-level analyses** consider a different input language that aims at modeling the semantics of programs; then the analyses do not input a program in a language such as C or Java, but a description that *models* the program to analyze. Such models either need to be constructed manually or are computed by a separate tool. In both cases, the construction of the model may hide either difficulties or sources of inaccuracy that need be precisely taken into account.

Target properties. A second obvious element of characterization of a program analysis is the set of semantic properties it aims at computing. Among the most important families of target properties, we can cite safety properties, liveness properties, and information flow properties.

- A **safety property** essentially states that a program will never exhibit a behavior observable within finite time. Such behaviors include termination, computing a particular

set of values, and reaching some kind of error state (such as integer overflows, buffer overruns, uncaught exceptions or dead-locks).

Hence, a program analysis for some safety properties chases program behaviors that are observable within the finite time.

Historically this class is called *safety property* because the goal of the analysis is to prove the absence of bad behaviors and the bad behaviors are mostly those that occur on finite executions.

- A **liveness property** essentially states that a program will never exhibit a behavior observable only after infinite time. Examples of such behaviors include non-termination, live-lock, or starvation.

  Hence, program analysis for liveness properties searches for an existence of program behaviors that are observable after infinite time.
- **Information flow properties** define a large class of program properties stating the absence of dependence between pairs of program behaviors (for instance, in the case of a web service, a user should not be able to derive the credential of another user from the information she can access). Unlike safety and liveness properties, information flow properties require reasoning about pairs of executions.

  More generally, so-called **hyperproperties** define a class of program properties that are characterized by predicates over several program executions.

The techniques to reason about these classes of semantic properties are different. As observed above, safety properties only require considering finite executions, whereas liveness properties require reasoning about infinite executions. As a consequence, the program analysis techniques and algorithms dedicated to each family of semantic properties will differ as well.

1.3.2 Static vs Dynamic

An important characteristic of a program analysis technique is *when* it is performed, or more precisely, whether it operates during or before program execution.

A first solution is to make the analysis at run-time, that is during the execution of the program. Such an approach is called *dynamic*, as it takes place while the program computes, typically over several executions.

Example 1.1 (User assertions) *User assertions provide a very classic case of dynamic approach to checking whether some conditions are satisfied by all program executions. Once the assertions are inserted, the process is purely dynamic: whenever an assertion is executed, its condition is evaluated, and an error is returned if the result is* **false**.

Note that some programming languages perform run-time checking of specific properties: for instance, in Java, any array access is preceded by a dynamic bound check, which returns an exception if the index is not valid; this mechanism is equivalent to an assertion and is also dynamic.

A second solution is to make the analysis *before* program execution. We call such an approach a *static* analysis, as it is done once and for all and independently from any execution.

Example 1.2 (Strong typing) *Many programming languages require compilers to carry out some type checking stage, which ensures that certain classes of errors will never occur during the execution of the input program. This is a perfect example of a static analysis since typing takes place independently from any program execution, and the result is known before the program actually runs.*

Static and dynamic techniques are radically different and come with distinct sets of advantages and drawbacks. While dynamic approaches are often easier to design and implement, they also often incur a performance cost at run-time, and they do not force developers to fix issues before program execution. On the other hand, after a static analysis is done once, the program can be run as usual, without any slow down. Also, some properties cannot be checked dynamically: as an example, if the property of interest is termination, dynamically detecting a non terminating execution would require constructing an infinite program run. Dynamic and static analyses have different aftermath once they detect a property violation. A dynamic analysis upon detecting a property violation can simply abort the program execution or apply a non-obtrusive surgery to the program state and let the execution continue with a risk of having behaviors unspecified in the programs. On the other hand, when a static analysis detects a property violation, developers can still fix the issue before their software is in use.

1.3.3 A Hard Limit: Uncomputability

Given a language of programs to analyze and a property of interest, one would wish an ideal program analysis that always computes in a fully automated way the exact result in finite time. For instance, let us consider the certification that a program (e.g., a piece of safety-critical embedded software) will never crash due to a runtime error. Then we would like to use a static program analysis that will always successfully catch any possible runtime error, that will always say when a program is runtime error free, and that will never require any user input.

Unfortunately, this is, in general, impossible.

The halting problem is not computable. The canonical example of a semantic property for which no exact and fully automatic program analysis can be found is *termination*. Given a programming language, we cannot have a program analysis that, for any program in that language, correctly decides in finite time whether the program will terminate or not.

Indeed, it is well known that the halting problem is *not computable*. We explain more precisely the meaning of this statement. In the following, we consider a *Turing-complete* language, that is, a language that is as expressive as a Turing machine (common general-purpose programming languages all satisfy this condition), and we denote the set of all the programs in this language by $\mathbb{L}$. Second, given a program $\mathtt{p}$ in $\mathbb{L}$, we say that an execution

e terminates if it reaches the end of p after finitely many computation steps. Last, we say that a program terminates if and only if its executions terminate.

Theorem 1.1 (Halting problem) *The* halting problem *consists in finding an algorithm* `halt` *such that:*

For every program $\mathtt{p} \in \mathbb{L}$, $\mathtt{halt}(\mathtt{p}) =$ **true** *if and only if* p *terminates.*

The halting problem is not computable*: there is no such algorithm* `halt`*, as proved simultaneously by Alonso Church (?) and Alan Turing (?) in 1936.*

This means that termination is beyond the reach of a fully automatic and precise program analysis.

Interesting semantic properties are not computable. More generally, any *non-trivial semantic properties* are also not computable. By semantic property, we mean a property that can be completely defined with respect to the set of executions of a program (as opposed to a syntactic property, which can be decided directly based on the program text). We call a semantic property non-trivial when there exist programs that satisfy it and programs that do not satisfy it. Obviously only such properties are worth the effort of designing a program analysis for.

It is easy to see that a particular non-trivial semantic property is uncomputable (i.e., the property cannot have an exact decision procedure (analyzer)). Otherwise, the exact decision procedure solves the halting problem. For example, consider a property: this program prints 1 and finishes. Suppose there exists an analyzer that correctly decides the property for any input program. This analyzer solves the halting problem as follows. Given an input program *P*, the analyzer checks its slightly changed version "*P*; `print 1`." That the analyzer says "yes" means *P* stops, and "no" means *P* does not stop.

Indeed, Rice's theorem settles the case that any non-trivial semantic property is not computable:

Theorem 1.2 (Rice theorem) *Let* $\mathbb{L}$ *be a Turing-complete language and* $\mathscr{P}$ *be a non-trivial semantic property of programs of* $\mathbb{L}$*. There exists no algorithm such that*

For every program $\mathtt{p} \in \mathbb{L}$*, it returns* **true** *if and only if* p *satisfies the semantic property* $\mathscr{P}$*.*

As a consequence, we should also give up hope of finding an ideal program analysis that can determine fully automatically when a program satisfies any interesting property such as the absence of runtime errors, the absence of information flows, and functional correctness.

Towards computability. However, this does not mean that no useful program analysis can be designed. It only means that the analyses we are going to consider will all need to suffer some kind of limitation, by giving up on automation, by targeting only a restricted class of programs (i.e., by giving up the "*For every program*" in Theorem 1.2), or by not always being able to provide an exact answer (i.e., by giving up the "*if and only if*" in Theorem 1.2). We will discuss these possible compromises in the next sections.

1.3.4 Automation and Scalability

The first way around the limitation expressed in Rice's theorem is to give up on *automation* and to let program analyses require some amount of user input. In this case, the user is asked to provide some information to the analysis, such as global or local invariants (an invariant is a logical property that can be proved to be inductive for a given program). This means that the analysis is partly *manual* since the user needs to compute part of the results by herself.

Obviously, having to supply such information can often become quite cumbersome when programs are large or complex, which is the main drawback of manual methods.

Worse still, this process may be error prone so that a human error may ultimately lead to wrong results. To avoid such mistakes, program analysis tools may independently verify the user-supplied information. Then when the user supplied information is wrong, the analysis tool will simply reject it and produce an error message. When the analysis tool can complete the verification and check the validity of the user supplied information, the correctness of the final result will be guaranteed.

Even when a program analysis is automatic, it may not always produce a result within a reasonable time. Indeed, depending on the complexity of the algorithms, a program analysis tool may not be able to scale to large programs due to time costs or other resource constraints (such as memory usage). Thus, *scalability* is another important characteristic of a program analysis tool.

1.3.5 Approximation: Soundness and Completeness

Instead of giving up on automation, we can relax the conditions about program analysis by *letting it sometimes return inaccurate results*.

It is important to note that inaccurate *does not mean wrong*. Indeed, if the kind of inaccuracy is known, the user may still draw (possibly partly) conclusive results from the analysis output. For example, suppose we are interested in program termination. Given an input program to verify, the program analysis may answer "yes" or "no" only when it is fully sure about the answer. When the analysis is not sure, it will just return an undetermined result "don't know". Such an analysis would be still useful if the cases where it answers "don't know" are not too frequent.

In the following paragraphs, we introduce two dual forms of inaccuracies (or equivalently, approximations) that program analysis may make. To fix the notations, we assume a semantic property of interest $\mathscr{P}$ and an analysis tool `analysis`, to determine whether this property holds.

Ideally, if `analysis` were perfectly accurate, it would be such that

$$\text{For all program } \mathtt{p} \in \mathbb{L}, \qquad \mathtt{analysis}(\mathtt{p}) = \textbf{true} \quad \Longleftrightarrow \quad \mathtt{p} \text{ satisfies } \mathscr{P}.$$

This equivalence property can be decomposed into a pair of implications:

$$\begin{cases} \text{For all program } \mathtt{p} \in \mathbb{L}, & \mathtt{analysis(p)} = \textbf{true} & \Longrightarrow & \mathtt{p} \text{ satisfies } \mathscr{P} \\ \text{For all program } \mathtt{p} \in \mathbb{L}, & \mathtt{analysis(p)} = \textbf{true} & \Longleftarrow & \mathtt{p} \text{ satisfies } \mathscr{P} \end{cases}$$

Therefore, we can weaken the equivalence by simply dropping either of these two implications. In both cases, we get a partially accurate tool, that may either return a conclusive answer or a non conclusive one ("don't know").

We now discuss in detail both of these implications.

Soundness. A *sound* program analysis satisfies the first implication:

Definition 1.2 (Soundness) *The program analyzer* `analysis` *is* sound *with respect to property* $\mathscr{P}$ *whenever, for any program* $\mathtt{p} \in \mathbb{L}$, $\mathtt{analysis(p)} = \textbf{true}$ *implies that* `p` *satisfies property* $\mathscr{P}$.

When a sound analysis (or analyzer) claims that the program has property $\mathscr{P}$, it guarantees that the input program indeed satisfies the property. We call such an analysis *sound* as it always errs on the side of caution: it will not claim a program satisfies $\mathscr{P}$, unless this property can be guaranteed. In other words, a sound analysis will reject all programs that do not satisfy $\mathscr{P}$.

Example 1.3 (Strong typing) *A classic example is that of strong typing that is used in program languages such as ML, that is based on the principle that "well-typed programs do not go wrong": indeed, well-typed programs will not present certain classes of errors whereas certain programs that will never crash may still be rejected.*

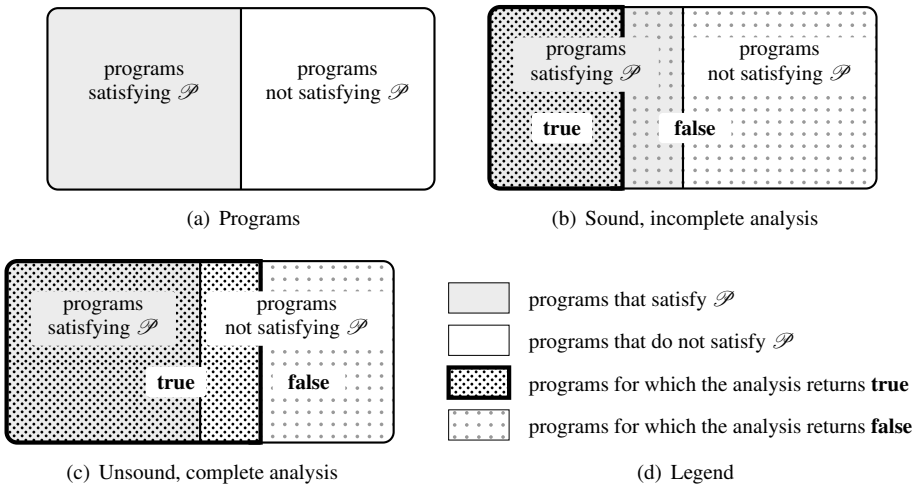
From a logical point of view, the soundness objective is very easy to meet since the trivial `analysis` defined to always return **false** obviously satisfies Definition 1.2. Indeed, this trivial analysis will simply reject any program. This analysis is not useful since it will never produce a conclusive answer. Therefore, in practice, the design of a sound analysis will try to give a conclusive answer as often as possible. This is in practice possible. As an example, the case of an ML program that cannot be typed (i.e., is rejected) although there exists no execution that crashes due to a typing error is rare in practice.

Completeness. A *complete* program analysis satisfies the second, opposite implication:

Definition 1.3 (Completeness) *The program analyzer* `analysis` *is* complete *with respect to property* $\mathscr{P}$ *whenever, for every program* $\mathtt{p} \in \mathbb{L}$, *such that* `p` *satisfies* $\mathscr{P}$, $\mathtt{analysis(p)} = \textbf{true}$.

A complete program analysis will accept every program that satisfies property $\mathscr{P}$. We call such an analysis *complete* because it does not miss a program that has the property. In other words, when a complete analysis rejects an input program, the completeness guarantees that the program indeed fails to satisfy $\mathscr{P}$.

Example 1.4 (User assertions) *The error search technique based on user assertions is complete in the sense of Definition 1.3. User assertions let developers improve the quality of their software thanks to runtime checks inserted as conditions in the source code, and that are checked during program executions. This practice can be seen as a very rudimentary form of verification for a*



(a) Programs (b) Sound, incomplete analysis

(c) Unsound, complete analysis (d) Legend

Figure 1.2
Soundness and completeness with Venn diagrams

limited class of safety properties, where a given condition should never be violated. Faults are reported during program executions, as assertion failures. If an assertion fails, then it means that at least one execution will produce a state where the assertion condition is violated.

As in the case of soundness, it is very easy to provide a trivial but useless complete analysis. Indeed, if `analysis` always returns **true**, then it never rejects a program that satisfies the property of interest; thus, it is complete, though it is of course of no use. To be useful, a complete analyzer should often reject programs that do not satisfy the property of interest. Building such useful complete analyses is a difficult task in general (just like it is also difficult to build useful sound analyses).

Soundness and completeness. Soundness and completeness are two dual properties. To better show them, we represent answers of sound and complete analyses using Venn diagrams in Figure 1.2 and following the legend in Figure 1.2(d):

- In Figure 1.2(a), we draw the set of all programs and divide it into two subsets, namely the programs that satisfy the semantic property $\mathscr{P}$ and the programs that do not satisfy $\mathscr{P}$. A sound and complete analysis would always return **true** exactly for the programs that are in the left part of the diagram.
- Figure 1.2(b) depicts the answers of an analysis that is *sound* but *incomplete*: we can see that it rejects all programs that do not satisfy the property but also rejects some that do satisfy it; whenever it returns **true**, we have the guarantee that the analyzed program satisfies $\mathscr{P}$.
- Figure 1.2(c) depicts the answers of an analysis that is *complete* but *unsound*: we can see that it accepts all programs that do satisfy the property but also accepts some that

do not satisfy it; whenever it returns **false**, we have the guarantee that the analyzed program does not satisfy $\mathscr{P}$.

Due to the computability barrier, we should not hope for a sound, complete and fully automatic analysis when trying to determine which programs satisfy any non-trivial execution property for a Turing complete language. In other words, when a program analysis is automatic, it is either unsound or incomplete. However, this does not mean it is impossible to design a program analysis that returns very accurate (sound and complete) results on an interested sets of input programs. However, even in that case, there will always exist input programs for which the analysis will return inaccurate (unsound or incomplete) results.

In the previous paragraphs, we have implicitly assumed that the program analysis tool `analysis` always terminates and never crashes. In general, non-termination or crashes of `analysis` should be interpreted conservatively. For instance, if `analysis` is meant to be sound, then its answer should be conservatively considered negative (**false**) whenever it does not return **true** within the allocated time bounds.

1.4 Families of Program Analysis Techniques

In this section, we describe several families of approaches to program analysis. Due to the negative result presented in Section 1.3.3, no technique can achieve a fully automatic, sound and complete computation of a non-trivial property of programs. We will show the characteristics of each of these techniques using the definitions of Section 1.3.

1.4.1 Testing: Checking a Set of Finite Executions

When trying to understand how a system behaves, the first idea that often comes to mind is to observe the executions of this system. In the case of a program that may not terminate and may have infinitely many executions, it is of course not feasible to fully observe all executions.

Therefore, the *testing* approach observes only a finite set of finite program executions. This technique is used by all programmers, from beginners to large teams designing complex computer systems. In industry, many levels of testing are performed at all stages of development such as unit testing (execution of sample runs on a basic function) and integration testing (execution of large series of tests on a completed system, including hardware and software).

Basic testing approaches, such as random testing (**?**) typically provide a low coverage of the tested code. However, more advanced techniques improve coverage. As an example *concolic testing* (**?**) combines testing with symbolic execution (computation of exact relations between input and output variables on a single control flow path) so as to improve coverage and accuracy.

Testing has the following characteristics:

- It is in general easy to automate, and many techniques (such as concolic testing) have been developed to synthesize useful sets of input data to maximize various measures of coverage;
- In almost all cases, it is unsound, except in the cases of programs that have only a finite number of finite executions (though it is usually prohibitively costly in that case);
- It is complete since a failed testing run will produce an execution that is incorrect with respect to the property of interest (such a counter-example is very useful in practice since it shows programmers exactly how the property of interest may be violated and often gives precise information on how to fix the program).

Besides, testing is often costly, and it is hard to achieve a very high path coverage on very large programs. On the other hand, a great advantage of testing is that it can be applied to a program in the conditions in which it is supposed to be run: for instance, testing a program on the target hardware, with the target operating system and drivers, may help diagnose issues that are specific to this combination.

When the semantics of programs is non-deterministic, it may not be feasible to reproduce an execution, which makes the exploitation of the results produced by testing problematic. As an example, the execution of a set of concurrent tasks depends on the scheduling strategy so that two runs with the same input may produce different results, if this strategy is not fully deterministic.

Another consideration is that testing will not allow attacking certain classes of properties. For instance, it will not allow proving that a program terminates, even over a finite set of inputs.

1.4.2 Assisted Proof: Relying on User Supplied Invariants

A second way to avoid the limitation shown in Section 1.3.3 consists in giving up on automation.

This is essentially the approach followed by *machine assisted* techniques. This means that users may be required to supply additional information together with the program to analyze. In most cases, the information that needs to be supplied is comprised of loop invariants and possibly some other intermediate invariants. This often requires some level of expertise. On the other hand, a large part of the verification can generally still be carried out in a fully automatic way.

We can cite several kinds of program analyses based on machine assisted techniques. A first approach is based on theorem proving tools like Coq (**?**), Isabelle/HOL (**?**) and PVS (**?**), and requires the user to formalize the semantics of programs and the properties of interest and to write down proof scripts, which are then checked by the prover. This approach is adapted to the proof of sophisticated program properties. It was applied to the verified CompCert compiler (**?**) from C to Power-PC assembly (the compiler is verified in the sense that it comes with a proof that it will compile any valid C program properly). It was also

used for the design of the micro-kernel SEL4 verified (**?**). A second approach leverages a tool infrastructure to prove a specific set of properties over programs in a specific language. The B-method (**?**) toolset implements such an approach. Also, tools such as the Why C program verification framework (**?**) or Dafny (**?**) input a program with a property to verify and attempt to prove the property using automatic decision procedures, while relying on the user for the main program invariants (such as loop invariants) and when the automatic procedures fail.

Machine-assisted techniques have the following characteristics:

- They are not fully automatic and often require the most tedious logical arguments to come from the human user;
- In practice, they are sound with respect to the model of the program semantics used for the proof, and they are also complete up to the abilities of the proof assistant to verify proofs (the expressiveness of the logics of the proof assistant may prevent some programs to be proved, though this is rarely a problem in practice).

In practice, the main limitation of machine-assisted techniques is the significant resources they require, in terms of time and expertise.

1.4.3 Model Checking: Exhaustive Exploration of Finite Systems

Another approach focuses on finite systems, that is, systems whose behaviors can be exhaustively enumerated, so as to determine whether all executions satisfy the property of interest. This approach is called *finite state model checking* (**???**) since it will check a model of a program using some kind of exhaustive enumeration. In practice, model checking tools use efficient data-structures to represent program behaviors and avoid enumerating all executions thanks to strategies that allow reducing the search space.

Note that this solution is very different from the testing approach discussed in Section 1.4.1. Indeed, testing samples a finite set of behaviors among a generally infinite set, whereas model checking attempts to check all executions of a finite system.

The finite model checking approach has been used both in hardware verification and in software verification.

Model checking has the following characteristics:

- It is automatic;
- It is sound and complete *with respect to the model*.

An important caveat is that the verification is performed at the model level and not at the program level. As a first consequence, this means that a model of the program needs to be constructed, either manually or by some automatic means. In practice, most model checking tools provide a frontend for that purpose. A second consequence is that the relation between this model and the input program should be taken into account when assessing the results; indeed, if the model cannot capture exactly the behaviors of the program (which is likely as programs are usually infinite systems since executions may be of arbi-

trary length), the checking of the synthesized model may be either incomplete or unsound, with respect to the input program. Some model checking techniques are able to automatically refine the model when they realize that they fail to prove a property due to a spurious counter-example; however, the iterations of the model-checking and refinement may continue indefinitely, so some kind of mechanism is required to guarantee termination. In practice, model-checking tools are often conservative and are thus sound and incomplete with respect to the input program. A large number of model checking tools have been developed for verifying different kinds of logical assertions on various models or programming languages. As an example, UPPAAL (**?**) verifies temporal logic formulas on timed automata.

1.4.4 Conservative Static Analysis: Automatic, Sound and Incomplete Approach.

Instead of constructing a finite model of programs, *static analysis* relies on other techniques to compute conservative descriptions of program behaviors using finite resources. The core idea is to finitely over-approximate the set of all program behaviors using a specific sets of properties, the computation of which can be automated (**??**). A (very simple) example is the type inference present in many modern programming languages such as variants of ML. Types (**??**) provide a coarse view of what a function does, but still does so in a very effective manner, since the correctness of type systems guarantees that a function of type `int -> bool` will always input an integer and return a boolean (when it terminates). Another contrived example is the removal of array bound checks by some compilers for optimization purposes, using numerical properties over program variables that are automatically inferred at compile-time. The next chapters will generalize this intuition and introduce many other forms of static analyses.

Besides compilers, static analysis has been very heavily used in order to design program verifiers and program understanding tools for all sorts of programming languages. Among many others, we can cite the ASTRÉE (**?**) static analyzer for proving the absence of runtime errors in embedded C codes, the Facebook INFER (**?**) static analyzer for the detection of memory issues in C/C++/Java programs, the JULIA (**?**) static analyzer for discovering security issues in Java programs, the POLYSPACE (**?**) static analyzer for ADA/C/C++ programs and the SPARROW (**?**) static analyzer for the detection of memory errors in C programs.

Static analysis approaches have the following characteristics:

- They are automatic;
- They produce sound results, as they compute a *conservative* description of program behaviors, using a limited set of logical properties. Thus, they will never claim the analyzed program satisfies the property of interest when it is not true;
- They are generally incomplete because they cannot represent all program properties and rely on algorithms that enforce termination of the analysis even when the input

program may have infinite executions. As a consequence, they may fail to prove correct some programs that satisfy the property of interest.

Static analysis tools generally input the source code of programs and do not require modeling the source code using an external tool. Instead, they directly compute properties taken in a fixed set of logical formulas, using algorithms that we present throughout the following chapters.

While a static analysis is incomplete in general, it is often possible to design a sound static analysis that gives the best possible answer on classes of interesting input programs, as discussed in Section 1.3.5. However, it is then always possible to craft a correct input program for which the analysis will fail to return a conclusive result.

Last, we remark that it is entirely possible to drop soundness so as to preserve automation and completeness. This leads to a different kind of analysis that produces an under-approximation of the programs actual behaviors, and allows to answer a very different kind of question. Indeed such an approach may guarantee that a given subset of the executions of the program can be observed. For instance, this may be useful to establish that this program has at least one successful execution. On the other hand, it does not allow to prove properties such as the absence of runtime errors.

1.4.5 Bug Finding: Error Search, Automatic, Unsound, Incomplete, Based on Heuristics

Some automatic program analysis tools sacrifice not only completeness but also soundness. The main motivation to do so is to simplify the design and implementation of analysis tools and to provide lighter weight verification algorithms. The techniques used in such tools are often similar to those used in model checking or static analysis, but relax the soundness objective. For instance, they may construct unsound finite models of programs so as to quickly enumerate a subset of the executions of the analyzed program (e.g., by only considering what happens in the first iteration of each loop (**?**)), whereas a sound tool would have to consider possibly unbounded iteration numbers. As an example, the commercial tool COVERITY (**?**) applies such techniques to programs written in a wide range of languages (e.g., Java, C/C++, JavaScript, Python...). Similarly, the tool CODESONAR (**?**) relies on such approaches so as to search for defects in C/C++ or Assembly programs. The CBMC tool (C Bounded Model Checker) (**?**) extracts models from C/C++ or Java programs and performs bounded model checking on them, which means that it explores models only up to a fixed depth. It is thus a case of a model checker that gives up on soundness in order to produce fewer alarms.

Since the main motivation of this approach is to discover bugs (and not to prove their absence), it is often referred to as *bug finding*.

Such tools are usually applied to improve the quality of non-critical programs at a low cost.

Bug-finding tools have the following characteristics:

	automatic	sound	complete	object	when
testing	yes	no	yes	program	dynamic
assisted proving	no	yes	yes/no	model	static
model checking of finite state model	yes	yes	yes	finite model	static
model checking, at program level	yes	yes	no	program	static
conservative static analysis	yes	yes	no	program	static
bug finding	yes	no	no	program	static

Figure 1.3
An overview of program analysis techniques

- They are automatic;
- They are neither sound nor complete; instead, they aim at discovering bugs rather quickly, so as to help developers.

1.4.6 A Summary

The table shown in Figure 1.3 summarizes the techniques for program analysis that we have introduced in this chapter and compare them based on five criteria. As we can see, due to the computability barrier, no technique can provide fully automatic, sound and complete analyses. Testing sacrifices the soundness. Assisted proving is not automatic (even if it is often partly automated, the main proof arguments generally need to be human-provided). Model-checking approaches can achieve soundness and completeness only with respect to finite models, and they generally give up completeness when considering programs (the incompleteness is often introduced in the modeling stage). Static analysis gives up completeness (though they may be designed to be precise for large classes of interested programs). Last, bug finding is neither sound nor complete.

As we remarked earlier, another important dimension is *scalability*. In practice, all approaches have limitations regarding scalability, although these limitations vary depending on the intended applications (e.g., input programs, target properties, algorithms used).

1.5 Roadmap

From now on, we will focus on *conservative static analysis*, from its design methodologies to its implementation techniques.

Definition 1.4 (Static Analysis) Static analysis *is an automatic technique for program-level analysis that approximates in a conservative manner semantic properties of programs before their executions.*

After a gentle introduction to static analysis in Chapter 2, we present a static analysis framework based on a compositional semantics in Chapter 3, a static analysis framework based on a transitional semantics in Chapter 4, and some advanced techniques in Chapter 5. These frameworks, thanks to a semantics-based viewpoint, are general so that they can guide the design of conservative static analyses for any programming language and for any semantic property. In Chapter 6, we present issues and techniques regarding the use of static analysis in practice. In Chapter 7, we discuss and demonstrate the implementation techniques to build a static analysis tool. In Chapter 8, we present how we use the general static analysis framework to analyze seemingly-complex features of realistic programming languages. In Chapter 9, we discuss several important families of semantic properties of interest and show how to cope with them using static analysis. In Chapter 10, we present several specialized, yet high-level frameworks for specific target languages and semantic properties. In Chapter 11, we summarize this book.

2 A Gentle Introduction to Static Analysis

Goal of this chapter. In this chapter, we provide an introduction to static analysis that does not require any background. This introduction aims at making the core concepts of static analysis intuitive and crisp. To this end, we define a basic programming language that describes sequences of transformations applied to points in a two-dimensional space [1]. The notions presented here extend to realistic programming languages. We will formalize them thoroughly in Chapter 3 in the case of a basic imperative language.

Recommended reading: [S], [D], [U].
We recommend this chapter to all readers, as it introduces the basic intuitions useful to understand many more advanced static analysis techniques and the way static analysis tools work. Understanding these concepts is important not only to design or implement a static analyzer but also to use it as well as possible.

2.1 Semantics and Analysis Goal: a Reachability Problem

Syntax of a very basic language. For the sake of making our first introduction to static analysis intuitive, we consider a very basic language with intuitive notions of states and executions. The language under study is inspired by drawing languages used for educational purposes such as introducing children to programming.

A *state* describes the configuration of a computer running a program, observed at a given instant. In general, this includes a description of the memory contents, the registers, and the program counter. In this chapter, a state will simply denote a point in the two-dimensional space, described by its real coordinates $(\mathtt{x}, \mathtt{y})$. We denote the set of such states by $\mathscr{S}$.

[1] This chapter has been partially inspired by graphical descriptions of the notion of abstraction such as the one presented in `http://web.mit.edu/16.399/www/lecture_13-abstraction1/Cousot_MIT_2005_Course_13_4-1.pdf`, even though this chapter focuses on different aspects of static analysis, transfer functions and abstract iterations.

We let programs define combinations of basic geometric operations. Basic operations comprise:

- initialization with a point that is non-deterministically chosen in a fixed region $\mathfrak{R}$ (for instance, the $[0,1] \times [0,1]$ square or any other geometrical shape specified by a set of points);
- geometrical translations (specified by a vector);
- geometrical rotations (specified by a center and an angle in degrees).

Moreover, a program is defined either as a sequence of operations or as a non-deterministic choice of two sequences of operations, or as a non-deterministic iteration of a sequence of operations (the number of iterations is chosen non-deterministically). To avoid starting from an undefined state, we assume that a program always begins with an initialization, which fixes the set of initial states.

The syntax of programs is defined by the grammar below (note that we only consider programs that start with an initialization statement even though this grammar does not express that constraint):

p	::=	**init**$(\mathfrak{R})$	initialization, with a state in $\mathfrak{R}$
	|	**translation**(u,v)	translation by vector (u,v)
	|	**rotation**(u,v,θ)	rotation defined by center (u,v) and angle θ
	|	p ; p	sequence of operations
	|	{p}**or**{p}	choice (the branch taken is non-deterministic)
	|	**iter**{p}	iteration (the number of iterations is non-deterministic)

Semantics. As observed in Chapter 1, static analysis aims at computing semantic properties of programs. Therefore, before we look into the definition of a static analysis, we need to define the *semantics* of programs, which should characterize the program executions. A common way to achieve this is to just let the semantics be the set of all the program executions. Such a semantics is often called *collecting semantics*.

An *execution* provides a complete view of a single run of the program. Since we assume that a program makes discrete computation steps at every clock tick, it is naturally described by a sequence of states. Each of the basic program constructions in the above grammer is quite simple so we do not formalize them fully. Intuitively, their semantics is defined as follows:

- the initialization operation simply produces a state in a given region;
- translation and rotation transformations induce basic execution steps, which perform the corresponding geometric transformations;
- sequences of operations yield sequences of execution steps;
- non-deterministic choices and iterations respectively select or repeat a block of code and construct executions that can be derived from those of the subprograms.

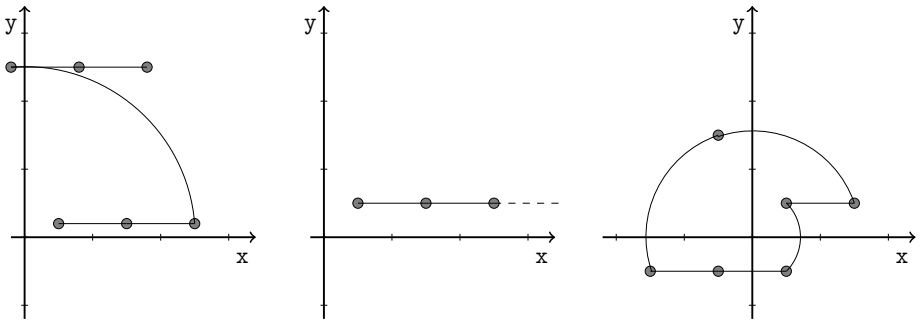


Figure 2.1
A few program executions

Example 2.1 (Semantics) *To make this semantics more intuitive, we consider the program below:*

```
init([0,1] × [0,1]);
translation(1,0);
iter{
    {
        translation(1,0)
    }or{
        rotation(0,0,90°)
    }
}
```

This program starts in the $[0,1] \times [0,1]$ *square, performs a translation and then performs a number of translations or rotations that are chosen non-deterministically (i.e., an oracle decides at run-time both the number of operations and their nature). Figure 2.1 shows three executions:*

- *in the first execution (shown in the left), the program starts from* $(0.5, 0.2)$*, performs two translations, one rotation, two translations, and then terminates;*
- *in the second execution (shown in the middle), the program starts at point* $(0.5, 0.5)$ *and repeats the same transition forever;*
- *in the third execution (shown in the right), the program starts at point* $(0.5, 0.5)$ *and then repeats forever the sequence made of one translation, two rotations, two translations, and one rotation.*

While the first execution is finite, the other two are actually infinite (which means that the program runs forever).

Semantic property of interest: reachability. It is now time to set the property of interest that we are going to consider in this chapter.

We aim for a *reachability* property that is specified by a zone made of points that should not be reached by any execution of the program. Intuitively, we assume that a set of points is fixed and defines a zone that we expect program executions to *never* reach. In other words, if any execution reaches this zone, we would consider it as an error. In the

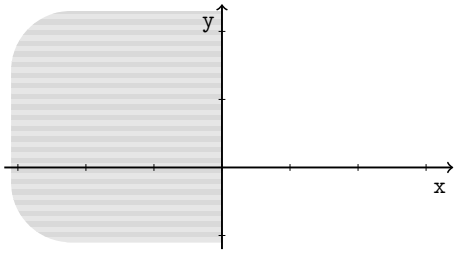


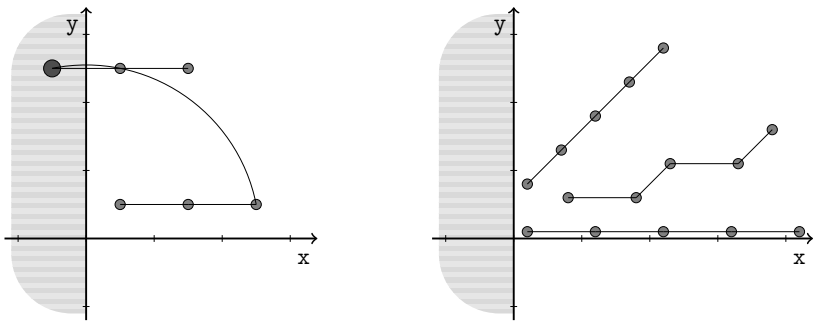
Figure 2.2
Region supposed to be unreachable: points with a negative x-coordinate

following, we search for a static analysis that is able to catch and reject any program with such an erroneous execution. When a program has no such offending execution, the analysis should, as often as possible, accept the program and issue a proof of correctness.

This semantic property is a very classic example of safety property (Section 1.3.1) (although not all safety properties are of that form). Very often, programmers would like to ensure similar properties in real programs. As an example, reaching a state where a C program will dereference a null pointer will produce an abrupt run-time error. Similarly, if a C program reaches a state where it writes over a dangling pointer, either the execution will fail abruptly, or some data will be corrupted. For these reasons, C programmers are usually interested in checking that their programs will never reach a state where they would dereference an invalid, null or dangling pointer. Thus, the reachability property that we study here is actually quite realistic, even though we are looking at a contrived language.

In the rest of this chapter, we will often use the region $\mathfrak{D}$ defined by $\mathfrak{D} = \{(\mathtt{x}, \mathtt{y}) \mid \mathtt{x} < 0\}$ to denote the set of states that program executions should never reach, although we will construct a program analysis technique that would work for other regions as well. This "error zone" is depicted in Figure 2.2. We let $\neg\mathfrak{D}$ denote the property that we would like to verify since it expresses that all the states a program may reach are *not* in $\mathfrak{D}$. To give more intuition, we study a couple of programs.

Example 2.2 (Reachability and incorrect executions) *First, we consider the program of Example 2.1. Obviously, it violates the property $\neg\mathfrak{D}$ since Figure 2.1 shows two executions that eventually reach a point $(\mathtt{x}, \mathtt{y})$, where $\mathtt{x} < 0$. As an example, Figure 2.3(a) displays an execution of the program studied in Example 2.1, which is incorrect as it reaches the error zone after three steps.*



(a) An incorrect execution (b) Correct executions

Figure 2.3
Reachability and programs

Example 2.3 (Reachability and program with only correct executions) *In this example, we study a second program:*

```
init([0,1] × [0,1]);
iter{
    {
        translation(1,0);
    }or{
        translation(0.5,0.5);
    }
}
```

Figure 2.3(b) displays a few executions of this program, and we observe they are all correct, in the sense that they never enter the error zone $\mathfrak{D}$. In fact, we can informally show that all executions of this program will stay in the safe zone $\neg\mathfrak{D}$ at all times:

- *they start at a point $(\mathtt{x},\mathtt{y})$ such that $0 \leq \mathtt{x} \leq 1$ which thus satisfies $\neg\mathfrak{D}$;*
- *during a loop iteration, $\mathtt{x}$ is increased by either 1 or 0.5 depending on the result of a non-deterministic choice; thus, the coordinate $\mathtt{x}$ remains non-negative.*

Static analysis for reachability. In the rest of this chapter, we define a static analysis (actually, a family of static analyses) that attempts to determine whether an input program satisfies the semantic property $\neg\mathfrak{D}$. The analysis should always return a sound result: if it returns **true** when applied to an input program p, we expect to have the guarantee that no execution of p will ever reach $\mathfrak{D}$. Therefore, a program such as that of Example 2.1 will be flagged as "possibly violating the property of interest." Ideally, we would also like

the analysis to be precise enough so that it can conclusively report that the program of Example 2.3 is correct.

An obvious way would be to enumerate all executions of the input program so as to determine all reachable configurations. But this would not be feasible, as even simple programs (such as those presented in Example 2.1 and Example 2.3) have infinitely many executions since the set of initial states is infinite, the length of executions is infinite, and the set of possible series of non-deterministic choices is infinite.

Therefore, we will seek other ways to determine the set of all reachable configurations.

2.2 Abstraction

Core principle of abstraction. In this section, we search for a way to reason about program executions that will produce a superset of the reachable states and that should be rather simple to compute.

To choose this superset, we first try to draw some intuition from the program studied in Example 2.3. In that example, we noticed that no execution reaches the region $\mathfrak{D}$, and we gave an informal proof of this fact:

- in the beginning, the x-coordinate is non-negative;
- at each step it may only grow which means that, if it is non-negative, it will remain so.

In the following, we will aim at making such reasoning steps automatic. We remark that the steps of this proof do not use all the information present in program states:

- the value of the y-coordinate is completely ignored;
- only the sign of the x-coordinate is considered in the proof (or more precisely the fact that the value of x may be either positive, zero, or negative).

This intuition forms the basis of *abstraction* (**?**): by retaining only rather coarse information about program states and considering how the program runs, we can still infer interesting information about the set of all program executions (in this case, that $\mathtt{x} \geq 0$ at all times). Such information is captured by a set of logical properties that the analysis may manipulate, using algorithms exposed in the next section. In the above example, the only logical properties that seem to matter in the proof are $\mathtt{x} \geq 0$ (used in the case of the program of Example 2.3) and the property "**true**," which is satisfied by any state (that is needed to describe the states the program of Example 2.1 may reach).

There are of course many possible sets of logical properties that could be used in such proofs. Thus, we need to make clear what logical properties the analysis may manipulate.

Definition 2.1 (Abstraction) *We call* abstraction *a set $\mathscr{A}$ of logical properties of program states, that are called* abstract properties *or* abstract elements. *A set of abstract properties is called an* abstract domain.

In this definition, the word *abstract* is used here as opposed to the word *concrete*: in the following, we use the "concrete" qualifier to denote actual program behaviors, whereas the

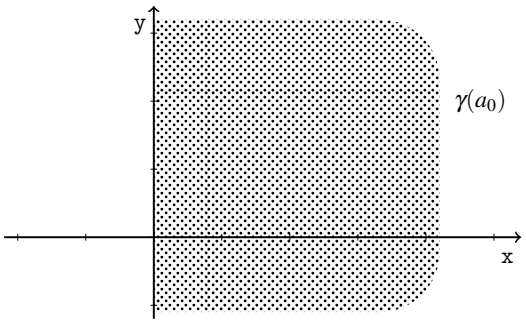


Figure 2.4
Abstraction based on the sign of the x component

"abstract" qualifier applies to the properties used in the (automatic) proofs. As an example, the *concrete semantics* is the actual semantics of programs as defined in Section 2.1. By contrast, an *abstract semantics* shall define a computable over-approximation of the concrete semantics expressed in terms of abstract states (actually, the goal of static analysis is precisely to compute such a sound abstract semantics).

The mathematical and computer representation of abstract elements is crucial for the definition of all the static analysis algorithms that we are going to consider. Ideally, the abstract properties should come with an efficient computer representation and with analysis algorithms (the algorithms are discussed further in this chapter) since we intend to develop a static analyzer that relies on these predicates. Thus, it is important to distinguish the abstract elements from their meaning:

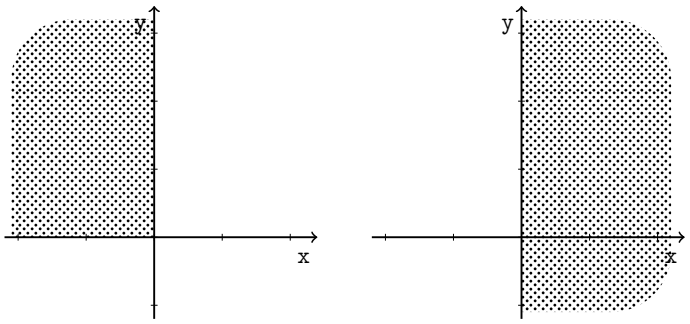
Definition 2.2 (Concretization) *Given an abstract element a of $\mathscr{A}$, we call* concretization *the set of program states that satisfy it. We denote it by $\gamma(a)$.*

Example 2.4 (Abstraction by the sign of the x-coordinate) *As a first example, we present the abstraction used above in order to informally demonstrate that the program of Example 2.3 never reaches the region $\mathfrak{D}$. This abstraction has two elements a_0, a_1, where*

- *a_0 denotes all the states $(\mathtt{x},\mathtt{y})$ such that $\mathtt{x} \geq 0$ ($\gamma(a_0)$ is the infinite half-plane area that is filled with dots in Figure 2.4);*
- *a_1 denotes the set of all the states such that $\gamma(a_1) = \mathscr{S}$ (the whole two-dimensional space).*

In Figure 2.4 (and subsequent pictures that depict abstract elements), we represent the points described by an abstract element as a zone filled with dots. This region is syntactically different from the abstract element itself: the latter is the representation of the former, and the analysis manipulates only the representation. Even though we will often focus less on this distinction in this chapter (and sometimes implicitly assimilate abstract elements and the regions they denote), it will play a great role in subsequent chapters.

There exist of course many possible choices of abstractions, which are less contrived than the one of Example 2.4. Some abstractions describe more expressive sets of logical



(a) Concretization of $[\mathtt{x} \leq 0, \mathtt{y} \geq 0]$ (b) Concretization of $[\mathtt{x} \geq 0]$

Figure 2.5
Signs abstraction

properties than others. Furthermore, some abstractions yield simpler computer representations and less costly algorithms than others. In the following paragraphs, we will present a few other examples of abstractions, which also have a simple and intuitive graphical representation.

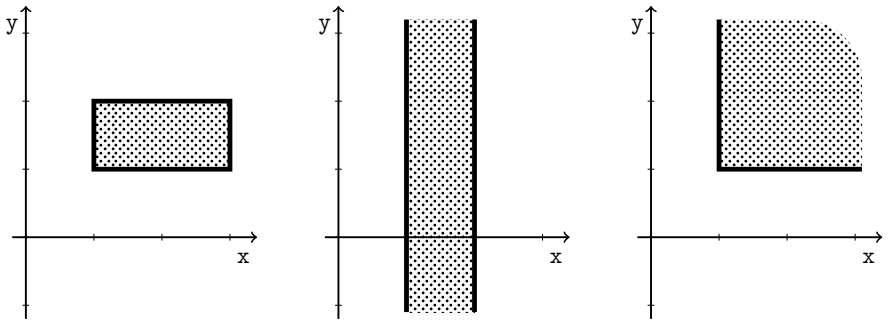
Signs abstraction. The abstraction of Example 2.4 treats x and y differently and is very specific to the property $\mathfrak{D}$ defined in Section 2.1. It would not work if we wanted a static analysis to prove that y never becomes negative. Similarly, it would not apply if the property to prove was that x does not become positive. However, this abstraction generalizes into a more expressive one, which describes a set of states using two pieces of information: the possible values of the sign of x *and* the possible values of the sign of y. For each variable, this abstraction records whether it may be positive, negative, non-negative, etc. The concretizations of a few abstract elements are shown in Figure 2.5:

- the left diagram shows the concretization of the abstract element that expresses the fact that x is negative, and y is non-negative;
- the right diagram shows the concretization of the abstract element that expresses the fact that x is positive and this abstract element carries no information about y.

We can observe that this signs abstract domain can express any property the previous domain could express, but it can also describe some properties that were beyond the reach of the previous domain.

Intervals abstraction. In practice, abstractions based on signs are often too weak to capture strong program properties, but other more precise abstractions have been proposed.

Using inequalities and range constraints over variables is a very natural approach to reason over numerical properties. Similarly, we can use range constraints over program variables so as to more precisely describe what values they may take. This is the principle of the *intervals abstraction* (**?**):



(a) Concretization of a_0 (b) Concretization of a_1 (c) Concretization of a_2

Figure 2.6
Intervals abstraction

Definition 2.3 (Intervals abstraction) *The abstract elements of the interval abstraction are defined by constraints of the form* $l_{\mathtt{x}} \leq \mathtt{x}$, $\mathtt{x} \leq h_{\mathtt{x}}$, $l_{\mathtt{y}} \leq \mathtt{y}$, *and* $\mathtt{y} \leq h_{\mathtt{y}}$.

An abstract element is thus composed of at most four finite bound constraints. We remark that such an abstract element may denote the empty set of points, since some sets of constraints cannot be satisfied, such as $1 \leq \mathtt{x}, \mathtt{x} \leq 0$. We do not write down infinite bound constraints (cases where no lower and/or upper bound is given for a given variable).

Intuitively, interval abstract domain elements correspond to rectangles in the two-dimensional space, the sides of which are parallel to the axes.

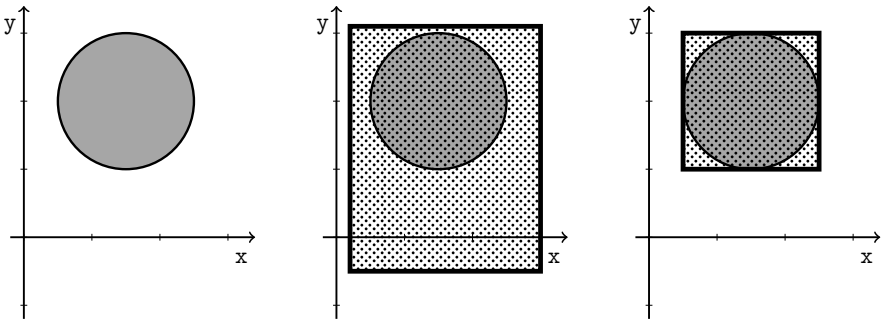
Example 2.5 (Intervals abstraction) *The following three abstract elements illustrate the kind of constraints that can be expressed by the intervals abstract domain, together with their concretizations, shown in Figure 2.6:*

- a_0 *corresponds to numerical constraints* $1 \leq \mathtt{x} \leq 3$ *and* $1 \leq \mathtt{y} \leq 2$ *(the concretization of* a_0 *is shown in Figure 2.6(a));*
- a_1 *corresponds to numerical constraints* $1 \leq \mathtt{x} \leq 2$ *(the concretization of* a_1 *is shown in the middle Figure 2.6(b));*
- a_2 *corresponds to numerical constraints* $1 \leq \mathtt{x}$ *and* $1 \leq \mathtt{y}$ *(the concretization of* a_2 *is shown in the right Figure 2.6(c)).*

Obviously, the intervals abstract domain is more expressive than the signs abstract domain. Indeed, any abstract element of the signs abstract domain also corresponds to an element in the intervals abstract domain.

The representation of an abstract element of the intervals domain boils down to at most two numerical constants per variable. Thus, this domain over-approximates a set of points in the two-dimensional space with at most four numerical constants, which take very little space in memory during the analysis.

We can introduce at this stage the concept of *best abstraction*. Given any set of points (which correspond to program states), we would like to define an abstract element in the in-



(a) A concrete set (b) Abstractions (c) Best abstraction

Figure 2.7
Best abstraction

tervals abstract domain that over-approximates our initial set. For instance, let us consider the set of program states defined by the disc shown in Figure 2.7(a). Then any box that encloses the disc is a valid over-approximation of this set: indeed, any such box describes all the points in the disc (and more), so it provides a conservative approximation of the disc. However, there exist many such enclosing boxes. Yet, some of these abstractions are more desirable than others. As we mentioned earlier, the goal of abstraction is to account for the concrete set of points using a simple description, at the cost of adding some additional points that are not in the concrete set. Adding fewer points that are not in the concrete set is better since it means the abstraction characterizes the set of points in a less ambiguous and more informative way. In the case of the intervals abstract domain, we can actually solve this problem in an elegant manner; indeed, the *smallest* rectangle that encloses any non-empty set of points is well-defined, using the greatest lower bounds and least upper bounds over both coordinates (the case of the empty set of points is trivial, as the empty rectangle is also an element of the abstract domain). In particular, Figure 2.7(c) shows the best approximation of the disc.

More generally, the best abstraction (**?**) is defined as a function that interprets any set of concrete points into an *optimal* abstract element:

Definition 2.4 (Best abstraction) *We say that a is the* best abstraction *of the concrete set S if and only if $S \subseteq \gamma(a)$ and for any a' that is an abstraction of S (i.e., $S \subseteq \gamma(a')$): then a' is a coarser abstraction than a. If S has a best abstraction, then the best abstraction is unique. When it is defined, we let α denote the function that maps any concrete set of states into the best abstraction of that set of states.*

As observed above, the intervals abstract domain has a best abstraction function. While computing a precise abstraction (if possible the best abstraction) is preferable in general, we will often encounter useful analyses that cannot compute the best abstraction, or such that the best abstraction cannot even be defined in the sense of Definition 2.4. The im-

possibility to define or compute the best abstraction is in no way a serious flaw for the analysis, as it will only cause it to err on the side of caution (i.e., to return conservative but sound results). Using an over-approximating abstract element is fine, though we prefer to compute the most tightly encompassing one, if it exists.

Finally, we remark that interval constraints cannot capture in a precise manner any complex numerical constraint over both x and y. For instance, it cannot express in an exact manner the property that x is smaller than y. We thus call it a *non-relational abstraction*. Intuitively, an abstract state characterizes each variable by an interval independently from the other variables. On one hand, this simplifies the shape of abstract elements and their representation; on the other hand, it limits the expressiveness of the abstraction.

Convex polyhedra abstraction. The obvious way to overcome the limitation inherent in the non-relational abstraction is to extend the abstract domain with relational constraints. Augmenting the abstract domain with all linear constraints allows achieving that:

Definition 2.5 (Convex polyhedra abstraction) *The abstract elements of the convex polyhedra abstract domain (?) are conjunctions of linear inequality constraints.*

This abstract domain can describe precisely any concrete set that can be described by the signs and intervals abstract domains. It can also describe many other sets of concrete points in a much more precise way than the previous abstractions.

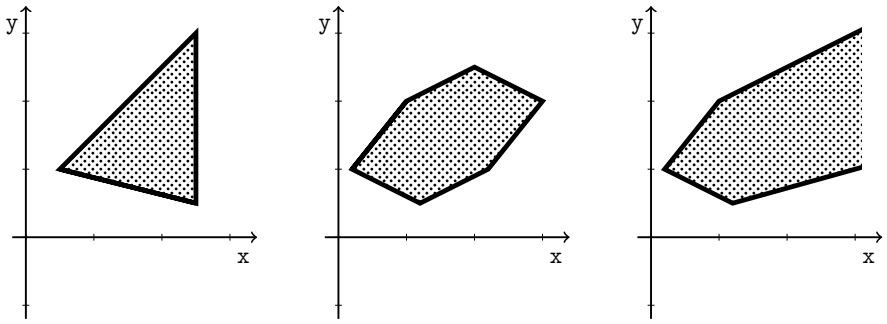
Example 2.6 (Convex polyhedra abstraction) *Figure 2.8 displays the concretization of three convex polyhedra a_0, a_1, and a_2:*

- *a_0 describes the conjunction of the three linear constraints below:*

$$\begin{array}{rcrcl} \mathtt{x} & - & \mathtt{y} & \geq & -0.5 \\ \mathtt{x} & & & \leq & 2.5 \\ \mathtt{x} & + & 4\mathtt{y} & \geq & 4.5 \end{array}$$

- *a_1 consists of the conjunction of six linear constraints, and is of bounded size (we do not list the constraint representation as it would be more involved);*
- *a_2 consists of the conjunction of four linear constraints and describes an unbounded zone (its concretization describes points where $\mathtt{x}, \mathtt{y}$ may be arbitrarily large).*

There exist several representations for the abstract elements of the convex polyhedra abstract domain. We have already mentioned the representation based on a conjunction of linear inequalities. Since we also noticed that their concretization corresponds exactly to convex polyhedra (hence the name of the abstraction), the abstract elements also have a geometrical representation, based on their sets of vertexes and edges. Actual static analysis algorithms based on convex polyhedra exploit both representations. However, these representations are significantly more complex and costly than in the case of the previous abstract domains: while signs only required a couple of bits per variable, and intervals only required at most two bounds per variable, defining a convex polyhedron typically involves



(a) Concretization of a_0 (b) Concretization of a_1 (c) Concretization of a_2

Figure 2.8
Convex polyhedra abstraction

a large number of coefficients or vertexes and edges (in theory there exists no upper bound on the number of cosntraints).

Another interesting remark about the convex polyhedra abstraction is that concrete sets of points have no best abstraction in general. A disc of diameter 1 provides an example of a concrete set without a smallest enclosing convex polyhedron. On the other hand, *some* concrete sets have a best abstraction (in particular, any set that is a convex polyhedron is its own smallest enclosing convex polyhedron).

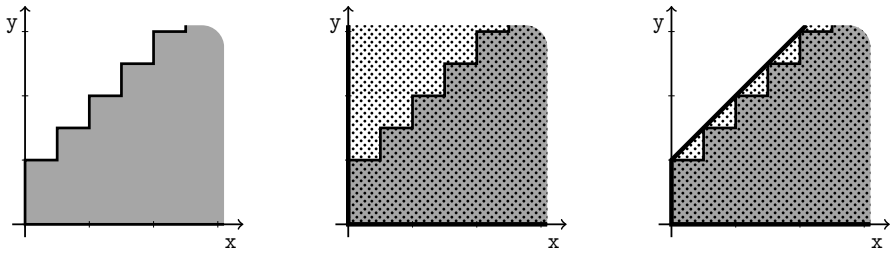
In the previous paragraphs, we have provided a few common examples of abstract domains, but many others can be defined and are useful to capture all sorts of constraints (simple or complex, relational or non-relational).

Abstraction of the semantics of a program. We can now refine the goal of the rest of the chapter. We have set up the notion of abstraction of sets of program states and have shown a few basic abstract domains, adapted to express different kinds of properties. In the next sections, we aim at defining static analysis algorithms to compute in a fully automatic way an over-approximation of the states that a program may reach. Such an over-approximation will be described by an abstract element in one of the abstract domains that we have sketched. It is called a *conservative abstraction* of the program semantics.

Example 2.7 (Abstractions of reachable states) *We consider the program of Example 2.3. Figure 2.9(a) shows all the states that this program may reach: a few program executions were sketched in Figure 2.3(b), and we consider here the set of* all *the states that can be reached in at least one execution. We then show the best abstractions that can be computed for this set of states:*

- *using the intervals abstract domain in Figure 2.9(b);*
- *using the convex polyhedra abstract domain in Figure 2.9(c).*

Obviously, the abstraction based on convex polyhedra is much tighter, even though it is still approximate, due to convexity.



(a) Reachable states (b) Intervals abstraction (c) Convex polyhedra abstraction

Figure 2.9
Program reachable states and abstraction

As shown in Example 2.7, not all abstractions of the semantics of the program are equivalent. Abstractions that describe fewer points are more selective, since they filter out more concrete points, and are thus more likely to help prove that the reachable states are included into a specific set, to prove the property of interest. This means that set inclusion here is fundamental to our study:

- for an abstract element to be a conservative abstraction of the semantics of programs, it should include all the points that are reachable according to the semantics;
- if two abstract elements a_0, a_1 are such that $\gamma(a_0)$ is included into $\gamma(a_1)$, then it means that a_0 is more precise than a_1 in the sense that it allows proving stronger semantic properties.

2.3 A Computable Abstract Semantics: Compositional Style

As the notion of abstraction has been set up in Section 2.2, we now show how to derive step-by-step an over-approximation for the states that are visited by a program.

In this section, we introduce a compositional approach to static analysis, based on the step-by-step computation of the effect of each program command. More precisely, given an abstraction of a set of states that denotes a pre-condition (i.e., a set of program execution starting points), we propose to compute an abstract state, that over-approximates the set of all the states that may be observed after running that command from the pre-condition (this set is usually called a post-condition). To analyze a sequence of commands, this technique composes the analyses of each sub-command, which is why it is called *compositional*.

Intuitively, this approach incrementally discovers an over-approximation of the set of reachable states of the program. Thus, it also makes it possible to verify that a program never reaches any state in the error zone. Using an accumulator, it is also possible to keep track of an abstraction of all the reachable states of the program.

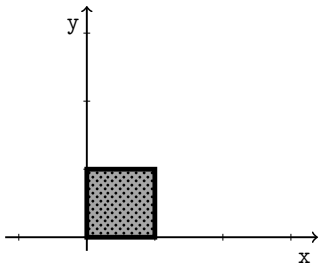


Figure 2.10
Analysis of initialization

2.3.1 Abstraction of Initialization

We start with the effect of the initialization statement that appears at the beginning of programs. At the concrete level, a program initialization statement simply asserts that the initial state of a program execution is located in a given region $\mathfrak{R}$.

To produce an abstraction of the result of initialization, the static analysis simply needs to produce an abstract element that over-approximates the region $\mathfrak{R}$.

When the abstract domain features a best abstraction function α and when the best abstraction of the region $\mathfrak{R}$ is computable, then the abstract element $\alpha(\mathfrak{R})$ provides a solution. This is the case of the intervals abstract domain and of the signs abstract domain.

When the abstract domain does not have a best abstraction function or when this best abstraction is not computable, any abstract element a such that $\gamma(a)$ includes $\mathfrak{R}$ can be chosen. For instance, the convex polyhedra abstract domain does not feature a best abstraction function; however,

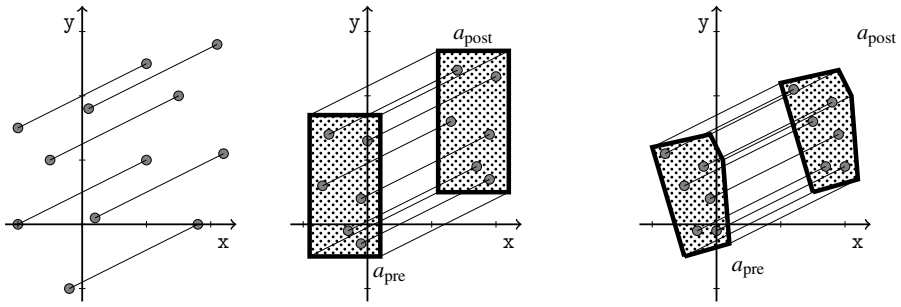
- if $\mathfrak{R}$ is a convex polyhedron, then it can be used as an over-approximation of itself;
- otherwise, an enclosing box can be found by using the intervals abstract domain abstraction, and this enclosing box is also an enclosing convex polyhedron, so it also provides an admissible solution (although an imprecise one).

Example 2.8 (Initialization) *We consider the program of Example 2.3. Figure 2.10 shows the best abstraction of the initial states, both with the intervals abstract domain and with the convex polyhedra abstract domain. We remark that this abstraction is* exact*; namely, it incurs no loss of precision.*

2.3.2 Abstraction of Post-Conditions

We now discuss basic geometric transformations and try to find a systematic way to over-approximate their output, when given an abstraction of their input. We first fix some terminology:

- an *abstract pre-condition* is an abstraction of the states that can be observed *before* a program fragment;



(a) Concrete semantics (b) Intervals (c) Convex polyhedra

Figure 2.11
Abstraction of the result of a translation

- an *abstract post-condition* is an abstraction of the states that can be observed *after* that program fragment.

Effect of a translation. We assume an abstract pre-condition a_{pre} and consider program `translation`(u,v). When the program is run in state $(\mathtt{x},\mathtt{y})$ in $\gamma(a_{\text{pre}})$, the result is the state $(\mathtt{x}+u,\mathtt{y}+v)$. Thus, the set of all the images of the points in $\gamma(a_{\text{post}})$ can be obtained very simply by translating $\gamma(a_{\text{pre}})$ by (u,v). Thus, to produce an over-approximation of the effect of the translation, we simply need to compute an abstract element a_{post} that contains all the points obtained by translating a point in $\gamma(a_{\text{pre}})$.

Example 2.9 (Translation) *We consider* `translation`$(2,1)$ *and the computation of abstract post-condition with a couple of example abstract domains. The effect of the program is shown in Figure 2.11(a): any execution boils down to a pair of states.*

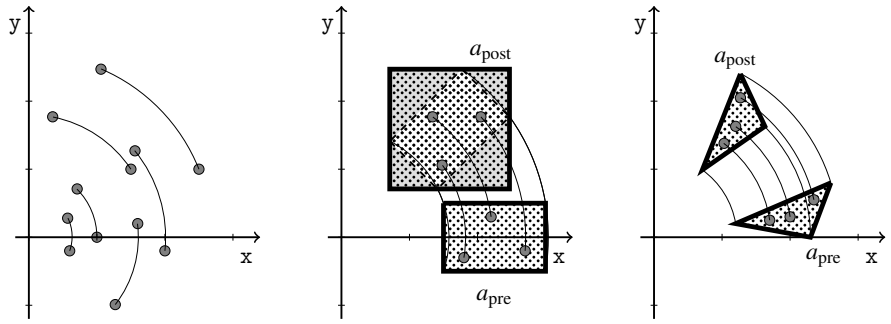
In Figure 2.11(b), we demonstrate the computation of an abstract post-condition with the abstract domain of intervals under the assumption of a given abstract pre-condition. The element a_{post} is obtained directly from a_{pre} by applying translation $(2,1)$. If we consider a translation defined by another vector, an abstract post-condition in the intervals abstract domain can be derived from the abstract pre-condition in a similar way.

The case of the abstract domain of convex polyhedra is similar to the case of intervals, as shown in Figure 2.11(c).

In both cases, we note that the abstract post-condition not only contains all the points in the image of the concretization of the pre-condition, but it also contains no other point*; in this sense, the post-condition is* exact*.*

Effect of a rotation. We now consider a program of the form `rotation`(u,v,θ). The same reasoning towards the design of an automatic algorithm to compute abstract post-conditions from an abstract pre-condition as for the translation still holds. We discuss this transformation in the following example:

Example 2.10 (45°rotation) *To fix the ideas, we assume $(u,v)=(0,0)$ and $a=45^\circ$ (45°rotation around the origin). A few concrete executions are depicted in Figure 2.12(a).*



(a) Concrete semantics (b) Intervals (c) Convex polyhedra

Figure 2.12
Abstraction of the result of a 45°rotation

First, we discuss the case of polyhedra, as it is actually simpler than the case of intervals. Figure 2.12(c) shows that the abstract post-condition can be computed exactly in the same way as for the translation in the previous paragraph. Indeed, if the analysis computes the image of the abstract pre-condition by the rotation, the resulting convex polyhedron contains all the images of the points in the concretization of the pre-condition and thus provides a precise over-approximation of the points that the program may reach after the rotation.

The case of intervals is shown in Figure 2.12(b). Intuitively, rotating the pre-condition should give a safe over-approximation of the points that the program may produce after the rotation, but the rotated box is not a valid element of the intervals abstract domain. Indeed, we defined the intervals abstract domain as the set of (finite or infinite) rectangles that are parallel to the axes, and the image of the pre-condition by the rotation is not parallel to the axes. Therefore, to produce a conservative post-condition, that is also an element of the abstract domain, the analysis should produce a bigger box, which is parallel to the axes, as shown in Figure 2.12(b). But this result is somewhat imprecise: indeed, as usual, the area filled with dots describes the result of the analysis, and the part of that zone that has a gray background corresponds to points that cannot be observed when running the program from any point in the pre-condition, yet these points have to be included in the result of the analysis due to the limited expressiveness of the intervals abstraction. Such imprecisions may ultimately prevent the analysis from proving the property of interest.

Conservative abstract transfer functions. Based on the two transformations that we have studied so far, we now summarize how the analysis should compute post-conditions in the abstract level. In general, we call an abstract operation that accounts for the effect of a basic program statement a *transfer function*. The definition below formalizes the soundness property that all transfer functions should satisfy.

Definition 2.6 (Sound analysis by abstract interpretation (compositional style)) *We consider a static analysis function* `analysis` *that inputs a program and an abstract pre-condition and returns an abstract post-condition. We say that* `analysis` *is sound if and only if the following condition holds:*

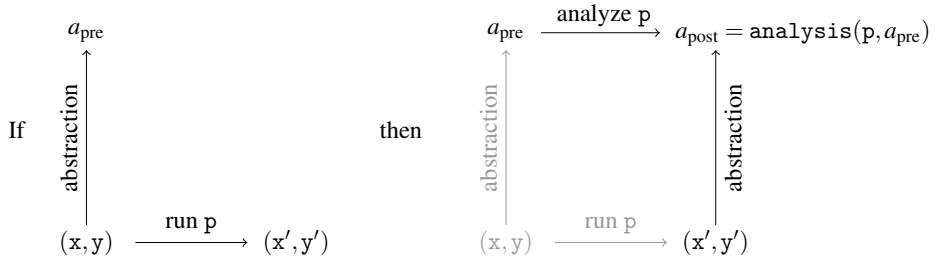


Figure 2.13
Sound analysis of a program p

If an execution of p *from a state* $(\mathtt{x},\mathtt{y})$ *generates the state* $(\mathtt{x}',\mathtt{y}')$,
then for all abstract element a such that $(\mathtt{x},\mathtt{y}) \in \gamma(a)$,
$$(\mathtt{x}',\mathtt{y}') \in \gamma(\mathtt{analysis}(\mathtt{p}, a))$$

Intuitively, this property states that the analysis should cover all executions of the program: whenever there exists an execution starting from a state that lies inside the abstract pre-condition, the output state should also belong to the abstract post-condition. The diagram of Figure 2.13 gives an intuitive presentation of the soundness property: when a concrete state can be described by an abstract pre-condition (bottom to top arrow in the left diagram) and is the starting point of an execution that reaches a final state (left to right arrow in the left diagram), running the analysis will close the diagram and return an over-approximation of the post state, as shown in the right diagram.

The transfer functions shown in the previous paragraphs for translations and rotations, for both the intervals and polyhedra abstract domains, satisfy the soundness property. Furthermore, we will make sure in the following that the analysis algorithms that we design for other program constructions still preserve this property.

This technique is an instance of *abstract interpretation*: it lets the analysis evaluate each program construction one by one, a bit like a standard interpreter would, albeit in the abstract domain.

At this point, we have defined:

$$\mathtt{analysis}(\mathbf{translation}(u,v),a) = \begin{cases} \text{return an abstract state that contains} \\ \text{the translation of } a \end{cases}$$

$$\mathtt{analysis}(\mathbf{rotation}(u,v,\theta),a) = \begin{cases} \text{return an abstract state that contains} \\ \text{the rotation of } a \end{cases}$$

Definition 2.6 entails that the analysis will produce sound results in the sense of Definition 1.2 when considering the property $\neg\mathfrak{D}$ of interest. Since the analysis over-approximates the states the program may reach, if it claims that $\neg\mathfrak{D}$ is not reachable, then we are sure that the program cannot reach $\neg\mathfrak{D}$.

On the other hand, this definition does not rule out imprecisions. Thus, it accepts analyses that produce coarse over-approximations. In the previous paragraphs we saw both precise analyses and imprecise analyses:

- with the convex polyhedra abstract domain, both the analysis functions for the translation and rotation are precise;
- on the other hand, with the intervals abstract domain, the analysis function for the rotation is imprecise.

Such imprecisions entail that the analysis is not complete in the sense of Definition 1.3, and that it may fail to prove that a given region is unreachable.

In the following, we continue the definition of the `analysis` function that can compute sound abstract post-conditions for any program in our language. We will proceed by induction over the syntax of programs. Indeed, we have already seen how to handle basic operations (initialization, translations and rotations); thus, we will now consider inductive cases.

The case of sequences of operations is trivial: to compute an abstract post-condition for $\mathtt{p}_0;\mathtt{p}_1$, we start from the abstract pre-condition a, compute an abstract post-condition $\mathtt{analysis}(\mathtt{p}_0, a)$ for $\mathtt{p}_0$, and then feed the result as the pre-condition to compute an abstract post-condition for $\mathtt{p}_1$:

$$\mathtt{analysis}(\mathtt{p}_0;\mathtt{p}_1, a) \quad = \quad \mathtt{analysis}(\mathtt{p}_1, \mathtt{analysis}(\mathtt{p}_0, a))$$

The other cases (for non-deterministic choice and iteration) are a bit more complex than the sequence case.

2.3.3 Abstraction of Non-Deterministic Choice

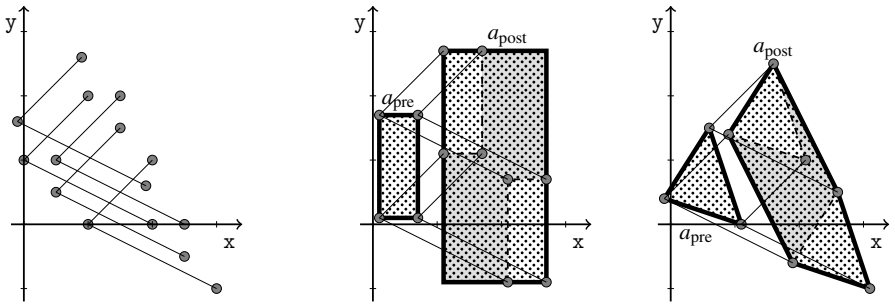
We now assume that $\mathtt{p}_0$ and $\mathtt{p}_1$ are two programs that we already know how to analyze and we propose constructing a way to over-approximate post-conditions for $\{\mathtt{p}_0\}\mathtt{or}\{\mathtt{p}_1\}$.

Let a be an abstract pre-condition and state $(\mathtt{x},\mathtt{y}) \in \gamma(a)$. Intuitively, we should consider two cases:

- either $\mathtt{p}_0$ is executed, and the result is in $\gamma(\mathtt{analysis}(\mathtt{p}_0, a))$;
- or $\mathtt{p}_1$ is executed, and the result is in $\gamma(\mathtt{analysis}(\mathtt{p}_1, a))$.

Thus, the analysis should simply produce an over-approximation of both $\mathtt{analysis}(\mathtt{p}_0, a)$ and $\mathtt{analysis}(\mathtt{p}_1, a)$.

The computation of an over-approximation for two abstract elements can be done in a systematic way for all the abstract domains that we have considered in Section 2.2. We can remark that this operation computes an over-approximation for the union of two sets of points viewed as abstract elements. Thus, we denote this abstract operation by `union`. In the case of intervals, the analysis should simply compute the minimum of lower bounds and the maximum of greatest bounds for both dimensions. In the case of convex polyhedra,



(a) Concrete semantics (b) Intervals (c) Convex polyhedra

Figure 2.14
Abstraction of the result of a non-deterministic choice

it should simply produce a convex hull for both abstract elements. To summarize:

$$\texttt{analysis}(\{\mathrm{p}_0\}\textbf{or}\{\mathrm{p}_1\},a) \quad = \quad \texttt{union}(\texttt{analysis}(\mathrm{p}_1,a),\texttt{analysis}(\mathrm{p}_0,a))$$

Example 2.11 (Analysis of non-deterministic choice) *In this example, we consider the very simple program below, and we show its analysis with both intervals and convex polyhedra:*

$$\{\texttt{translation}(2,1)\}\textbf{or}\{\texttt{translation}(-2,-1)\}$$

Figure 2.14(b) shows the computation of an abstract post-condition in the intervals abstract domain, and Figure 2.14(c) shows the computation of an abstract post-condition in the convex polyhedra abstract domain. These two cases are quite similar since each branch of the non-deterministic choice boils down to a geometric translation (which induces a translation of the shape of abstract elements), and the analysis should then return an over-approximation of the effects of both branches. In both domains, this operation incurs a significant loss of precision due to the approximation of the convex hull.

The above example shows another reason for the incompleteness of our analysis, as it cannot express precise disjunctive properties. This is a common issue in static analysis, and we present several solutions to this problem in Section 5.1.

2.3.4 Abstraction of Non-Deterministic Iteration

Non-deterministic iteration is the last construction that we have to define the analysis for. It is also the most complex construction to analyze since it can produce executions of any length and even infinite executions. Therefore, the analysis should compute in *finite time* an over-approximation for *infinitely many arbitrarily long executions*. Still, we observed in Example 2.3 that we can still derive interesting properties about such programs with rather short informal proofs. Thus, we are now going to generalize this approach and design analysis algorithms that compute an over-approximation for the set of output states of a loop.

Note that the abstract post-condition that is produced as the analysis result only describes the final states of the terminating program executions. This result means that "if a concrete execution terminates, then this abstract post-condition holds." The result does not mean that "the iteration will terminate with this abstract post-condition." This is due to the fact that the halting problem cannot be computed exactly in finite time.

In the following, we consider the following program p that consists of a loop with body b:

$$\mathtt{p} ::= \left\{ \begin{array}{l} \mathbf{iter}\{ \\ \quad \mathtt{b} \\ \} \end{array} \right.$$

We can discriminate the executions of p depending on the number of iterations of the loop; indeed, an execution of program p executes b either zero time, or one time, or two times, or three times, and so on. Thus, p is conceptually equivalent to the following (infinite) program:

$$\begin{array}{l} \{\} \\ \mathbf{or}\{\mathtt{b}\} \\ \mathbf{or}\{\mathtt{b};\mathtt{b}\} \\ \mathbf{or}\{\mathtt{b};\mathtt{b};\mathtt{b}\} \\ \mathbf{or}\{\mathtt{b};\mathtt{b};\mathtt{b};\mathtt{b}\} \\ \vdots \end{array}$$

This program fully eliminates the loop and resorts only to the **or** construct which can be analyzed as observed in Section 2.3.3, though it obviously cannot be completely written since it would be infinite. However, if we focus on the executions that spend at most k iterations in the loop, we can easily write a program without a loop that has exactly the same behaviors. For all integer k, we let $\mathtt{b}_k$ denote the program that iterates b k times ($\mathtt{b}_0$ is $\{\}$, $\mathtt{b}_1$ is b, $\mathtt{b}_2$ is b;b...). Moreover, we write $\mathtt{p}_k$ for $\{\mathtt{b}_0\}\mathbf{or}\{\mathtt{b}_1\}\mathbf{or}\ldots\mathbf{or}\{\mathtt{b}_{k-1}\}\mathbf{or}\{\mathtt{b}_k\}$. In short:

$$\begin{array}{llll} \text{program} & \mathtt{p}_0 & \text{is} & \{\} \\ \text{program} & \mathtt{p}_1 & \text{is} & \{\}\mathbf{or}\{\mathtt{b}\} \\ \text{program} & \mathtt{p}_2 & \text{is} & \{\}\mathbf{or}\{\mathtt{b}\}\mathbf{or}\{\mathtt{b};\mathtt{b}\} \\ \text{program} & \mathtt{p}_3 & \text{is} & \{\}\mathbf{or}\{\mathtt{b}\}\mathbf{or}\{\mathtt{b};\mathtt{b}\}\mathbf{or}\{\mathtt{b};\mathtt{b};\mathtt{b}\} \\ & & & \vdots \end{array}$$

Then we observe the following equivalence, which relates these programs all together:

$$\mathtt{p}_{k+1} \quad \text{is equivalent to} \quad \mathtt{p}_k\mathbf{or}\{\mathtt{p}_k;\mathtt{b}\}$$

Indeed, an execution of $\mathtt{p}_{k+1}$ either executes the loop at most k times (hence, it is an execution of $\mathtt{p}_k$), or it runs it $k+1$ times exactly (and then it is an execution of $\mathtt{p}_k;\mathtt{b}$). Conversely, one can can show that an execution of $\mathtt{p}_k\mathbf{or}\{\mathtt{p}_k;\mathtt{b}\}$ is also an exexcution of $\mathtt{p}_{k+1}$.

Therefore, the analysis of this sequence of programs can be computed recursively as follows:

$$\mathtt{analysis}(\mathtt{p}_{k+1},a) = \mathtt{union}(\mathtt{analysis}(\mathtt{p}_k,a),\mathtt{analysis}(\mathtt{b},\mathtt{analysis}(\mathtt{p}_k,a)))$$

This approach corresponds to the analysis algorithm that inputs an abstract pre-condition a, stores it into a variable R and iterates the operation:

$$\mathtt{R} \leftarrow \mathtt{union}(\mathtt{R},\mathtt{analysis}(\mathtt{b},\mathtt{R}))$$

Moreover, as shown above any execution of p can be characterized by the number of times it iterates over the loop. Thus, any execution is ultimately covered by repeating this iterative abstract computation.

The following example illustrates this approach:

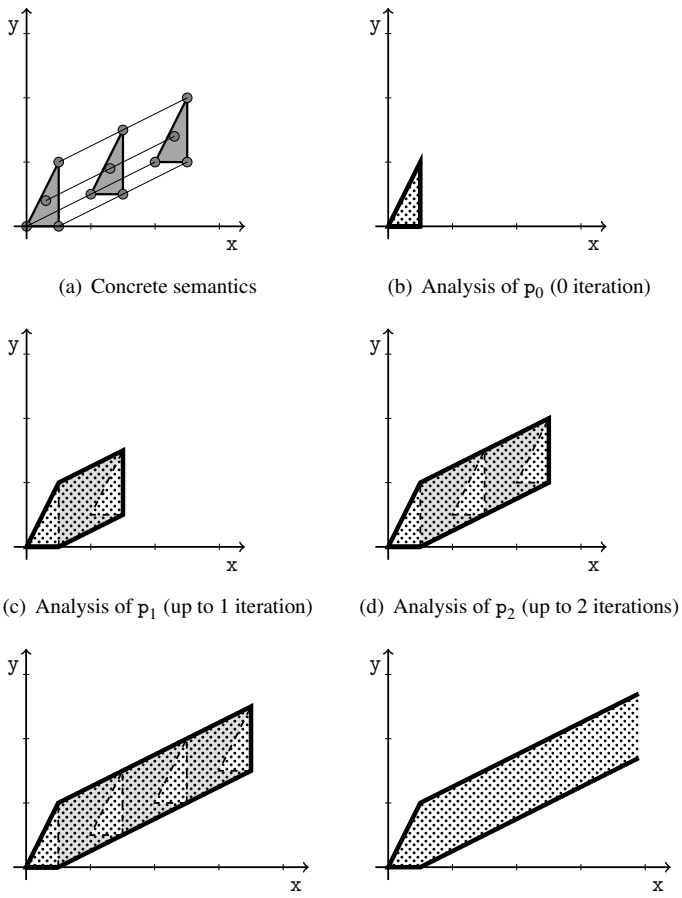
Example 2.12 (Abstract iteration) *We consider the program below, which starts at a point located in a triangle and iterates a basic geometric translation a non-deterministically chosen number of times:*

```
init({(x,y) | 0 ≤ y ≤ 2x and x ≤ 0.5});
iter{
    translation(1,0.5)
}
```

We assume that the analysis uses the convex polyhedra abstract domain. Then the set of states observed after initialization and before the **iter** *statement is shown in Figure 2.15(b). We show in Figure 2.15(c), Figure 2.15(d) and Figure 2.15(e) the first three iterations of the analysis algorithm that was sketched above. The imprecision is inherent in the computation of over-approximations (in gray) of abstract elements as in the previous examples.*

While this process does not terminate, we note that repeating it forever would yield the result shown in Figure 2.15(f), which also provides a sound approximation of all the possible output states of the program.

The iterative algorithm demonstrated in Example 2.12 will actually always terminate if using the signs abstract domain. Indeed, this abstract domain has a finite number of abstract elements and when the iterative algorithm computes $\mathtt{R} \leftarrow \mathtt{union}(\mathtt{R},\mathtt{analysis}(\mathtt{b},\mathtt{R}))$, the value of R will converge after finitely many steps: whenever it updates R, the new value is either the same as the previous one (and then so will be all the other subsequent values since they are computed using the same formula), or the new value denotes a *strictly* less precise property. Since the number of abstract properties is finite, the latter case will occur at most finitely many times. Therefore, at some point, the value of R stabilizes, and then



(a) Concrete semantics

(b) Analysis of p_0 (0 iteration)

(c) Analysis of p_1 (up to 1 iteration)

(d) Analysis of p_2 (up to 2 iterations)

(e) Analysis of p_3 (up to 3 iterations)

(f) Expected result

Figure 2.15
Abstract iteration

this value over-approximates the behaviors observed after *any* number of iterations. As a consequence, the termination of signs analysis is guaranteed.

However, we have not solved the issue of termination in the general case yet. The iteration technique used in Example 2.12 will obviously not allow for a terminating analysis with the convex polyhedra abstraction or with the interval abstraction.

To ensure termination of the analysis, we need to enforce the convergence of abstract iterates, possibly at the price of a coarser result. Note that a common way to prove that an algorithm terminates involves finding a strictly positive value that decreases strictly over time toward a finitely reachable basis. Thus, a way to enforce the termination of the

analysis is to exhibit such a measure. Very often, non-termination is due to some loop indexes not being incremented properly, preventing such a decreasing measure to exist. Intuitively, this is the issue the iterative analysis algorithm we sketched above suffers from, as shown in Example 2.12.

Another interesting observation is that abstract elements are made of finite sets of constraints. Therefore, another way to over-approximate abstract elements that arise in the abstract iteration would consist in forcing this number of constraints to decrease (possibly down to zero) until it stabilizes, hereby recovering termination.

Given the current constraint a_0, suppose that analyzing one more iteration generates a_1. To have an approximate constraint that subsumes both, we can let the analysis:

- keep all constraints of a_0 that are also satisfied in a_1;
- discard all constraints of a_0 that are not satisfied in a_1 (hence to subsume a_1).

Applying this method to abstract iterates will produce a sequence of abstract elements with a positive, decreasing number of constraints until the sequence stabilizes. This method is an instance of a general technique called *widening*, which enforces the convergence of abstract iterates. We denote this operator by `widen`:

$$\text{operator } \texttt{widen} \quad \begin{cases} \text{over-approximates unions} \\ \text{enforces convergence} \end{cases}$$

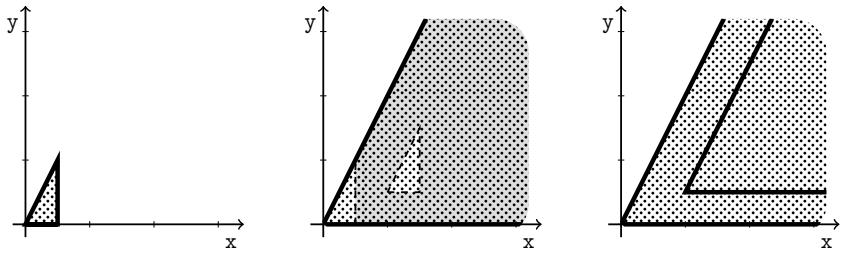
Stabilization holds when the concretization of the next iterate is included into that of the previous one. For all the abstract domains considered in this chapter, this inclusion can be decided in the abstract level simply by checking geometric inclusion. We thus let `inclusion` denote a function that inputs two abstract elements a_0, a_1 and returns **true** only when it can prove that $\gamma(a_0) \subseteq \gamma(a_1)$.

$$\text{operator } \texttt{inclusion} \qquad \text{returns } \textbf{true} \text{ only when it succeeds checking inclusion}$$

As a conclusion, the following algorithm computes an abstract post-condition for the loop construction:

$$\texttt{analysis}(\textbf{iter}\{\texttt{p}\}, a) \quad = \quad \begin{cases} \texttt{R} \leftarrow a; \\ \text{repeat} \\ \qquad \texttt{T} \leftarrow \texttt{R}; \\ \qquad \texttt{R} \leftarrow \texttt{widen}(\texttt{R}, \texttt{analysis}(\texttt{p}, \texttt{R})); \\ \text{until } \texttt{inclusion}(\texttt{R}, \texttt{T}) \\ \text{return } \texttt{T}; \end{cases}$$

This iteration technique will produce a sound result since it over-approximates the abstract elements produced by the sequence of iterates without widening, and its limit (reached



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2 and limit

Figure 2.16
Abstract iteration with widening

after finitely many iterates) also over-approximates all the abstract elements produced by the sequence of iterates without widening and, thus, the states that the program may reach.

The following example illustrates its use in practice:

Example 2.13 (Abstract iteration with widening) *We consider the same program as in Example 2.12. Figure 2.15 shows the sequence of abstract iterates using the widening technique. This sequence converges after only two iterations and produces a (rather coarse) over-approximation of the reachable states of the program (shown in Figure 2.15(a)). The most interesting point is the computation of the abstract element shown in Figure 2.16(b) from the two triangles obtained in the first two iterations:*

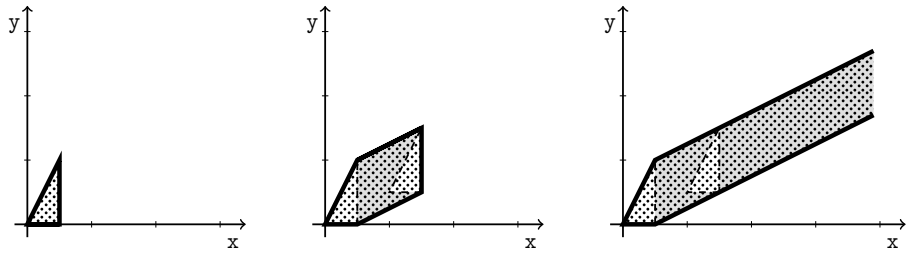
- *the constraints* $0 \leq \mathtt{y}$ *and* $\mathtt{y} \leq 2\mathtt{x}$ *are stable as they are satisfied in the translated triangle; thus, they are preserved;*
- *the constraint* $\mathtt{x} \leq 0.5$ *is not preserved; thus, it is discarded.*

The result obtained in the example clearly shows that widening is another source of imprecision and, thus, of potential incompleteness. Indeed, to ensure convergence in finite time, the analysis weakens the abstract elements more aggressively, adding many points that cannot be observed in any real program execution, as we can see in Figure 2.16(b).

Fortunately, there exist many techniques to make the analysis of loops more precise. The example below demonstrates a classic such technique on the same code.

Example 2.14 (Loop unrolling) *We note that we can rewrite a program with a loop in different ways than the one used so far in this section. In particular, the program of Example 2.12 is equivalent to the following program:*

```
init({(x,y) | 0 ≤ y ≤ 2x and x ≤ 0.5});
{}or{
    translation(1,0.5)
}
iter{
    translation(1,0.5)
}
```



(a) Iteration 0 (b) Iteration 1, `union` (c) Iteration 2, `widen`, limit

Figure 2.17
Abstract iteration with widening and unrolling

In essence, analyzing this second version instead has the following effect on the analysis: for the first iteration, the `union` *operator will be used, and for all subsequent iterations,* `widen` *will be used instead. When computing widening at iteration 2, all constraints are stable but the constraint* $\mathtt{x} \leq 1.5$*. This produces the result shown in Figure 2.17(c). Thus, both the result of the first iteration (shown in Figure 2.16(b)) and the widening output (shown in Figure 2.17(c)) are a lot more precise than with the standard widening iteration technique presented in Example 2.13.*

2.3.5 Verification of the Property of Interest

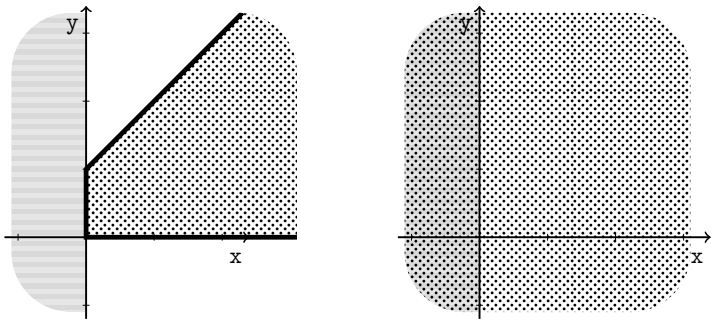
The analysis function that we have designed allows verifying the reachability property of interest that we introduced in Section 2.1.

While the analysis function that we have shown so far only returns an over-approximation of the output states (and not of all the intermediate reachable states), it actually computes as intermediate results over-approximations for *all* the reachable states of the input program. Let us consider the case of a sequence $\mathtt{p}_0;\mathtt{p}_1$. The analysis then returns $\mathtt{analysis}(\mathtt{p}_1, \mathtt{analysis}(\mathtt{p}_0, a_{\text{pre}}))$. We observe that after analyzing $\mathtt{p}_0$ and before analyzing $\mathtt{p}_1$, the analysis holds an over-approximation of all the states that can be observed after executing $\mathtt{p}_0$ and before executing $\mathtt{p}_1$ (the abstract element $\mathtt{analysis}(\mathtt{p}_0, a_{\text{pre}})$). The same holds for each kind of instruction of our language.

As a consequence, the analysis can attempt at verifying the property of interest by checking that the abstract elements computed at each step have an empty intersection with $\mathfrak{D}$, or equivalently, are included in $\neg\mathfrak{D}$. This inclusion can be fully verified in the abstract level, using the same inclusion test as we have used for checking the termination of the sequences of abstract iterates.

We assume the analysis uses the abstract domain of convex polyhedra and illustrate successful and unsuccessful analyses in the two examples below:

Example 2.15 (Successful verification) *Figure 2.18(a) shows the over-approximation computed for the set of all the reachable states of the program of Example 2.3. In this case, the over-approxima-*



(a) Program of Example 2.3 (b) Program of Example 2.1

Figure 2.18
Abstractions of reachable states

tion does not *intersect* $\mathfrak{D}$*: thus, the analysis proves the program correct. Again, this result is in line with the conclusion of Example 2.7 that this program is correct.*

Example 2.16 (Unsuccessful verification) *Figure 2.18(b) shows the over-approximation computed for the set of all the reachable states of the program of Example 2.1 (region filled with dots). Since this zone corresponds to the whole field and intersects the error zone* $\mathfrak{D}$*, the analysis cannot prove the property of interest for this program. This was to be expected. Example 2.2 has shown executions of this program that enter* $\mathfrak{D}$*. While the analysis rightfully flags this program as "potentially wrong," it does not produce a proof that the program is definitely wrong (though we will see in Section 5.5 that there exist static analysis techniques to achieve this proof in certain cases).*

2.4 A Computable Abstract Semantics: Transitional Style

In Section 2.3, the `analysis` function has no explicit machinery to collect all intermediate, reachable states. In other words, it is extensionally defined, analogous to the denotational (or compositional) approach to the semantics. Its inductive definition over the syntactic structure of the program returns just a post-state of the input program from a pre-state. No collection of intermediate states is manifest in the definition. As we discussed in Section 2.3.5 though, a simple monitoring mechanism on top of `analysis` can collect all occurring intermediate states.

In this section, we introduce a different style of the analysis function. The new analysis function computes, from the outset, all occurring intermediate states. This formulation is analogous to an operational approach to the semantics.

This transitional style provides us with another convenient perspective that sheds a new light over static analysis. In subsequent chapters, we will observe that the compositional style is better suited for some problems, whereas the transitional style is a better fit for

others. Therefore, understanding both styles is beneficial to better grasp static analysis techniques in general.

2.4.1 Semantics as State Transitions

In the transitional style, we view an execution of a program as a sequence of transitions between states. This transition sequence exposes all the states that occur during the execution.

Let us consider the example language (Section 2.1) of this chapter. In this language, a program execution moves a point in the two-dimensional space. In this case, a state can be defined as a pair of a statement label l and a point p in the two-dimensional space.

A single transition

$$(l, p) \hookrightarrow (l', p')$$

between states represents that the program at statement label l transforms the point p to p' and passes it to the next statement label l' for continuation.

One proper transition corresponds to a "single-step" execution of a basic statement. For a compound statement that consists of other statements, its execution consists of the transitions of its sub-statements.

An example of transition sequences for an example program will be shown shortly in the next section, after we define how we represent programs and what we mean by "statement labels."

State transitions and the collection of all states. Let our analysis goal be to collect all the states occurring in all possible transition sequences of the input program. Given such a set of all reachable states, we can check, for example, whether every reachable state remains in a safe zone of our interest.

Figure 2.19 illustrates transition sequences and the collection of states occurring in the sequences. Each node s_i is a state (l, p): a pair of a statement label and a point set in the two-dimensional space that is to be transformed by the statement at the label. Here, we schematically show the transition sequences and occurring states. We will show in Example 2.17 concrete examples of transition sequences.

Statement labels and execution order We view a program just as a collection of statements with a well-defined execution order. We assign a unique label to each statement of the program. This label can be understood as the so called *program counter* or *program point*. The execution order, between statements, so called *control flow*, is specified by a relation between the labels (from current program points to next program points).

Since our language has a non-deterministic choice and non-deterministic iterations, the execution order is non-deterministic too. Entering the or-statement $\{\mathtt{p}\}\mathbf{or}\{\mathtt{p}'\}$, the next statement to execute is either $\mathtt{p}$ or $\mathtt{p}'$. Entering the iteration statement $\mathbf{iter}\{\mathtt{p}\}$, the next

$s_1 \hookrightarrow s_2 \hookrightarrow s_5 \hookrightarrow s_3 \hookrightarrow s_8 \hookrightarrow \cdots$

$s_6 \hookrightarrow s_7 \hookrightarrow s_8 \hookrightarrow s_3 \hookrightarrow s_4$

$s_9 \hookrightarrow s_{10} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_{13}$

$s_{12} \hookrightarrow s_7 \hookrightarrow s_2 \hookrightarrow s_3 \hookrightarrow s_4 \hookrightarrow s_{14}$

States s_1, s_6, s_9, and s_{12} are initial states.

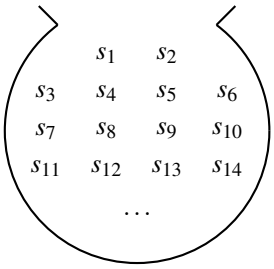


Figure 2.19
Transition sequences and the set of occurring states

statement to execute is either the loop body p or the next statement after the exit of the loop. The next statement of the loop body is again the iteration statement.

For example, consider an example program in Figure 2.20. Each statement has a unique label. Figure 2.20(a) shows the program text with statement labels in circles. The statement corresponding to a label is circumscribed by a dotted box. Figure 2.20(b) shows a graphical representation of the program with its execution order as directed edges. Rectangular nodes are either basic statements or heads of compound statements. Numbered circle nodes are statement labels.

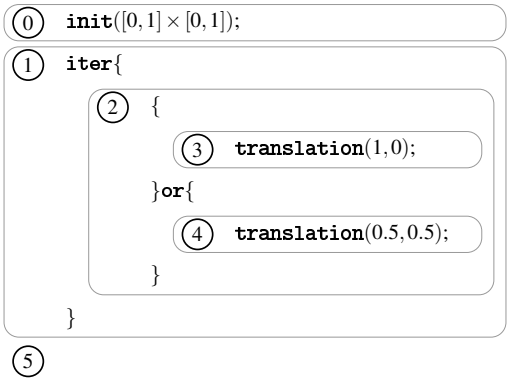
The non-deterministic function (or relation) for the execution order is defined as follows (as visible in the graph view of Figure 2.20(b)).

$$\begin{array}{lllllll} \texttt{next}(0) & = & 1 & & & & \\ \texttt{next}(1) & = & 2 & \texttt{next}(1) & = & 5 \\ \texttt{next}(2) & = & 3 & \texttt{next}(2) & = & 4 \\ \texttt{next}(3) & = & 1 & \texttt{next}(4) & = & 1 \end{array}$$

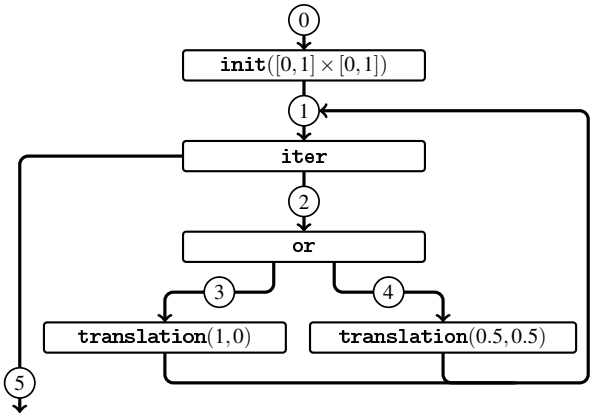
Note that, in general, for most modern languages the execution order (control flow) is not syntactically obvious. For example, when a language has a dynamic jump construct such as dynamic goto label, dynamic method dispatch, higher-order function call, or the raise of an exception whose target is computed only during execution, the exact execution order is not available before the static analysis. For such languages, determining the execution order should be a part of the static analysis under design or needs to be computed beforehand by another separate static analysis.

In Chapter 4, we will present a formal framework that covers such dynamic control-flow cases. Our example language in this chapter is one whose control-flow is obvious from the syntax.

Example 2.17 (Transition sequences) *For the example program in Figure 2.20, two examples of state transition ($\hookrightarrow$) sequences starting from statement 0 are as follows: recall that a state (l, p) is a pair of a statement label (l) and a point (p) right before being transformed by the corresponding statement. The left sequence is a transition sequence when the program terminates after two*



(a) Text view, with labels



(b) Graph view, with labels

Figure 2.20
Example program with statement labels

iterations; the right one is when the same program terminates after one iteration.

$$
\begin{array}{lcl}
(0,p_0) & \hookrightarrow & (1,p_1) \\
& \hookrightarrow & (2,p_1) \\
& \hookrightarrow & (3,p_1) \\
& \hookrightarrow & (1,p_2) \\
& \hookrightarrow & (2,p_2) \\
& \hookrightarrow & (4,p_2) \\
& \hookrightarrow & (1,p_4) \\
& \hookrightarrow & (5,p_4)
\end{array}
\qquad
\begin{array}{lcl}
(0,p_0) & \hookrightarrow & (1,p_1) \\
& \hookrightarrow & (2,p_1) \\
& \hookrightarrow & (4,p_1) \\
& \hookrightarrow & (1,p_3) \\
& \hookrightarrow & (5,p_3)
\end{array}
$$

where

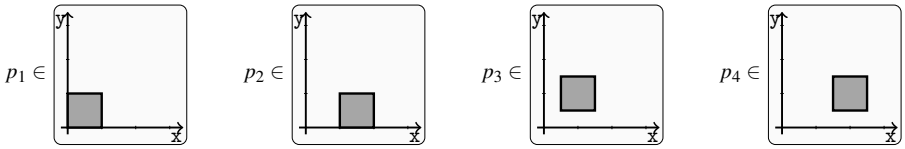


Figure 2.21 shows, on top of the graph view of the program, the right transition sequence.

2.4.2 Abstraction of States

Collecting the exact set of all the states that can occur during program executions (transition sequences) is in general either too costly or impossible in finite time. Indeed, due to loops, program executions may be arbitrarily long. Moreover, the set of initial states is also potentially infinite. The situation is worse for other conventional languages that receive inputs from outside. The number of possible inputs is usually combinatorially explosive or even infinite. That is, the number of transition sequences can be infinite too.

Hence, as discussed in Section 2.3, the static computation of the set of all possible states cannot be exact in general. Our static computation may only be an approximation in an abstract world.

Now the question is what abstract world we are going to use. As an illustration among many candidates, let us use the following statement-wise abstract world:

> For each statement (program point), an abstract element approximates the set of points that can occur at that program point during executions. The abstract elements for point sets are convex hull pre-conditions as used in Section 2.3. In other words, an abstract state is a set of pairs of statement labels and abstract pre-conditions.

Figure 2.22 schematically shows the state abstraction that we are using on top of the graphic view of a program. The areas in the two-dimensional plane depict the set of points that can occur during executions.

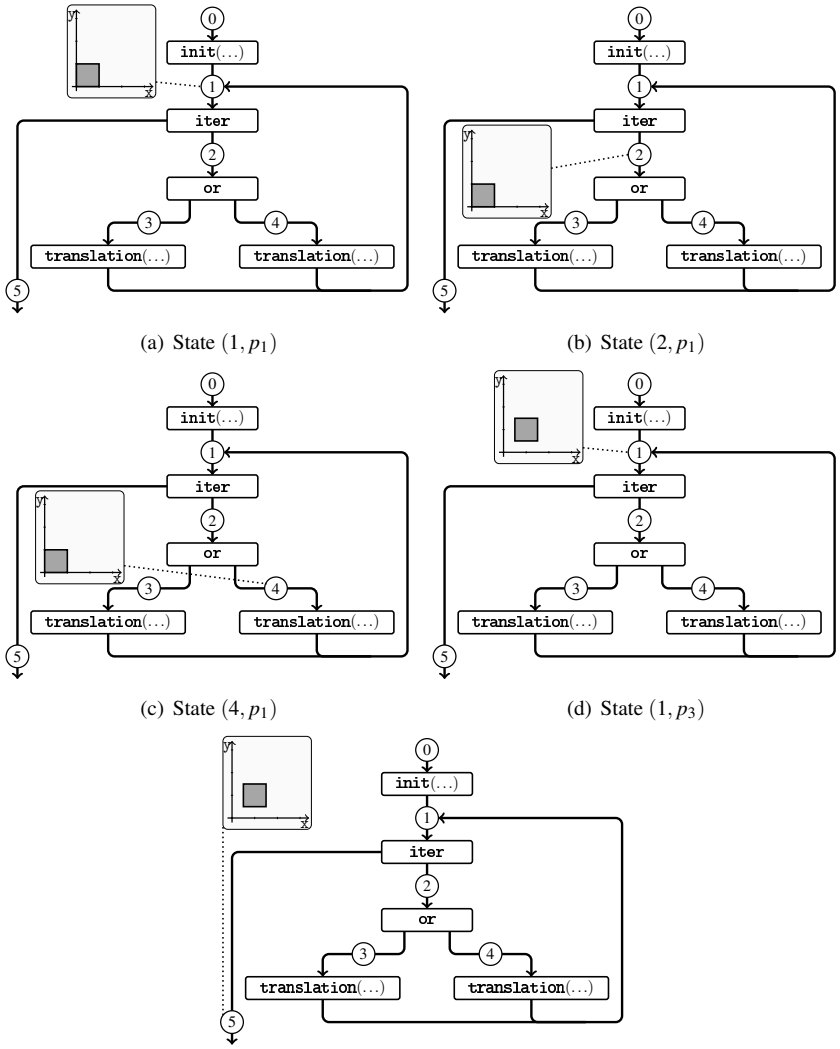
2.4.3 Abstraction of State Transitions

The abstract state transition is defined over the abstract states of the preceding section. Note that an abstract state is a set of pairs of statement labels and abstract pre-conditions. An abstract transition transforms an abstract state into another abstract state.

Let $Step^\sharp$ be such an abstract state transition function. Given an abstract state X, $Step^\sharp(X)$ returns an abstract post-state. The $Step^\sharp$ function is defined by the one-step abstract transition operator $\hookrightarrow^\sharp$, lifted for a set (for an abstract state):

$$Step^\sharp(X) = \{x' \mid x \in X, x \hookrightarrow^\sharp x'\}.$$

The one-step abstract transition $x \hookrightarrow^\sharp x'$ is the same as the post-condition computations in Section 2.3 except that the proper transition happens only for non-compound basic state-



(a) State $(1, p_1)$

(b) State $(2, p_1)$

(c) State $(4, p_1)$

(d) State $(1, p_3)$

(e) State $(5, p_3)$

Each point p_i belongs to the rectangular area of the corresponding two-dimensional plane.

Figure 2.21
States on top of the graph view of the program

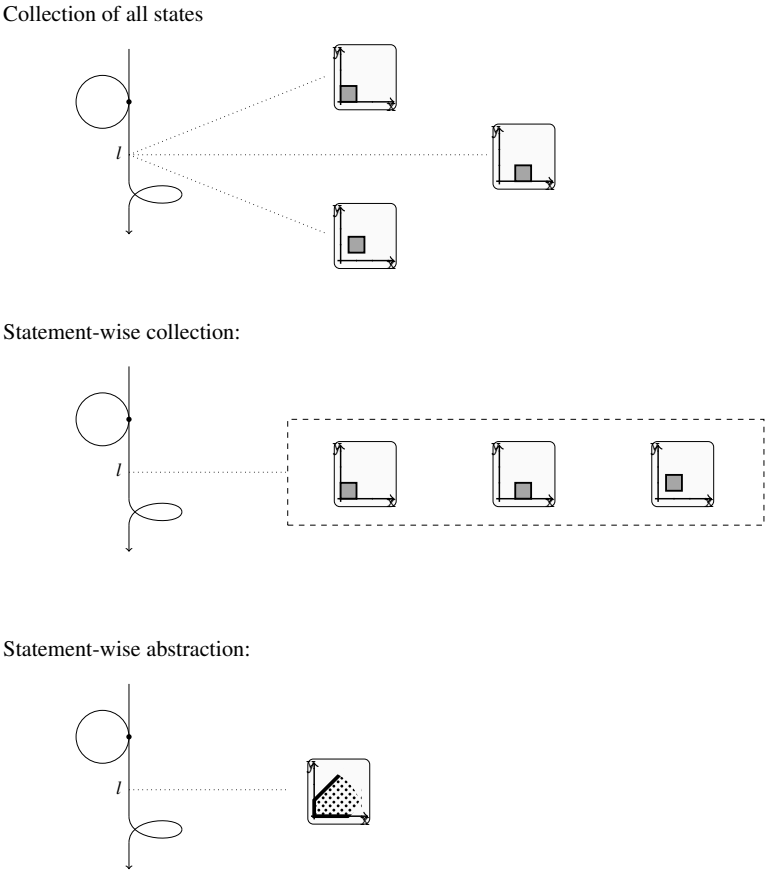


Figure 2.22
Statement-wise abstraction of all the possible states for statement label l

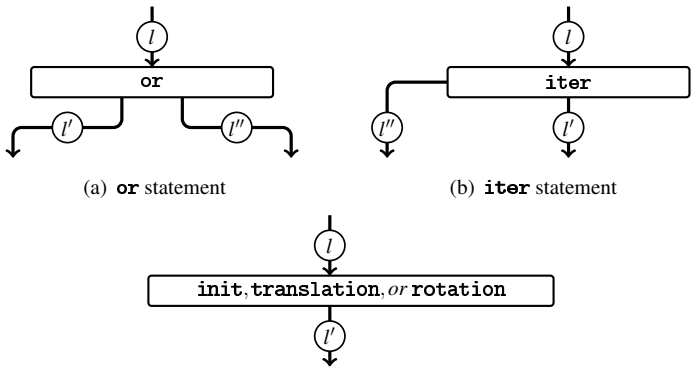
ments, and we reference the `next` function for the next label:

$$(\mathbf{or}_l, a_{\text{pre}}) \hookrightarrow^\sharp (\texttt{next}(l), a_{\text{pre}}) \quad \text{for an } \mathbf{or} \text{ statement at } l \text{ (Figure 2.23(a))}$$

$$(\mathbf{iter}_l, a_{\text{pre}}) \hookrightarrow^\sharp (\texttt{next}(l), a_{\text{pre}}) \quad \text{for an } \mathbf{iter} \text{ statement at } l \text{ (Figure 2.23(b))}$$

$$(\mathtt{p}_l, a_{\text{pre}}) \hookrightarrow^\sharp (\texttt{next}(l), \texttt{analysis}(\mathtt{p}_l, a_{\text{pre}})) \quad \text{else, for a basic statement } \mathtt{p}_l \text{ at } l \text{ (Figure 2.23(c))}$$

Note that the above *Step*$^\sharp$ function is sound because the `analysis` function (Section 2.3) is sound for basic statements (Figure 2.13).



(a) **or** statement (b) **iter** statement

(c) basic statement

Figure 2.23
Next labels, depending on the statement of l, both l' and l'', or l' only.

2.4.4 Analysis by Global Iterations

The static analysis for collecting all abstract states should be sound. The soundness means that the set of concrete states implied by the analysis result over-approximates the reality.

Definition 2.7 (Sound analysis by abstract interpretation in transitional style) *Let* $\texttt{analysis}_T$ *be a static analysis function in transitional style that inputs a program and returns a set of abstract states. We say that* $\texttt{analysis}_T$ *is* sound *if and only if the following condition holds:*

If S is the set of states occurring in a transition sequence of p *from initial state* s_0*, then for any abstract element a such that* $s_0 \in \gamma(a)$*,*

$$S \subseteq \gamma(\texttt{analysis}_T(\texttt{p}, a))$$

For the input program p and an abstract state I that over-approximates all the possible initial states, the analysis result $\texttt{analysis}_T(\texttt{p}, I)$ is a set of pairs (l, a_{pre}) of statement label l and abstract pre-condition a_{pre}. The soundness ensures that the abstract pre-conditions at label l over-approximate all the points that may occur at statement l during execution.

Such a sound analysis function $\texttt{analysis}_T(\texttt{p}, I)$ is composed of the sound abstract transition function $Step^\sharp$ in Section 2.4.3 as follows: letting $Step^{\sharp i}(I)$ denote the abstract post-states after i consecutive abstract transitions from I,

$$\begin{aligned} Step^{\sharp 0}(I) &= I \\ Step^{\sharp i+1}(I) &= Step^\sharp(Step^{\sharp i}(I)). \end{aligned}$$

This abstract state $Step^{\sharp i}(I)$ is sound: it subsumes all the states after i transitions from an initial state implied by I. This soundness of i consecutive applications of $Step^\sharp$ is clear because each step by $Step^\sharp$ over-approximates the results of a single-step transition. For our example language, the set I is $\{(0, \textbf{true})\}$, where the label 0 lies at the initial statement and its abstract pre-condition **true** implies all the points in the two-dimensional plane.

Then the analysis accumulates all the abstract states occurring at each step of the abstract transition from the initial abstract state I:

$$Step^\sharp{}^0(I) \cup Step^\sharp{}^1(I) \cup Step^\sharp{}^2(I) \cup \cdots.$$

Now, a natural question is how to devise an algorithm that accumulates the above sequence. Please note that the above sequence is equivalent to what we illustrated in Section 2.3.4 when we devised the analysis function for the **`iter`** statement, whose body is now $Step^\sharp$.

The analysis algorithm is to compute the "limit" ($\lim_{i\to\infty} C_i$) of the following sequence C_i:

$$C_i = Step^\sharp{}^0(I) \cup Step^\sharp{}^1(I) \cup \cdots \cup Step^\sharp{}^i(I).$$

Because the following equivalence holds

$$C_{k+1} \qquad \text{is equivalent to} \quad C_k \cup Step^\sharp(C_k),$$

the analysis algorithm can be defined such that from I, stores it into a variable `C` and iterates the operation

$$\texttt{C} \leftarrow \texttt{C} \cup Step^\sharp(\texttt{C})$$

until stable.

Hence, the analysis algorithm for the input program `p` is a monolithic global iteration:

$$\texttt{analysis}_T(\texttt{p}, I) \quad = \quad \begin{cases} \texttt{C} \leftarrow \{I\} \\ \text{repeat} \\ \qquad \texttt{R} \leftarrow \texttt{C} \\ \qquad \texttt{C} \leftarrow \texttt{union}_T(\texttt{C}, Step^\sharp(\texttt{C})) \\ \text{until } \texttt{inclusion}_T(\texttt{C}, \texttt{R}) \\ \text{return R} \end{cases}$$

The i-th iteration of the algorithm covers all states observed up-to i execution steps of the input program.

The $\texttt{union}_T$ and $\texttt{inclusion}_T$ operators do the same as the `union` and `inclusion` operators (Section 2.3.4), respectively, except that they are label-wise. That is, the $\texttt{inclusion}_T(\texttt{C}, \texttt{R})$ returns true only when *at every statement label* the local point-set implied from `C` is included in that from `R`. Similarly, the $\texttt{union}_T$ summarizes *for each statement label* its local set of collected pre-conditions. The summarization is done by applying the `union` operator

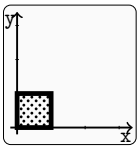
to reduce each local set of pre-conditions into a single pre-condition:

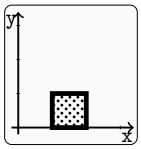
$$\texttt{union}_T(\texttt{C},\texttt{C}') \quad = \quad \begin{cases} \texttt{X} \leftarrow \{\} \\ \text{for each label } l \text{ in } \texttt{C} \cup \texttt{C}' \\ \qquad \texttt{S}_l \leftarrow \{a \mid (l,a) \in \texttt{C}\} \\ \qquad \texttt{S}'_l \leftarrow \{a \mid (l,a) \in \texttt{C}'\} \\ \qquad \texttt{T}_l \leftarrow \texttt{union}(\texttt{S}_l \cup \texttt{S}'_l) \\ \qquad \texttt{X} \leftarrow \texttt{X} \cup \{(l,\texttt{T}_l)\} \\ \text{return } \texttt{X} \end{cases}$$

Example 2.18 (Abstract transitions) *Consider the example program in Figure 2.20. Suppose we use the convex-polyhedra abstractions for point sets. The above analysis algorithm* $\texttt{analysis}_T$ *stores the following* C_i *iterates into the variable* C *after i iterations. Remember that* C_i *covers up-to i transitions of the input program:*

$$\begin{array}{lllll} \textit{after 0 iteration,} & C_0 & \stackrel{let}{=} & \{I\} \\ \textit{after 1 iteration,} & C_1 & \stackrel{let}{=} & \texttt{union}_T(C_0, \{(1,a_1)\}) \\ \textit{after 2 iterations,} & C_2 & \stackrel{let}{=} & \texttt{union}_T(C_1, \{(2,a_1),(5,a_1)\}) \\ \textit{after 3 iterations,} & C_3 & \stackrel{let}{=} & \texttt{union}_T(C_2, \{(3,a_1),(4,a_1)\}) \\ \textit{after 4 iterations,} & C_4 & \stackrel{let}{=} & \texttt{union}_T(C_3, \{(1,a_2),(1,a_3)\}) \\ & \vdots & & \vdots \end{array}$$

where

$a_1 =$ 

$a_2 =$ 

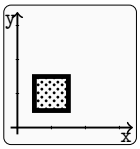
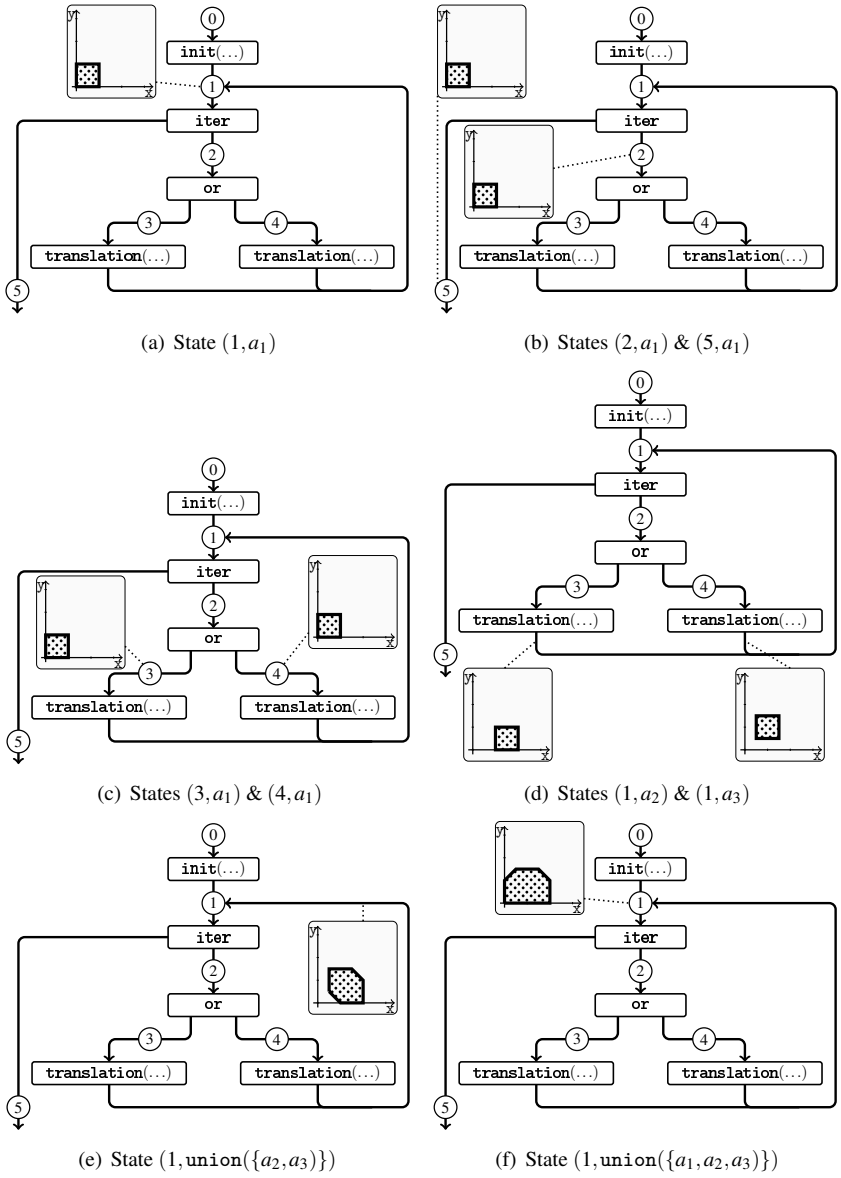
$a_3 =$ 

Figure 2.24 shows the snapshots of computing the iterates C_i *on top of the graph view of the program:*

- *Figure 2.24(a), Figure 2.24(b), and Figure 2.24(c), show the pre-condition* a_1 *at statement labels 1, 2, 3, 4, and 5 until* C_3.
- *Figure 2.24(d), Figure 2.24(e), and Figure 2.24(f) are snapshots of computing* C_4 *from* C_3.
- *Figure 2.24(d) shows two new pre-conditions of statement 1 (post-conditions after statement 3 and 4) resulting from* $Step^\sharp(C_3)$. *They will be "unioned" with other pre-conditions at statement 1.*
- *Figure 2.24(e) shows the result of unioning* a_2 *and* a_3 *during* $\texttt{union}\{a_1,a_2,a_3\}$ *at statement 1.*
- *Figure 2.24(f) shows the final result of* $\texttt{union}\{a_1,a_2,a_3\}$, union *of the above and* a_1.

Analysis algorithm with the termination guarantee. Now, note that the previous $\texttt{analysis}_T$ algorithm does not guarantee termination. If a program has a loop, the analysis may iterate



(a) State $(1, a_1)$

(b) States $(2, a_1)$ & $(5, a_1)$

(c) States $(3, a_1)$ & $(4, a_1)$

(d) States $(1, a_2)$ & $(1, a_3)$

(e) State $(1, \texttt{union}(\{a_2, a_3\}))$

(f) State $(1, \texttt{union}(\{a_1, a_2, a_3\}))$

Figure 2.24
Abstract transition snapshots in the graph view of the program

forever collecting ever new abstract pre-conditions. In order to guarantee the termination, we need to use the widening idea that we introduced to analyze the iteration statement in Section 2.3.4.

A terminating analysis $\texttt{analysis}_T$ should use a widening operation in place of the $\texttt{union}_T$ operation:

$$\texttt{analysis}_T(\texttt{p},I) \quad = \quad \begin{cases} \texttt{C} \leftarrow \{I\} \\ \text{repeat} \\ \qquad \texttt{R} \leftarrow \texttt{C} \\ \qquad \texttt{C} \leftarrow \texttt{widen}_T(\texttt{C}, \mathit{Step}^\sharp(\texttt{C})) \\ \text{until } \texttt{inclusion}_T(\texttt{C},\texttt{R}) \\ \text{return } \texttt{R} \end{cases}$$

The $\texttt{widen}_T$ operator ensures the termination of the sequence of iterations. It makes sure the number of collected constraints for abstract pre-conditions will always decrease.

The $\texttt{widen}_T$ function is identical to the $\texttt{union}_T$ operation except that at the **iter** statements, we use the `widen` operator in place of the `union` operator. It is because the **iter** statement is the only place where iteration happens during program execution. At other statements, we use the `union` operator as before:

$$\texttt{widen}_T(\texttt{C},\texttt{C}') \quad = \quad \begin{cases} \texttt{X} \leftarrow \{\} \\ \text{for each label } l \text{ in } \texttt{C} \cup \texttt{C}' \\ \qquad \texttt{S}_l \leftarrow \{a \mid (l,a) \in \texttt{C}\} \\ \qquad \texttt{S}'_l \leftarrow \{a \mid (l,a) \in \texttt{C}'\} \\ \qquad \texttt{T}_l \leftarrow \text{if the statement at } l \text{ is } \textbf{iter} \\ \qquad\qquad \text{then } \texttt{widen}(\texttt{union}(\texttt{S}_l), \texttt{union}(\texttt{S}'_l)) \\ \qquad\qquad \text{else } \texttt{union}(\texttt{S}_l \cup \texttt{S}'_l) \\ \qquad \texttt{X} \leftarrow \texttt{X} \cup \{(l, \texttt{T}_l)\} \\ \text{return } \texttt{X} \end{cases}$$

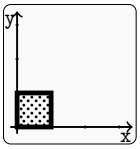
Note that, as opposed to the `union` operator, the `widen` operator is sensitive to the order of its arguments. The $\texttt{widen}(a,a')$ extrapolates a by a' as defined in Section 2.3.4. When either argument is **false** (the abstract pre-condition for the empty set of points in the two-dimensional plane), the widen operation simply returns the other argument.

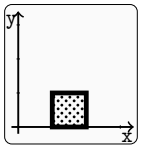
Example 2.19 (Abstract transitions with widening) *Consider again the example program in Figure 2.20 and suppose we use the convex-polyhedra abstractions for sets of points. The above widening analysis algorithm* $\texttt{analysis}_T$ *stores the following iterates* C_i *in the variable* `C` *after i iterations*

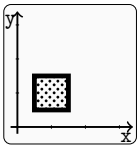
as before but using $\texttt{widen}_T$ *in place of* $\texttt{union}_T$*:*

$$\begin{array}{lll}
\textit{after 0 iteration,} & C_0 \stackrel{let}{=} & \{I\} \\
\textit{after 1 iteration,} & C_1 \stackrel{let}{=} & \texttt{widen}_T(C_0, \{(1,a_1)\}) \\
\textit{after 2 iterations,} & C_2 \stackrel{let}{=} & \texttt{widen}_T(C_1, \{(2,a_1),(5,a_1)\}) \\
\textit{after 3 iterations,} & C_3 \stackrel{let}{=} & \texttt{widen}_T(C_2, \{(3,a_1),(4,a_1)\}) \\
\textit{after 4 iterations,} & C_4 \stackrel{let}{=} & \texttt{widen}_T(C_3, \{(1,a_2),(1,a_3)\}) \\
 & \vdots & \vdots
\end{array}$$

where

$a_1 =$ 

$a_2 =$ 

$a_3 =$ 

The widening operation becomes effective at iteration 4 (C_4*), when the algorithm brings the effect of the loop body back to the loop head (statement label 1). Figure 2.24(e) shows the abstract state produced when the two results from the* **or** *statement in the loop body are unioned (*$\texttt{union}(\{a_2,a_3\})$*). The algorithm brings this result to the loop head and widens it with the old pre-condition (*a_1*). In the algorithm, this widening operation*

$$\texttt{widen}(a_1, \texttt{union}(\{a_2,a_3\}))$$

at the loop head happens during the computation of C_4 *at iteration 4:*

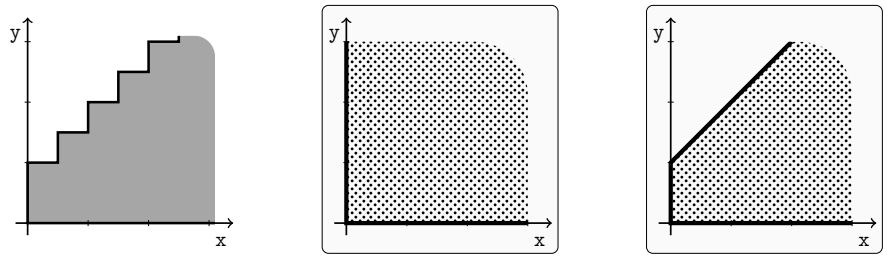
$$\texttt{widen}_T(C_3, \{(1,a_2),(1,a_3)\}).$$

The result corresponds to all the points such that $x \geq 0$ *and* $y \geq 0$*, as shown in Figure 2.25(b). It is a rather coarse over-approximation of the actual results, which is shown in Figure 2.25(a).*

The analysis accuracy can be improved by the "loop-unrolling" technique discussed in Example 2.14 (Section 2.3.4). This technique rewrites a loop "$\texttt{iter}\ \{b\}$*" into "*$\{\}\texttt{or}\{b\};\texttt{iter}\ \{b\}$*" before the analysis. The analysis of the unrolled, first iteration ("*$\{\}\texttt{or}\{b\}$*") will bring the unioned result to the subsequent loop head. For our example program the analysis result right after the unrolled first iteration is shown in Figure 2.24(f). Widening it at the subsequent loop head with the analysis result of the loop body will generate the result shown in Figure 2.25(c). This result is still an over-approximation of the reality, yet it is more accurate than the result shown in Figure 2.25(b).*

2.5 Core Principles of a Static Analysis

The previous sections sketched the design of static analyses in the context of a simplistic graphical language. First, in Section 2.1, we selected semantic properties of interest and formalized the semantics of programs, with respect to which these properties should be proved. Then in Section 2.2, we showed how to define abstractions of the standard se-



(a) Actual reachable set (b) A first approximation (c) Better approximation

Figure 2.25
Static analysis results

mantics of programs. Last, Section 2.3 and Section 2.4 presented two static analyses for this graphical language. In fact, both analyses were derived from a presentation of the semantics of programs (one in compositional style and one in transitional style).

This three-stage approach is actually general and has many fundamental and practical advantages, both for designing and for using static analysis tools.

Indeed, let us first recall the role of each stage:

1. **Selection of the semantics and properties of interest:**
   This stage is critical as it fixes the goal of the analysis. It describes the behaviors of programs and the properties that need to be verified. This description is often formal, even though we did it with prose in this chapter.
2. **Choice of the abstraction:**
   The abstraction describes the properties that are supposed to be manipulated by the analysis. These properties should be strong enough to express the properties of interest and all the invariants that are required to infer these properties.
3. **Derivation of the analysis algorithms from the semantics and from the abstraction:**
   The analysis algorithms follow from the choices made in the first two phases for the semantics and for the abstraction. In the two analyses presented in this chapter, we have observed that the analysis closely follows the semantics: for instance, the compositional analysis (Section 2.3) follows similar steps as a basic program interpreter, in the same order, but using abstract domain predicates instead of regular states.

From the static analysis point of view, this approach puts the choice of the reference semantics and property of interest at the forefront of the design process, as it should be, since this semantics and property define the actual goal of the analysis. It also addresses the selection of the predicates to use before the design of the algorithms to compute these predicates, although it does not preclude from revising these choices after testing the analysis, as discussed at the end of this section.

As observed in Section 2.3 and Section 2.4, most of the choices related to the analysis algorithms are dictated by the abstraction and by the way programs get evaluated according to the concrete semantics. Therefore, this construction also allows justifying the soundness of the analysis step by step: indeed, whenever we defined the way a program construction should be handled by the analysis, we ensured that the analysis does not forget any program behavior, according to the abstraction. Thus, the mathematical proof of soundness follows the design of the analysis closely. We will discuss this more in Chapter 3.

Similarly, this approach also allows tying the analysis and the "standard" semantics of programs. It is actually possible to follow the same process when implementing a static analyzer, as we will show in Chapter 7.

Lastly, this methodology also simplifies the troubleshooting of the analysis when it falls short, either in terms of precision (ability to compute strong invariants and achieve the proof of the property of interest) or in terms of scalability (ability to cope with input programs that are large enough). In particular, let us consider the case where the analysis fails to prove the property of interest. Then after investigating the analysis results, the user can diagnose which step "went wrong":

- the first point to check is that the base semantics allows expressing all the steps needed to prove the property of interest and that the abstraction preserves them; indeed, if the abstraction throws important information away, there is no hope that the analysis algorithms will infer predicates that the abstraction cannot capture and will recover from the loss of precision;
- when the semantics and abstraction are strong enough, the imprecisions stem from the analysis algorithms, and one needs to identify which abstract operation (for instance, the computation of the abstract post-condition for some basic operations in the language or the computation of an over-approximation for union) discards important information away, which causes the analysis to fail.

When the analysis tool can be parameterized, the user can often remedy such issues by choosing settings carefully. We will discuss this point in more depth in Chapter 6.

4 A General Static Analysis Framework Based on a Transitional Semantics

Goal of chapter. In this chapter, we provide a formal introduction to static analysis by abstract interpretation in the transitional style. This framework is general and can be instantiated for different languages and different abstractions. We shows a step-by-step recipe for constructing a sound static analysis in this framework. Following the recipe will result in a sound static analysis. This soundness guarantee is summarized in theorems whose proofs are in appendix. We assume that the readers are already familiar with key concepts of static analysis in abstract interpretation. The intuition conveyed in Chapter 2 supports most of the contents of this framework. Understanding the concept of abstraction and being familiar with its formal notions of Section 3.2.1 are necessary.

Recommended reading: **[S]**, **[D]**, **[U]**.
We recommend this chapter to all readers since it defines the core concepts of static analysis and is fundamental to the understanding of most of the following chapters in the book. Readers less interested in the foundations may skip some parts of the analysis design, whereas readers who would like to fully understand how static analyses achieve sound results may want to read the proofs supplied in Appendix. Moreover, readers interested in implementation may also combine the reading of this chapter with that of Chapter 7.

Chapter outline: recipe for the construction of an abstract interpreter in a transitional-style semantics. We present a general framework of designing a sound static analysis by abstract interpretation in transitional style. The presentation of this framework is not bound to a particular programming language. We present the framework solely in the semantic level, without referring to the syntax of a specific target programming language. A transition-style semantics allows this parameterization.

1. in Section 4.1, we define semantics as state transitions and show such semantics for an example program snippet. We then present a recipe for defining the concrete state-transition semantics.
2. in Section 4.2, we present a recipe for defining an abstract state-transition semantics that is a sound upper-approximation of a concrete semantics as defined in Section 4.1.

3. in Section 4.3, we presents analysis algorithms that compute abstract state-transition semantics as defined in Section 4.2.
4. in Section 4.4, we fix a simple imperative language and illustrate a use of the recipes of Section 4.1 and Section 4.2 in defining a correct analysis.

4.1 Semantics as State Transitions

We use a transitional style approach in order to define semantics. In this style we define concrete and abstract semantics in the small-step operational semantics.

This style of semantics is handy for languages whose compositional semantics (also known as *denotational semantics*) is not obvious. For example, if the target programming language has dynamic jumps (such as function calls, local gotos, jump labels as values, non-local gotos, function pointers, functions as values, dynamic method dispatches, or exception raises) then defining its compositional semantics becomes a burden. With gotos, program may loop with an arbitrary portion of the program, not tamed to a particular construct such as the while-loop. Defining the compositional semantics for such language feature needs an advanced knowledge in programming language semantics. Transitional semantics (one style of *operational semantics*) on the other hand is free from the need to be compositional and is relatively easy to define, once we understand how programs operate.

The transitional style is also a good fit for the proof of the reachability property. For this property, the static analysis goal is to over-approximate the set of reachable states of the input program. This set is obvious in the transitional style because the semantics explicitly exposes all the intermediate states of program executions.

4.1.1 Concrete Semantics

We start from the concrete semantics. The concrete semantics of a programming language defines the run-time behaviors of its programs. Informally, the concrete semantics of a language is what programmers have in mind about the run-time behaviors of their programs when they program.

The transitional-style semantics of a program is defined as the set of all possible sequences of state transitions from the initial states. We write a concrete state transition as

$$s \hookrightarrow s'$$

A sequence

$$s_0 \hookrightarrow s_1 \hookrightarrow s_2 \hookrightarrow \cdots$$

of state transitions is the chain that links the transitions

$$s_0 \hookrightarrow s_1,\ s_1 \hookrightarrow s_2,\ \cdots.$$

A state $s \in \mathbb{S}$ of the program is a pair (l,m) of a program label l and the machine state m at that program label during execution. The program label denotes the part of the program that is to be executed next. The machine state is usually the memory state that contains the effect of the program's hitherto execution and a data for the program's continuation. For the example language of Chapter 2, a machine state is a point in the two-dimensional space because program's execution step transforms a point to another point. For conventional imperative languages with local blocks and function calls, the machine state would consist of a memory (a table from locations to storable values), an environment (a table from program variables to locations), and a continuation (a stack of return contexts – a return context is a program label and an environment to resume at the return of a function).

One step of the state transition relation

$$(l,m) \hookrightarrow (l',m')$$

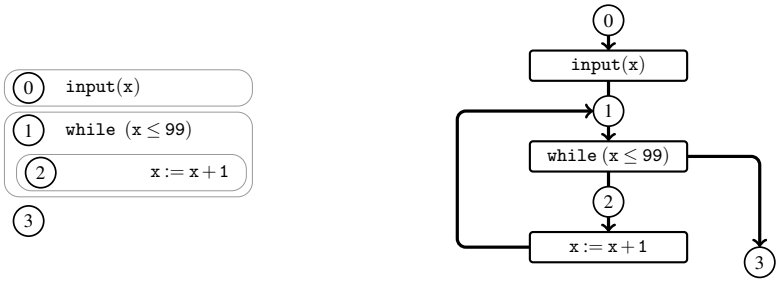
is defined by the language construct of the program part pointed to by l. The next memory state m' is the result of executing the program part at l by one step.

For simple languages, the next label l' of the program (called *control flow*) is determined by the program syntax. In case the control flow is not determined solely by the syntax but is determined by the program execution (such as goto target as values, function pointers, functions as values, or dynamic method dispatches) the next program label l' is an evaluation result from the current program label l and the current machine state m.

Example 4.1 (Concrete transition sequence) *Consider the following program*

```
input(x);
while (x ≤ 99)
        {x := x+1}
```

The labeled representations of this program in text and graph are respectively:



Let the initial state be the empty memory $\emptyset$. Transition sequences for some integer inputs are:

For input 100: $(0,\emptyset) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$.
For input 99: $(0,\emptyset) \hookrightarrow (1, x \mapsto 99) \hookrightarrow (2, x \mapsto 99) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$.
For input 0: $(0,\emptyset) \hookrightarrow (1, x \mapsto 0) \hookrightarrow (2, x \mapsto 0) \hookrightarrow (1, x \mapsto 1) \hookrightarrow \cdots \hookrightarrow (3, x \mapsto 100)$.

A transition sequence for a program can be infinitely long if the program has non-terminating executions. The number of transition sequences can be infinite too if the initial states can be infinitely many.

Set of reachable states. We restrict our analysis interest to computing the set of reachable states, the set of all states that can occur in the transition sequences of the input program.

Example 4.2 (Reachable states) *For the program in Example 4.1, let us assume that the possible inputs are only 0, 99, and 100. Then the set of all reachable states are the set of states occurring in the three transition sequences:*

$$\begin{array}{ll} & \{(0,\emptyset), (1, x \mapsto 100), (3, x \mapsto 100)\} \\ \cup & \{(0,\emptyset), (1, x \mapsto 99), (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\ \cup & \{(0,\emptyset), (1, x \mapsto 0), (2, x \mapsto 0), (1, x \mapsto 1), \cdots, (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\ = & \{(0,\emptyset), (1, x \mapsto 0), \cdots, (1, x \mapsto 100), (2, x \mapsto 0), \cdots, (2, x \mapsto 99), (3, x \mapsto 100)\} \end{array}$$

Given a program, the set of all its reachable states is intuitive in the operational sense. Starting from the set of all initial states of the program, we keep adding next states to the set. The next states are those produced by the application of the single-step transition $\hookrightarrow$ to each state in the current set. We keep adding next states until no more addition is possible. The final set is the set of all reachable states.

We will define this set of reachable states in mathematical terms. This mathematical formalization is a necessary step in our framework because it will later be a reference in proving the soundness of a designed static analysis. We will see that the mathematical concept called *the least fixpoint of a monotonic function* exactly defines the reachable set.

Given a program, let I be the set of its initial states and *Step* be the powerset-lifted version of $\hookrightarrow$:

$$\begin{array}{l} \mathit{Step} : \wp(\mathbb{S}) \to \wp(\mathbb{S}) \\ \mathit{Step}(X) = \{s' \mid s \hookrightarrow s', s \in X\} \end{array}$$

Note that the set of states that can occur right after i transitions from the set I of initial states is

$$\mathit{Step}^i(I)$$

where

$$\begin{array}{rcl} \mathit{Step}^0(X) & = & X \\ \mathit{Step}^{i+1}(X) & = & \mathit{Step}(\mathit{Step}^i(X)). \end{array}$$

Example 4.3 (*Step*^{i} operation) *For the program in Example 4.1, assuming the set $I = \{(0,\emptyset)\}$ of initial states, and that the possible inputs are 0, 99, and 100,*

$$\begin{array}{rcl} \mathit{Step}^0(I) & = & I \\ \mathit{Step}^1(I) & = & \{(1,x\mapsto 100),(1,x\mapsto 99),(1,x\mapsto 0)\} \\ \mathit{Step}^2(I) & = & \{(3,x\mapsto 100),(2,x\mapsto 99),(2,x\mapsto 0)\} \\ \mathit{Step}^3(I) & = & \{(1,x\mapsto 100),(1,x\mapsto 1)\} \\ \mathit{Step}^4(I) & = & \{(3,x\mapsto 100),(2,x\mapsto 1)\} \\ \mathit{Step}^5(I) & = & \{(1,x\mapsto 2)\} \\ \mathit{Step}^6(I) & = & \{(2,x\mapsto 2)\} \\ \mathit{Step}^7(I) & = & \{(1,x\mapsto 3)\} \\ & \vdots & \end{array}$$

Thus, the accumulated set of all reachable states of a program is the collection of $\mathit{Step}^i(I)$ for all $i \geq 0$:

$$I \cup \mathit{Step}^1(I) \cup \mathit{Step}^2(I) \cup \cdots. \tag{4.1}$$

We can define this set inductively as follows. Let C_i be the accumulated set $I \cup \mathit{Step}^1(I) \cup \cdots \cup \mathit{Step}^i(I)$ of reachable states in 0-to-i transition steps. Then C_i can be inductively defined as:

$$\begin{array}{rcl} C_0 & = & I \\ C_{i+1} & = & I \cup \mathit{Step}(C_i) \end{array}$$

The base C_0 is the initial set I. As of the inductive case, note that $\mathit{Step}(C_i)$ generates states occurring after one more step from C_i, that is, in 1-to-$(i+1)$ steps of transitions. Hence the set C_{i+1} of states in 0-to-$(i+1)$ steps of transitions is $I \cup \mathit{Step}(C_i)$.

The accumulated set (4.1) of all reachable states is the limit of the sequence $(C_i)_{i\in\mathbb{N}}$, a set C such that accumulating further by $I \cup \mathit{Step}(C)$ remains the same as C. That is, the limit is the least solution of the following equation:

$$X = I \cup \mathit{Step}(X).$$

Such limit corresponds, in mathematics, to the one called *the least fixpoint* of the continuous monotonic function F

$$\begin{array}{l} F : \wp(\mathbb{S}) \to \wp(\mathbb{S}) \\ F(X) = I \cup \mathit{Step}(X) \end{array}$$

written as

$$\mathbf{lfp}F.$$

The least fixpoint $\mathbf{lfp}F$ of F is constructive as follows:

Theorem 4.1 (Least fixpoint) *The least fixpoint* $\mathbf{lfp}F$ *of* $F(X) = I \cup Step(X)$ *is*

$$\bigcup_{i \geq 0} F^i(\emptyset)$$

where $F^0(X) = X \quad$ *and* $\quad F^{n+1}(X) = F(F^n(X))$.

The above theorem, which is a version of the Kleene fixpoint theorem (Theorem A.1), holds because the function $F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ is *continuous* over the powerset $\wp(\mathbb{S})$ with the set-inclusion order. (The readers may refer to Appendix A.5 for the definition of continuous functions. Intuitively, any function that can be exactly implemented as a computer program is continuous.)

Definition 4.1 (Concrete semantics, the set of reachable states) *Given a program, let* $\mathbb{S}$ *be the set of states and* $\hookrightarrow$ *be the one-step transition relation from a state to a state. Let* I *be the set of its initial states and Step be the powerset-lifted version of* $\hookrightarrow$*: one-step transition relation over states:*

$$Step : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}.$$

Let

$$F(X) = I \cup Step(X).$$

Then the concrete semantics of the program, the set of all reachable states from I*, is defined as the least fixpoint* $\mathbf{lfp}F$ *of* F.

4.1.2 Recipe for Defining a Concrete Transitional Semantics

When building a static analysis for programs written in a programming language L, the first step is to define its concrete semantics. The concrete semantics is the basis for later steps towards a static analysis.

1. For the target programming language, define the set of states between which a single-step transition relation $\hookrightarrow$ is to be defined. Let us name this set $\mathbb{S}$.
2. Define the $s \hookrightarrow s'$ relation between states s and $s' \in \mathbb{S}$ and let *Step* be its natural powerset-lifted version

$$Step : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}.$$

3. Given a program of the language with its set $I \subseteq \mathbb{S}$ of initial states, let

$$F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$F(X) = I \cup Step(X)$$

The concrete semantics, defined as the set of all the reachable states of the program, is the least fixpoint of the continuous function F:

$$\mathbf{lfp}F = \bigcup_{i \geq 0} F^i(\emptyset).$$

The concrete semantics is not what we implement as a static analyzer. Implementing this concrete semantics is rather equivalent to implementing an interpreter that actually runs the programs of the target language.

The next steps towards a static analysis consist of defining an abstract version of the concrete semantics and checking its soundness. These steps will reference the concrete semantics.

The concrete semantics as a mathematical object (as the least fixpoint) provides the foundation upon which static analysis design and its soundness check are conveniently formalized and proven. We will see that our intuition as sketched in Chapter 2 about sound static analysis and its algorithm have correspondences in mathematics.

Before we continue we add two definitions:

Definition 4.2 (Semantic domain and semantic function) *We assume the concrete semantics of a program by the least fixpoint of a function $F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$. Then we call the function F* concrete semantic function *and the space $\wp(\mathbb{S})$ over this semantic function is defined* concrete semantic domain *or simply* concrete domain *whose partial order is the subset order.*

4.2 Abstract Semantics as Abstract State Transitions

Given a concrete semantics, we now focus on the design of an abstract version that is finitely computable.

An abstract semantic functions $F^\sharp$ will have the same structure as the concrete semantic function F:

$$\begin{array}{ll} F : \wp(\mathbb{S}) \to \wp(\mathbb{S}) & F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \\ F(X) = I \cup \mathit{Step}(X) & F^\sharp(X^\sharp) = I^\sharp \cup^\sharp \mathit{Step}^\sharp(X^\sharp) \end{array}$$

where $I^\sharp$, $\cup^\sharp$, and $\mathit{Step}^\sharp$ are the abstract versions of, respectively, I, $\cup$, and Step. The concrete semantics of the target language consists of two parts: a semantic domain $\wp(\mathbb{S})$ and a semantic function over the domain $F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$. An abstract semantics also consists of two parts, an abstract domain $\mathbb{S}^\sharp$ and an abstract semantic function $F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ over it. The whole purpose of this abstract version is to derive a finitely computable, yet sound semantics for any program of the target language.

The forthcoming framework will guide us towards the definition of such an abstract domain $\mathbb{S}^\sharp$ and the abstract semantic function $F^\sharp$ such that the abstract semantics of the input program is always finitely computable and is an upper-approximation of the concrete semantics $\mathbf{lfp}F$.

4.2.1 Abstraction of the Semantic Domain

A concrete semantic domain $\mathbb{D}$ is the powerset of concrete states

$$\begin{aligned} \mathbb{D} &= \wp(\mathbb{S}) \\ \mathbb{S} &= \mathbb{L} \times \mathbb{M} \end{aligned}$$

The set $\mathbb{S}$ of concrete states is defined as the Cartesian product (set of pairs) of $\mathbb{L}$ (program labels) and $\mathbb{M}$ (machine states). The concrete semantics of a program, the set of all the reachable states of the program, is an element of the concrete domain $\mathbb{D}$. Given a program, the $\mathbb{L}$ set is defined to be its finite and fixed set of labels.

Program-label-wise reachability. In this chapter we will consider a class of abstractions that come from one particular analysis goal: program-label-wise reachability. We are interested in the reachable set for each program label. For each program label we want to know the set of memories that can occur at that label during executions. Such analysis is sometimes called *flow-sensitive*, because program labels are in this case assigned along the control flow of programs.

In this case, we can view the abstraction goes in two steps (Figure 4.1). We first partition the set of states by the program labels of the states

$$\begin{array}{ccc} \text{from collection of all states} & \text{to} & \text{label-wise collection} \\ \wp(\mathbb{L} \times \mathbb{M}) & \overset{\text{abstraction}}{\longrightarrow} & \mathbb{L} \to \wp(\mathbb{M}) \end{array}$$

then we abstract the local set of memories collected at each label into a single abstract memory

$$\begin{array}{ccc} \text{from label-wise collection} & \text{to} & \text{label-wise abstraction} \\ \mathbb{L} \to \wp(\mathbb{M}) & \overset{\text{abstraction}}{\longrightarrow} & \mathbb{L} \to \mathbb{M}^\sharp. \end{array}$$

An element of this abstract domain is a table from each program label to an abstract memory.

The program labels are usually syntactic elements, such as those assigned to every statement and/or expression of the program (as in Example 4.1 or Section 2.4). For any input program we thus assume that its label set $\mathbb{L}$ is finite and fixed before the analysis.

For this class of abstractions, what remains is to design the space $\mathbb{M}^\sharp$ of abstract memories. The $\mathbb{M}^\sharp$ is an abstraction of the powerset $\wp(\mathbb{M})$. This abstraction is the parameter and can be defined in many ways depending on the target properties to compute by static analysis.

Abstract domain by Galois connection. We first design an abstract domain, a space over which the abstract semantics of programs can be finitely computable.

In this chapter, an abstract domain is a partial order that has a least element called *bottom* (written $\bot$) and such that each totally-ordered subset (such a subset is called a *chain*) has a

$$\wp(\mathbb{L}\times\mathbb{M}) \ni \quad \text{collection of all states} \quad \begin{cases} (0,m_0),(0,m_0'),\cdots, & \text{at } 0\\ (1,m_1),(1,m_1'),\cdots, & \text{at } 1\\ \quad\vdots & \\ (n,m_n),(n,m_n'),\cdots. & \text{at } n \end{cases}$$

$$\mathbb{L}\to\wp(\mathbb{M}) \ni \quad \text{label-wise collection} \quad \begin{cases} (0,\{m_0,m_0',\cdots\})\\ (1,\{m_1,m_1',\cdots\})\\ \quad\vdots\\ (n,\{m_n,m_n',\cdots\}) \end{cases}$$

$$\mathbb{L}\to\mathbb{M}^\sharp \ni \quad \text{label-wise abstraction} \quad \begin{cases} (0,M_0^\sharp)\\ (1,M_1^\sharp)\\ \quad\vdots\\ (n,M_n^\sharp) \end{cases}$$

where each $M_l^\sharp$ over-approximates the set $\{m_l,m_l',\cdots\}$ of memories collected at label l.

Figure 4.1
Label-wise abstraction of states

least-upper bound. Such structure is called *CPO* (complete partial order) (Appendix A.5).

Note that different structures other than CPO can also be used. As in Chapter 3, abstract domains as $\sqcup$-semilattices also work. The generality of CPO and that of $\sqcup$-semilattice are not comparable.

An abstract domain needs to preserve the partial order of the concrete domain in the sense that the partial order in the abstract domain has a corresponding partial order in the concrete domain. This concept is captured by the Galois-connection between the two domains. Please note that other solutions rather than the Galois-connection would work too (for example, Section 3.3). In this Chapter we choose to use the Galois-connection approach.

We design an abstract domain as a CPO that is Galois-connected (Definition 3.5 in Section 3.2.1) with the concrete domain:

$$(\wp(\mathbb{L}\times\mathbb{M}),\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{L}\to\mathbb{M}^\sharp,\sqsubseteq).$$

The abstraction function α defines how each element (a set of states) in the concrete domain is abstracted into an element in the abstract domain. The adjoined concretization function γ defines the set of concrete states that is implied by each abstract state. The

partial order $\sqsubseteq$ is the label-wise order:

$$a^\sharp \sqsubseteq b^\sharp \quad \text{iff} \quad \forall l \in \mathbb{L} : a^\sharp(l) \sqsubseteq_M b^\sharp(l)$$

where $\sqsubseteq_M$ is the partial order of $\mathbb{M}^\sharp$.

The above Galois connection (abstraction) can be understood as the composition of two Galois connections:

$$\begin{array}{l}(\wp(\mathbb{L}\times\mathbb{M}),\subseteq)\\ \quad\underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}}\quad (\mathbb{L}\to\wp(\mathbb{M}),\sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \subseteq)\\ \qquad\underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (\mathbb{L}\to\mathbb{M}^\sharp,\sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \sqsubseteq_M)\end{array}$$

The first abstraction α_0 partitions the set of states by the labels of each state and collects the memories for each label:

$$\alpha_0\left\{\begin{array}{l}(0,m_0),(0,m_0'),\cdots,\\(1,m_1),(1,m_1'),\cdots,\\\quad\vdots\\(n,m_n),(n,m_n'),\cdots\end{array}\right\}=\left\{\begin{array}{l}(0,\{m_0,m_0',\cdots\}),\\(1,\{m_1,m_1',\cdots\}),\\\quad\vdots\\(n,\{m_n,m_n',\cdots\})\end{array}\right\}$$

This partitioning is a Galois connection.

What remains is to find a Galois connection pair α_1 and γ_1 for the second abstraction, which is a label-wise abstraction of the collected memory sets:

$$\alpha_1\left\{\begin{array}{l}(0,\{m_0,m_0',\cdots\}),\\(1,\{m_1,m_1',\cdots\}),\\\quad\vdots\\(n,\{m_n,m_n',\cdots\})\end{array}\right\}=\left\{\begin{array}{l}(0,M_0^\sharp),\\(1,M_1^\sharp),\\\quad\vdots\\(n,M_n^\sharp)\end{array}\right\}$$

Thus, defining this second Galois connection pair boils down to defining a Galois connection pair between the powerset of memories and an abstract memory domain:

$$(\wp(\mathbb{M}),\subseteq)\underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}}(\mathbb{M}^\sharp,\sqsubseteq_M).$$

Notations. Before we continue let us first brief notations to use in the rest of this chapter.

- An element of $A\to B$, which is a map from A to B, is interchangeably an element in $\wp(A\times B)$. For example, an element in $\mathbb{L}\to\mathbb{M}^\sharp$ of the abstract states is interchangeably an element in $\wp(\mathbb{L}\times\mathbb{M}^\sharp)$, a set (so called *graph*) of pairs of labels and abstract memories. Note that in the above examples of this subsection we already used this graph notation to represent the abstract state function.

- A relation $f \subseteq A \times B$ is interchangeably a function $f \in A \to \wp(B)$ defined as

$$f(a) = \{b \mid (a,b) \in f\}.$$

  For example, the concrete one-step transition relation $\hookrightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is interchangeably a function $\hookrightarrow \in \mathbb{S} \to \wp(\mathbb{S})$.
- For function $f : A \to B$, we write $\wp(f)$ for its powerset version defined as:

$$\begin{aligned}&\wp(f) : \wp(A) \to \wp(B)\\ &\wp(f)(X) = \{f(x) \mid x \in X\}\end{aligned}$$

- For function $f : A \to \wp(B)$, we write $\breve{\wp}(f)$ as a shorthand for $\cup \circ \wp(f)$

$$\begin{aligned}&\breve{\wp}(f) : \wp(A) \to \wp(B)\\ &\breve{\wp}(f)(X) = \bigcup\{f(x) \mid x \in X\}.\end{aligned}$$

  For example, powerset-lifted function $Step : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ of relation $\hookrightarrow$

$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\}$$

  is equivalently, by regarding $\hookrightarrow$ as a function of $\mathbb{S} \to \wp(\mathbb{S})$,

$$Step = \breve{\wp}(\hookrightarrow).$$

- For functions $f : A \to B$ and $g : A' \to B'$, we write (f,g) for

$$\begin{aligned}&(f,g) : A \times A' \to B \times B'\\ &(f,g)(a,a') = (f(a), g(a')).\end{aligned}$$

4.2.2 Abstraction of Semantic Functions

The abstract semantic function $F^\sharp$ over the abstract state is defined as follows.

Given a concrete semantic function F

$$\begin{aligned}&\mathbb{S} = \mathbb{L} \times \mathbb{M}\\ &F : \wp(\mathbb{S}) \to \wp(\mathbb{S})\\ &F(X) = I \cup Step(X)\end{aligned}$$

where

$$\begin{array}{lcl} Step & = & \breve{\wp}(\hookrightarrow) \quad \text{(relation } \hookrightarrow \text{ as a function)}\\ \hookrightarrow & \subseteq & (\mathbb{L} \times \mathbb{M}) \times (\mathbb{L} \times \mathbb{M}),\end{array}$$

its abstract version is defined to be

$$\begin{aligned}&\mathbb{S}^\sharp = \mathbb{L} \to \mathbb{M}^\sharp\\ &F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp\\ &F^\sharp(X^\sharp) = \alpha(I) \cup^\sharp Step^\sharp(X^\sharp)\end{aligned}$$

where

$$\begin{array}{rcll} \mathit{Step}^\sharp & = & \wp(\mathrm{id}, \sqcup_M) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp) & \text{(relation } \hookrightarrow^\sharp \text{ as a function)} \\ \hookrightarrow^\sharp & \subseteq & (\mathbb{L} \times \mathbb{M}^\sharp) \times (\mathbb{L} \times \mathbb{M}^\sharp). & \end{array} \tag{4.2}$$

The $\mathit{Step}^\sharp$ function is the one-step transition function over the abstract states $\mathbb{L} \to \mathbb{M}^\sharp$. The reason for its definition (4.2) is as follows:

- $\breve{\wp}(\hookrightarrow^\sharp)$: to an abstract state in $\mathbb{L} \to \mathbb{M}^\sharp$ as a set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$ it applies the abstract transition function $\hookrightarrow^\sharp$ to each element and collect the results, returning a set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$.
- $\pi \circ \breve{\wp}(\hookrightarrow^\sharp)$: the operator π partitions the result $\subseteq \mathbb{L} \times \mathbb{M}$ of $\breve{\wp}(\hookrightarrow^\sharp)$ by the labels in $\mathbb{L}$, returning a set $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp)$ where each label has only one pair, a set representation of an element in $\mathbb{L} \to \wp(\mathbb{M}^\sharp)$.
- $\wp(\mathrm{id}, \sqcup_M) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp)$: to the result $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp)$ of $\pi \circ \breve{\wp}(\hookrightarrow^\sharp)$ applying $\wp(\mathrm{id}, \sqcup_M)$ returns a set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$, in effect an abstract state $\in \mathbb{L} \to \mathbb{M}^\sharp$, so that each label is to be paired with a single abstract memory. This single abstract memory is the least-upper-bound $\sqcup_M$ of the set of abstract memories at each label.

Example 4.4 *Suppose the program has two labels l_1 and l_2. That is, $\mathbb{L} = \{l_1, l_2\}$. Given an abstract state $\{(l_1, M_1^\sharp), (l_2, M_2^\sharp)\}$, $\mathit{Step}^\sharp$ first applies $\breve{\wp}(\hookrightarrow^\sharp)$ to it:*

$$\hookrightarrow^\sharp(l_1, M_1^\sharp) \cup \hookrightarrow^\sharp(l_2, M_2^\sharp).$$

Suppose $\hookrightarrow^\sharp(l_1, M_1^\sharp)$ returns $\{(l_1, M'^\sharp_1), (l_2, M''^\sharp_1)\}$ and $\hookrightarrow^\sharp(l_2, M_2^\sharp)$ returns $\{(l_1, M'^\sharp_2)\}$. Then the result is

$$\{(l_1, M'^\sharp_1), (l_2, M''^\sharp_1), (l_1, M'^\sharp_2)\}.$$

The subsequent application of the operator π partitions the result by labels into

$$\{(l_1, \{M'^\sharp_1, M'^\sharp_2\}), (l_2, \{M''^\sharp_1\})\}.$$

The final organization operation $\wp(\mathrm{id}, \sqcup_M)$ returns the post abstract state $\in \mathbb{L} \to \mathbb{M}^\sharp$:

$$\{(l_1, M'^\sharp_1 \sqcup_M M'^\sharp_2), (l_2, M''^\sharp_1)\}.$$

Conditions for sound $\hookrightarrow^\sharp$ and $\cup^\sharp$. The abstract one-step transition relation $\hookrightarrow^\sharp$ must satisfy, as a function,

$$\breve{\wp}(\hookrightarrow) \circ \gamma \subseteq \gamma \circ \breve{\wp}(\hookrightarrow^\sharp).$$

Figure 4.2 depicts the above condition in a diagram. The condition is natural: the abstract one-step transition relation $\hookrightarrow^\sharp$ must cover the cases of the corresponding concrete one-step relation $\hookrightarrow$ in the concrete semantics. Note that abstract state $X^\sharp \in \mathbb{L} \to \mathbb{M}^\sharp$ is considered a set $\in \mathbb{L} \times \mathbb{M}^\sharp$ as the argument for $\breve{\wp}(\hookrightarrow^\sharp)$.

In the same vein, the condition for $\cup^\sharp$ to be sound is

$$\cup \circ (\gamma, \gamma) \subseteq \gamma \circ \cup^\sharp.$$

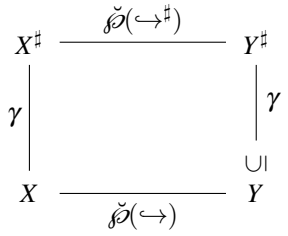


Figure 4.2
Sound one-step abstract transition $\hookrightarrow^\sharp$

All the above postulates (concrete semantic domains, concrete semantic function F, Galois-connected abstract domains, abstract semantic function $F^\sharp$, sound $\hookrightarrow^\sharp$, and sound $\cup^\sharp$) contributes to the correctness of the resulting abstract semantics.

4.2.3 Recipe for Defining an Abstract Transition Semantics

In summary, the recipe of achieving a sound static analysis based on the transitional semantics is as follows. Such static analysis over-approximates the concrete semantics of the input programs.

1. Define $\mathbb{M}$ to be the set of memory states that can occur during program executions. Let $\mathbb{L}$ be the finite and fixed set of labels of a given program.
2. Define a concrete semantics as the $\mathbf{lfp}F$ where

$$\begin{array}{rll}
\text{concrete domain} & \wp(\mathbb{S}) = \wp(\mathbb{L}\times\mathbb{M}) \\
\text{concrete semantic function} & F:\wp(\mathbb{S})\to\wp(\mathbb{S}) \\
& F(X) = I\cup \mathit{Step}(X) \\
& \mathit{Step} = \breve{\wp}(\hookrightarrow) \\
& \hookrightarrow \subseteq (\mathbb{L}\times\mathbb{M})\times(\mathbb{L}\times\mathbb{M})
\end{array}$$

   The $\hookrightarrow$ is the one-step transition relation over $\mathbb{L}\times\mathbb{M}$.
3. Define its abstract domain and abstract semantic function as

$$\begin{array}{rll}
\text{abstract domain} & \mathbb{S}^\sharp = \mathbb{L}\to\mathbb{M}^\sharp \\
\text{abstract semantic function} & F^\sharp:\mathbb{S}^\sharp\to\mathbb{S}^\sharp \\
& F^\sharp(X^\sharp) = \alpha(I)\cup^\sharp \mathit{Step}^\sharp(X^\sharp) \\
& \mathit{Step}^\sharp = \wp(\mathrm{id},\sqcup_M)\circ\pi\circ\breve{\wp}(\hookrightarrow^\sharp) \\
& \hookrightarrow^\sharp \subseteq (\mathbb{L}\times\mathbb{M}^\sharp)\times(\mathbb{L}\times\mathbb{M}^\sharp)
\end{array}$$

The $\hookrightarrow^\sharp$ is the one-step abstract transition relation over $\mathbb{L} \times \mathbb{M}^\sharp$. Function π partitions a set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$ by the labels in $\mathbb{L}$ returning an element in $\mathbb{L} \to \wp(\mathbb{M}^\sharp)$ represented as a set $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp)$.

4. Check the abstract domains $\mathbb{S}^\sharp$ and $\mathbb{M}^\sharp$ are CPOs, and forms a Galois-connection respectively with $\wp(\mathbb{S})$ and $\wp(\mathbb{M})$:

$$(\wp(\mathbb{S}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{S}^\sharp, \sqsubseteq) \quad \text{and} \quad (\wp(\mathbb{M}), \subseteq) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

where the partial order $\sqsubseteq$ of $\mathbb{S}^\sharp$ is label-wise $\sqsubseteq_M$:

$$a^\sharp \sqsubseteq b^\sharp \quad \text{iff} \quad \forall l \in \mathbb{L} : a^\sharp(l) \sqsubseteq_M b^\sharp(l).$$

5. Check the abstract one-step transition $\hookrightarrow^\sharp$ and abstract union $\cup^\sharp$ satisfy:

$$\begin{array}{rcl} \breve{\wp}(\hookrightarrow) \circ \gamma & \subseteq & \gamma \circ \breve{\wp}(\hookrightarrow^\sharp) \\ \cup \circ (\gamma, \gamma) & \subseteq & \gamma \circ \cup^\sharp \end{array}$$

6. Then sound static analysis can be defined as follows:
   (a) (Theorem 4.2) In case $\mathbb{S}^\sharp$ is of finite-height (every its chain is finite) and $F^\sharp$ is monotone or extensive, then
   $$\bigsqcup_{i \geq 0} {F^\sharp}^i(\bot)$$
   is finitely computable and over-approximates the concrete semantics $\mathbf{lfp}F$.
   (b) (Theorem 4.3) Otherwise, find a widening operator $\bigtriangledown$ (Definition 3.11), then the following chain $X_0 \sqsubseteq X_1 \sqsubseteq \cdots$
   $$X_0 = \bot \qquad X_{i+1} = X_i \bigtriangledown F^\sharp(X_i)$$
   is finite and its last element over-approximates the concrete semantics $\mathbf{lfp}F$.

Theorem 4.2 (Sound static analysis by $F^\sharp$) *Given a program, let F and $F^\sharp$ be defined as in Section 4.2.3. If $\mathbb{S}^\sharp$ is of finite-height (every chain $\mathbb{S}^\sharp$ is finite) and $F^\sharp$ is monotone or extensive, then*

$$\bigsqcup_{i \geq 0} {F^\sharp}^i(\bot)$$

is finitely computable and over-approximates $\mathbf{lfp}F$*:*

$$\mathbf{lfp}F \subseteq \gamma(\bigsqcup_{i \geq 0} {F^\sharp}^i(\bot)) \quad \textit{or equivalently} \quad \alpha(\mathbf{lfp}F) \sqsubseteq \bigsqcup_{i \geq 0} {F^\sharp}^i(\bot).$$

The proof of the above theorem is in Appendix B.3.1.

In case the abstract domain may have an infinite chain or the abstract semantic function $F^\sharp$ can be neither monotone nor extensive, by means of a special operator called *widening operator* we can still finitely compute an upper approximation of the concrete semantics of programs:

Theorem 4.3 (Sound static analysis by $F^\sharp$ and widening operator $\triangledown$) *Given a program, let F and $F^\sharp$ be defined as in Section 4.2.3. Let $\triangledown$ be a widening operator as defined in Definition 3.11 (page 87).*

Then the following chain $Y_0 \sqsubseteq Y_1 \sqsubseteq \cdots$

$$Y_0 = \bot \qquad Y_{i+1} = Y_i \triangledown F^\sharp(Y_i)$$

is finite and its last element Y_{lim} over-approximates $\mathbf{lfp}F$*:*

$$\mathbf{lfp}F \subseteq \gamma(Y_{\text{lim}}) \quad \textit{or equivalently} \quad \alpha(\mathbf{lfp}F) \sqsubseteq Y_{\text{lim}}.$$

The proof of the above theorem is in Appendix B.3.2.

Note that if a widening operator is used with a monotone or extensive $F^\sharp$, the second condition for a widening operator in Definition 3.11 can be relaxed from "for all sequence $(a_n)_{n\in\mathbb{N}}$" to "for all chain $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$".

Using the results of the analysis. Like Theorem 3.6 (page 90) for compositional-style abstract semantics, the above two soundness theorems formalize what the analysis achieves: the analysis computes an over-approximation of *all* the states that the program may reach during its executions.

The same discussion we had after Theorem 3.6 regarding the use of a sound analysis in practice applies here too. If the over-approximate reachable set does not intersect with the set of error states, then we are sure that the program will not generate an error state. Otherwise, we cannot conclude anything. A non-empty intersection may be due to an imprecision of the analysis, or to the fact that the program can indeed generate an error state. Upon this unsettled case, the analysis has to generate *alarms* so that users should inspect the analysis results so as to decide whether the alarms are true or not. This so-called *triage* process is discussed in details in Section 6.3.

4.3 Analysis Algorithms Based on Global Iterations

4.3.1 Basic Algorithms

Once we have designed an abstract semantics function $F^\sharp$ as in the recipe (Section 4.2.3), the analysis implementation is straightforward from Theorem 4.2 and Theorem 4.3.

Algorithm from Theorem 4.2. If the abstract domain $\mathbb{S}^\sharp$ is of finite-height and $F^\sharp$ is monotone or extensive, the increasing chain

$$\bot \sqsubseteq (F^\sharp)^1(\bot) \sqsubseteq (F^\sharp)^2(\bot) \sqsubseteq \cdots$$

is finite and its biggest element is equal to

$$\bigsqcup_{i\geq 0} F^{\sharp i}(\bot).$$

$$
\begin{array}{l}
\mathtt{C} \leftarrow \bot \\
\text{repeat} \\
\quad \mathtt{R} \leftarrow \mathtt{C} \\
\quad \mathtt{C} \leftarrow F^\sharp(\mathtt{C}) \\
\text{until } \mathtt{C} \sqsubseteq \mathtt{R} \\
\text{return } \mathtt{R}
\end{array}
$$

Figure 4.3
Algorithm without widening

Hence, the analysis algorithm is a simple loop, shown in Figure 4.3.

As a side note, the algorithms in the gentle introduction Section 2.4.4 are based on the following algorithm that computes an upper bound of $\bigsqcup_{i\geq 0} F^{\sharp i}(\bot)$.

$$
\begin{array}{l}
\mathtt{C} \leftarrow \bot \\
\text{repeat} \\
\quad \mathtt{R} \leftarrow \mathtt{C} \\
\quad \mathtt{C} \leftarrow \mathtt{C} \sqcup F^\sharp(\mathtt{C}) \\
\text{until } \mathtt{C} \sqsubseteq \mathtt{R} \\
\text{return } \mathtt{R}
\end{array}
$$

Note that this algorithm computes a chain

$$Y_0 = \bot \qquad Y_{i+1} = Y_i \sqcup F^\sharp(Y_i)$$

such that every element Y_i is an upper bound of its corresponding element X_i of the first algorithm

$$X_0 = \bot \qquad X_{i+1} = F^\sharp(X_i).$$

Algorithm from Theorem 4.3. If the abstract domain $\mathbb{S}^\sharp$ is of infinite-height or $F^\sharp$ is neither monotonic nor extensive, we have to use a widening operator $\bigtriangledown$. Even when the abstract domain $\mathbb{S}^\sharp$ is of finite-height, we can accelerate the analysis steps by using the widening operator.

Given a program, the analysis algorithm with $\bigtriangledown$ is shown in Figure 4.4. The algorithm computes a finite increasing chain as ensured in Theorem 4.3.

4.3.2 Worklist Algorithm

The basic iteration algorithms of Section 4.3.1 have a room for speedup. Let us re-consider the widening version with the operation $F^\sharp(\mathtt{C})$ being inlined in order to expose its perfor-

$$
\begin{array}{l}
\mathtt{C} \leftarrow \bot \\
\text{repeat} \\
\quad \mathtt{R} \leftarrow \mathtt{C} \\
\quad \mathtt{C} \leftarrow \mathtt{C} \bigtriangledown F^\sharp(\mathtt{C}) \\
\text{until } \mathtt{C} \sqsubseteq \mathtt{R} \\
\text{return } \mathtt{R}
\end{array}
$$

Figure 4.4
Algorithm with $\bigtriangledown$

mance bottleneck:

$$
\begin{array}{l}
\mathtt{C} \leftarrow \bot \\
\text{repeat} \\
\quad \mathtt{R} \leftarrow \mathtt{C} \\
\quad \mathtt{C} \leftarrow \mathtt{C} \bigtriangledown \underbrace{(\wp(\mathrm{id}, \sqcup) \circ \pi \circ \check{\wp}(\hookrightarrow^\sharp))}_{F^\sharp}(\mathtt{C}) \\
\text{until } \mathtt{C} \sqsubseteq \mathtt{R} \\
\text{return } \mathtt{R}
\end{array}
$$

Note that at each iteration, computing

$$\check{\wp}(\hookrightarrow^\sharp)(\mathtt{C})$$

applies the abstract transition operation $\hookrightarrow^\sharp$ to the state at every label in the program. The table C has as many entries as the number of labels in the program. When every statement of the program is uniquely labeled, one million-statement program has one million labels.

We can speedup the performance by reducing the program labels to visit at each iteration. We keep a worklist, a set of labels for which the transition needs to be applied because its input memories were changed. For each iteration, the transition is applied only for those labels in the worklist. The worklist for the next iteration becomes to consist of the labels whose input memories are changed in the previous iteration.

This worklist version is in Figure 4.5. As of notation, for the table C from all labels in the program to their abstract memories, $\mathtt{C}|_{\mathtt{WorkList}}$ is the same as the C table but restricted only for labels in WorkList.

This worklist algorithm calls for improvements regarding two specific points:

- Note that collecting the new worklist for next iteration

$$\mathtt{WorkList} \leftarrow \{l \mid \mathtt{C}(l) \not\sqsubseteq \mathtt{R}(l), l \in \mathbb{L}\}$$

  re-scans all the labels of the program.

$\mathtt{C} : \mathbb{L} \to \mathbb{M}^\sharp$
$F^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp)$
$\mathtt{WorkList} : \wp(\mathbb{L})$

$\mathtt{WorkList} \leftarrow \mathbb{L}$
$\mathtt{C} \leftarrow \bot$
repeat
  $\mathtt{R} \leftarrow \mathtt{C}$
  $\mathtt{C} \leftarrow \mathtt{C} \bigtriangledown F^\sharp(\mathtt{C}|_{\mathtt{WorkList}})$ (* only for WorkList *)
  $\mathtt{WorkList} \leftarrow \{l \mid \mathtt{C}(l) \not\sqsubseteq \mathtt{R}(l), l \in \mathbb{L}\}$ (* next WorkList *)
until $\mathtt{WorkList} = \emptyset$
return R

Figure 4.5
Analysis algorithm with worklist and widening

We can avoid this scanning of all the labels. Right after each application $\hookrightarrow^\sharp$ to $(l, \mathtt{C}(l))$, if the result state $(l', M^\sharp)$ is changed ($M^\sharp \not\sqsubseteq \mathtt{C}(l')$), we add l' to the new worklist.

- Generally, the widening operator deteriorates the precision of resulting abstract elements, thus, they should be applied only at necessity. Naive implementation of the widening operation $\mathtt{C} \bigtriangledown F^\sharp(C|_{\mathtt{WorkList}})$ would be a squandering, label-wise widening:

  for every $(l, M^\sharp) \in F^\sharp(\mathtt{C}|_{\mathtt{WorkList}})$ we widen $\mathtt{C}(l)$ by $M^\sharp$.

  The precision would be better if we apply the widening operation only when the l is the target of a cycling control flow (for example, the l-labeled statement is a **`while`** statement or a target statement of a cycling **`goto`**)(**?**). For other labels, we apply the least-upper-bound operation $\cup^\sharp$ instead.

4.4 Use Example of the Framework

4.4.1 Simple Imperative Language

We consider a simple imperative language (Figure 4.6) that is almost identical to the one in the previous chapter on the compositional style framework. One difference, in order to expose the merit of the transitional style, is the goto statement whose target label is not fixed in the program text but to be computed during execution by its argument expression. Labels in a program are integers that are uniquely assigned to every statement of the pgoram; we assume that the programmers know the labels for each statement of their programs. The

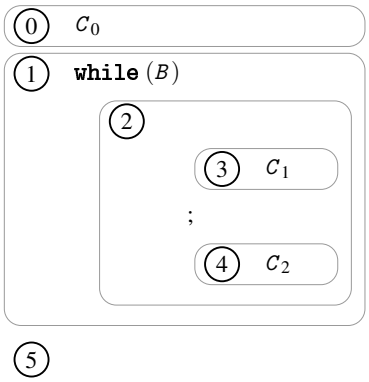
$\mathtt{x}$	$\in$	$\mathbb{X}$	program variables
C	$::=$		statements
	$\mid$	**skip**	nop statement
	$\mid$	C ; C	sequence of statements
	$\mid$	$\mathtt{x} := E$	assignment
	$\mid$	**input**$(\mathtt{x})$	read an integer input
	$\mid$	**if**$(B)\{C\}$**else**$\{C\}$	condition statement
	$\mid$	**while**$(B)\{C\}$	loop statement
	$\mid$	**goto** E	goto with dynamically computed label
E	$::=$		expression
	$\mid$	n	integer
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$E + E$	addition
B	$::=$		boolean expression
	$\mid$	**true** $\mid$ **false**	
	$\mid$	$E < E$	comparison
	$\mid$	$E = E$	equality
P	$::=$	C	program

Figure 4.6
Syntax of a simple imperative language

expressions compute values without mutating the memory. An expression E computes an integer or a program label. A boolean expression B computes a boolean value.

Program labels and execution order. Given a program, each of its statements has a unique label as a natural number. For example, Figure 4.7 shows an example program where a unique label is associated to each statement.

Except for **goto** E, the execution order (or *control flow*) between the statements in a program is clear from the program syntax. The function $\langle\langle C, l' \rangle\rangle$ defined below collects all the execution orders available from syntax between the statement labels in C. The label l' is the next label to continue after executing C. Thus, given a program $\mathtt{p}$ and label l_{end} for the end label of the program, $\langle\langle \mathtt{p}, l_{\text{end}} \rangle\rangle$ collects the function graphs of `next`, `nextTrue`, and `nextFalse`. We write $\mathtt{label}(C)$ for the label of statement C.



$\texttt{next}(0) = 1$
$\texttt{nextTrue}(1) = 2 \quad \texttt{next}(2) = 3$
$\texttt{nextFalse}(1) = 5 \quad \texttt{next}(3) = 4$
$\texttt{next}(4) = 1$

Figure 4.7
Example program with statement labels and execution order. All labels are statically known.

$$
\begin{array}{rcl}
\langle\langle C, l' \rangle\rangle = \text{case } C \text{ of} & & \text{(* let } l \text{ be } \texttt{label}(C) \text{ *)} \\
\textbf{skip} & : & \{\texttt{next}(l) = l'\} \\
\texttt{x} := E & : & \{\texttt{next}(l) = l'\} \\
\textbf{input}(\texttt{x}) & : & \{\texttt{next}(l) = l'\} \\
C_1; C_2 & : & \{\texttt{next}(l) = \texttt{label}(C_1)\} \cup \langle\langle C_1, \texttt{label}(C_2) \rangle\rangle \cup \langle\langle C_2, l' \rangle\rangle \\
\textbf{if}(B)\{C_1\}\textbf{else}\{C_2\} & : & \{\texttt{nextTrue}(l) = \texttt{label}(C_1), \texttt{nextFalse}(l) = \texttt{label}(C_2)\} \\
 & & \cup \langle\langle C_1, l' \rangle\rangle \cup \langle\langle C_2, l' \rangle\rangle \\
\textbf{while}(B)\{C\} & : & \{\texttt{nextTrue}(l) = \texttt{label}(C), \texttt{nextFalse}(1) = l'\} \cup \langle\langle C, l \rangle\rangle \\
\textbf{goto}\, E & : & \{\} \quad \text{(* to be determined at run-time by evaluating } E \text{ *)}
\end{array}
$$

4.4.2 Concrete State Transition Semantics

Given a program p, let $\mathbb{X}$ be the finite set of its variables. Each statement of the program is uniquely labeled and we assume that the `next`, `nextTrue`, and `nextFalse` functions are syntactically computed beforehand by $\langle\langle \texttt{p}, l_{\text{end}} \rangle\rangle$, except for the goto case.

Then the concrete semantics of the program, for the set I of input states, is the least fixpoint

$$\mathbf{lfp} F$$

of the continuous function

$$F : \wp(\mathbb{S}) \to \wp(\mathbb{S})$$
$$F(X) = I \cup Step(X)$$
$$Step(X) = \breve{\wp}(\hookrightarrow).$$

The set of states is the set of label-and-memory pairs

$$\mathbb{S} = \mathbb{L} \times \mathbb{M}$$

where

$$\begin{array}{lrcl} \text{memories} & \mathbb{M} & = & \mathbb{X} \to \mathbb{V} \\ \text{values} & \mathbb{V} & = & \mathbb{Z} \cup \mathbb{L}. \end{array}$$

The state transition relation $(l,m) \hookrightarrow (l',m')$ is defined as follows. The transition relation is defined by case analysis on statement labeled by l:

$$\begin{array}{rcll} \textbf{skip} & : & (l,m) \hookrightarrow (\texttt{next}(l),\ m) & \\ \textbf{input}(\mathtt{x}) & : & (l,m) \hookrightarrow (\texttt{next}(l),\ update_{\mathtt{x}}(m,z)) & \text{for an input integer } z \\ \mathtt{x} := E & : & (l,m) \hookrightarrow (\texttt{next}(l),\ update_{\mathtt{x}}(m, eval_E(m))) & \\ C_1; C_2 & : & (l,m) \hookrightarrow (\texttt{next}(l),\ m) & \\ \textbf{if}(B)\{C_1\}\textbf{else}\{C_2\} & : & (l,m) \hookrightarrow (\texttt{nextTrue}(l), filter_B(m)) & \\ & : & (l,m) \hookrightarrow (\texttt{nextFalse}(l), filter_{\neg B}(m)) & \\ \textbf{while}(B)\{C\} & : & (l,m) \hookrightarrow (\texttt{nextTrue}(l), filter_B(m)) & \\ & : & (l,m) \hookrightarrow (\texttt{nextFalse}(l), filter_{\neg B}(m)) & \\ \textbf{goto}\ E & : & (l,m) \hookrightarrow (eval_E(m),\ m) & \end{array}$$

The memory update operation $update_{\mathtt{x}}(m,v)$ returns a new memory that is the same as m except that its image for $\mathtt{x}$ is v. The expression-evaluation operation $eval_E(m)$ returns a value of expression E given memory m. The $filter_B(m)$ (respectively, $filter_{\neg B}(m)$) operation returns m if the value of boolean expression E for m is true (respectively, false). Otherwise, no corresponding transition relation happens.

4.4.3 Abstract State

An abstract domain $\mathbb{M}^\sharp$ is a CPO such that

$$(\wp(\mathbb{M}), \subseteq) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} (\mathbb{M}^\sharp, \sqsubseteq_M).$$

We define an abstract memory $M^\sharp$ for a set of memories as a single map from program variables to abstract values:

$$M^\sharp \in \mathbb{M}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$$

where $\mathbb{V}^\sharp$ is an abstract domain that is a CPO such that

$$(\wp(\mathbb{V}), \subseteq) \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} (\mathbb{V}^\sharp, \sqsubseteq_V).$$

We design $\mathbb{V}^\sharp$ as

$$\mathbb{V}^\sharp = \mathbb{Z}^\sharp \times \mathbb{L}^\sharp$$

where $\mathbb{Z}^\sharp$ is a CPO that is Galois connected with $\wp(\mathbb{Z})$, and $\mathbb{L}^\sharp$ is the powerset $\wp(\mathbb{L})$ of labels.

All abstract domains are Galois-connected CPOs. Because $\mathbb{Z}^\sharp$ and $\mathbb{L}^\sharp$ are Galois-connected CPOs, so are the compound domains built from them: $\mathbb{V}^\sharp$ the component-wise-ordered pairs of $\mathbb{Z}^\sharp$ and $\mathbb{L}^\sharp$, $\mathbb{M}^\sharp$ the point-wise-ordered vectors of $\mathbb{V}^\sharp$, and $\mathbb{S}^\sharp$ the point-wise-ordered vectors of $\mathbb{M}^\sharp$.

4.4.4 Abstract State Transition Semantics

For an abstract memory $M^\sharp$, we define the abstract state transition relation $(l, M^\sharp) \hookrightarrow^\sharp (l', M^{\sharp\prime})$ as follows.

Case the l-labeled statement of

$$\begin{array}{rcl}
\textbf{skip} & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{next}(l), M^\sharp) \\
\textbf{input}(\texttt{x}) & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{next}(l), update^\sharp_{\texttt{x}}(M^\sharp, \alpha(\mathbb{Z}))) \\
\texttt{x} := E & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{next}(l), update^\sharp_{\texttt{x}}(M^\sharp, eval^\sharp_E(M^\sharp))) \\
C_1; C_2 & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{next}(l), M^\sharp) \\
\textbf{if}(B)\{C_1\}\textbf{else}\{C_2\} & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{nextTrue}(l), filter^\sharp{}_B(M^\sharp)) \\
 & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{nextFalse}(l), filter^\sharp{}_{\neg B}(M^\sharp)) \\
\textbf{while}(B)\{C\} & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{nextTrue}(l), filter^\sharp{}_B(M^\sharp)) \\
 & : & (l, M^\sharp) \hookrightarrow^\sharp (\texttt{nextFalse}(l), filter^\sharp{}_{\neg B}(M^\sharp)) \\
\textbf{goto}\ E & : & (l, M^\sharp) \hookrightarrow^\sharp (l', M^\sharp) \quad \text{for } l' \in L \text{ of } (z^\sharp, L) = eval^\sharp_E(M^\sharp)
\end{array}$$

Let $F^\sharp$ be defined as the framework:

$$\begin{array}{l}
F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp \\
F^\sharp(S^\sharp) = \alpha(I) \cup^\sharp Step^\sharp(S^\sharp) \\
Step^\sharp = \wp(\mathrm{id}, \sqcup_M) \circ \pi \circ \check{\wp}(\hookrightarrow^\sharp).
\end{array}$$

If the $Step^\sharp$ and $\cup^\sharp$ are sound abstractions of, respectively, $Step$ and $\cup$, as required by the framework:

$$\begin{array}{rcl}
\check{\wp}(\hookrightarrow) \circ \gamma & \subseteq & \gamma \circ \check{\wp}(\hookrightarrow^\sharp) \\
\cup \circ (\gamma, \gamma) & \subseteq & \gamma \circ \cup^\sharp
\end{array}$$

then we can use $F^\sharp$ to soundly approximates the concrete semantics $\mathbf{lfp}F$ (Theorem 4.2 and Theorem 4.3) and is finitely computable as in the algorithms in Section 4.3.

Defining sound $\hookrightarrow^\sharp$. Note that the abstract transition relation $\hookrightarrow^\sharp$ is the same as $\hookrightarrow$ except that it uses the abstract correspondents for semantic operators: operator $eval^\sharp{}_E$ for $eval_E$, $update^\sharp{}_{\mathtt{x}}$ for $update_{\mathtt{x}}$, $filter^\sharp{}_B$ for $filter_B$, and $filter^\sharp{}_{\neg B}$ for $filter_{\neg B}$.

If each of the abstract semantic operators is a sound abstraction of its concrete correspondent, then $\hookrightarrow^\sharp$ is a sound abstraction of $\hookrightarrow$:

Theorem 4.4 (Soundness of $\hookrightarrow^\sharp$) *Consider the concrete one-step transition relation of Section 4.4.2 and the abstract transition relation of Section 4.4.4. If the semantic operators satisfy the following soundness properties:*

$$\begin{array}{rcl} \wp(eval_E)\circ\gamma_M & \subseteq & \gamma_V\circ eval^\sharp_E \\ \wp(update_{\mathtt{x}})\circ\times\circ(\gamma_M,\gamma_V) & \subseteq & \gamma_M\circ update^\sharp_{\mathtt{x}} \\ \wp(filter_B)\circ\gamma_M & \subseteq & \gamma_M\circ filter^\sharp_B \\ \wp(filter_{\neg B})\circ\gamma_M & \subseteq & \gamma_M\circ filter^\sharp_{\neg B} \end{array}$$

then $\check{\wp}(\hookrightarrow)\circ\gamma \sqsubseteq \gamma\circ\check{\wp}(\hookrightarrow^\sharp)$. (The $\times$ is the Cartesian product operator of two sets.)

The proof is included in Appendix B.3.

Defining sound $\cup^\sharp$. As of a sound $\cup^\sharp$, one candidate is the least upper bound operator $\sqcup$ if $\mathbb{S}^\sharp$ is closed by $\sqcup$, because

$$\begin{array}{rcll} (\gamma\circ\sqcup)(a^\sharp,b^\sharp) \;=\; \gamma(a^\sharp\sqcup b^\sharp) & \sqsupseteq & \gamma(a^\sharp)\cup\gamma(b^\sharp) & \text{by the monotonicity of } \gamma \\ & = & (\cup\circ(\gamma,\gamma))(a^\sharp,b^\sharp). & \end{array}$$

Adapting the analysis for a different abstraction. The above soundness theorem Theorem 4.4 also suggests a way to adapt the analysis for a different abstraction. We are to follow the recipe of Section 4.2.3, and for a different analysis we need to use different abstract domains and semantic operators. The new analysis is sound if the abstract transition relation $\hookrightarrow^\sharp$ is the same as ("homomorphic to") $\hookrightarrow$ except that it uses the abstract correspondents for semantic operators, and that each abstract semantic operator $f^\sharp : A^\sharp \to B^\sharp$ satisfies the pattern $\wp(f)\circ\gamma_A \subseteq \gamma_B\circ f^\sharp$.

In fact, many static analysis tools are parametric in the choice of the abstract domains and semantic operators. The practical consideration related to the parameter setting of the analysis are discussed in Section 6.2.

8 Static Analysis for Advanced Programming Features

Goal of the chapter. This chapter discusses the analysis of more realistic programming languages than those considered so far. Indeed, previous chapters focus on the foundations of static analysis, hence opt for a somewhat contrived language. We now aim at considering full-featured languages.

Real-world programming languages feature arrays, pointers, dynamic memory allocation, functions and other constructions than the basic arithmetic and boolean operators considered so far. This chapter provides a high level view of the static analysis techniques for such programming languages features. As the range of these features is quasi infinite (for example, exceptions, multi-staged meta programming, parameterized modules, dynamic dispatches, and etc.), and the academic literature on their analysis is also very wide and thorough, we do not aim for an exhaustive coverage. Instead, we consider a few common and representative constructions, and present the most significant techniques that cope with it. For each construction, we summarize the concrete semantics at a high level, discuss the salient properties related to static analysis and present the main abstractions. We do not provide in-depth discussion of the analysis algorithms (though, we give the underlying intuitions), as it would go beyond the scope of this introductory books (instead, we provide references). We stress the link between the properties of interest and the abstractions that are required to cope with them. Note that our approach benefits from the advantages of the abstract interpretation methodology as presented in previous chapters: indeed, we rely on the fact that the concrete semantics serves as a basis for the analysis design, and that the analysis algorithms derive from the abstraction, so that the choice of the semantics and of the abstraction is key.

Recommended reading: **[S]** (Master), **[D]** and **[U]** (2nd reading, depending on need). Engineers and students can skip this chapter in a first reading and revisit it when dealing with the analysis of software relying on more advanced programming features. Teachers will find here examples that can be used to illustrate classes and practical sessions.

Chapter outline.
Section 8.1 and Section 8.2 first discusses the step-by-step design of abstract interpreters

E	::=	$\cdots$	expression, as before (Figure 4.6, page 113)
	|	`&x`	location of a variable
	|	`malloc`	location of a newly allocated memory
	|	$*E$	dereference of a memory location
C	::=	$\cdots$	statement, as before (Figure 4.6, page 113)
	|	$*E := E$	indirect assignment
P	::=	C	program

Figure 8.1
Syntax of an example language with pointers and dynamic memory allocations

in the framework of Chapter 4 for two common dynamic programming features, namely pointers and functions. While doing so, we demonstrate the convenience of the semantics-based approach to static analysis. Indeed, the design of an analysis is driven by the concrete semantics and the abstraction. Therefore, the presence of pointers and procedures does not require fundamentally different frameworks to formalize, and prove the analysis correct, even though novel abstractions need to be used.

Subsequently, Section 8.3 and Section 8.4 focus more on the abstractions required to cope with complex programming languages. Section 8.3 and Section 8.4 describe a few common programming languages features, and corresponding basic abstractions. Features are treated separately, and abstractions are mostly presented at a high level, for the sake of concision. Section 8.3 focuses on features related to memory states, whereas Section 8.4 studies features related to control states.

8.1 For a Language with Pointers and Dynamic Memory Allocations

8.1.1 Language and Concrete Semantics

Let us consider an imperative language that admits memory locations as values. The syntax of this example language is shown in Figure 8.1. We extend the imperative language of Section 4.4 (Figure 4.6) with the constructs that generate and use memory locations.

A `malloc` expression allocates an isolated fresh memory and returns its address. We let the allocated size be always a unit size that can store integers, labels, or addresses. (A more realistic construct for the dynamic memory allocation will be considered in Section 8.3.4.) Dereference expression $*E$ computes a location value from E then returns the value stored at this location. Indirect assignment $*E_1 := E_2$ computes a location from E_1 and stores the value of E_2 into the location.

Example 8.1 *Consider the following program.*

$$\begin{aligned} &\mathtt{x} := \mathtt{malloc};\\ &\mathtt{y} := \mathtt{\&x};\\ *&\mathtt{x} := 5;\\ *&\mathtt{y} := *\mathtt{x} \end{aligned}$$

After the first statement "$\mathtt{x} := \mathtt{malloc}$*", the memory is* $\{\mathtt{x} \mapsto a\}$ *where the a is the address of a fresh memory location. After the second statement "*$\mathtt{y} := \mathtt{\&x}$*", the memory becomes* $\{\mathtt{x} \mapsto a, \mathtt{y} \mapsto \mathtt{x}\}$*. Then the assignment "*$*\mathtt{x} := 5$*" assigns integer 5 to the address stored in* x*, hence the memory becomes* $\{\mathtt{x} \mapsto a, \mathtt{y} \mapsto \mathtt{x}, a \mapsto 5\}$*. Then the last assignment "*$*\mathtt{y} := *\mathtt{x}$*" dereferences the address (a) stored in* x*, gets 5, and stores it to the address (*x*) stored in* y*. Hence, the memory in the end is*

$$\{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto \mathtt{x}, a \mapsto 5\}.$$

This concrete semantics is formalized in the followings.

Concrete transitional semantics. Given a program, its initial states I, and the set $\mathbb{L}$ of the labels for its statements, the transitional-style semantics (Chapter 4) is the least fixpoint of the following semantic function F:

$$\begin{aligned} &\mathbb{S} = \mathbb{L} \times \mathbb{M}\\ &F : \wp(\mathbb{S}) \to \wp(\mathbb{S})\\ &F(X) = I \cup \mathit{Step}(X) \end{aligned}$$

where

$$\mathit{Step} = \breve{\wp}(\hookrightarrow).$$

Before we define the transition relation $\hookrightarrow \subseteq \mathbb{S} \times \mathbb{S}$ between concrete states, we have to define the semantic domains. The memories are as before, maps from locations to values, yet the location (address) domain $\mathbb{A}$ is extended to include dynamically allocated memory addresses as well as program variables. Since such locations can also be values of expressions, the value domain $\mathbb{V}$ contains such locations as well as integers (for integer expressions) and labels (for goto-target expressions):

$\mathbb{M}$	$=$	$\mathbb{A} \to \mathbb{V}$	memories
$\mathbb{A}$	$=$	$\mathbb{X} \cup \mathbb{H}$	addresses (locations)
$\mathbb{V}$	$=$	$\mathbb{Z} \cup \mathbb{L} \cup \mathbb{A}$	values
$\mathbb{X}$			set of variables
$\mathbb{H}$			set of allocated heap addresses
$\mathbb{L}$			set of statement labels

The $\mathbb{H}$ set denotes the locations generated by the **malloc** expressions. Given a program, every **malloc** expression is assumed to have a unique number μ, writing $\mathtt{malloc}_\mu$. Let

the set of these malloc-site numbers be $\mathbb{N}_{site}$. Then we can model the domain of memory addresses as

$$\mathbb{H} = \mathbb{N}_{site} \times \mathbb{N}.$$

Thus, the addresses of fresh locations from a $\texttt{malloc}_\mu$ are $(\mu,0),(\mu,1),\cdots$.

Now we define the transition relation $\hookrightarrow$ for the indirect assignment, which is the only new statement added to the imperative language of Section 4.4 (Figure 4.6, page 113). If label l is an indirect assignment statement, then its transition is:

$$*E_1 := E_2 \quad : \quad (l,m) \hookrightarrow (\texttt{next}(l), update(m, eval_{E_1}(m), eval_{E_2}(m)))$$

where the $update(m,v_1,v_2)$ returns the same memory as m except that location v_1 has value v_2. For example, for assignment statement "$*\texttt{x} := 3$" the evaluation of x will return a location, a stored value in x, into which integer 3 is stored.

The $eval_E(m)$ computes the value of expression E given memory m. For expressions, there are new constructs that generate or dereference locations. The evaluation function *eval* for these three pointer expressions, together with the reminder of the variable expression case, are:

$eval_{\texttt{x}}(m)$	$=$	$fetch(m,\texttt{x})$	stored value in a variable x
$eval_{\texttt{\&x}}(m)$	$=$	x	variable as a location &x
$eval_{\texttt{malloc}_\mu}(m)$	$=$	(μ,z)	new number z for malloc site μ, as a fresh location
$eval_{*E}(m)$	$=$	$fetch(m, eval_E(m))$	dereference of a location

where $fetch(m,v)$ returns the value of m at location v.

In summary, the types of the above semantic operations are

$$\begin{aligned} eval_E : &\quad \mathbb{M} \to \mathbb{V} \\ update : &\quad \mathbb{M} \times \mathbb{V} \times \mathbb{V} \to \mathbb{M} \\ fetch : &\quad \mathbb{M} \times \mathbb{V} \to \mathbb{V} \end{aligned}$$

Though the programs in the above language may easily have two expressions point to the same location (*alias*), when we design a a static analysis we do not need a separate concern about how to handle the alias behavior. This is because all behaviors of programs are defined within the semantics, and the alias behavior is one of the phenomenas that appear from the semantics. Thus, in designing a static analysis it is sufficient to focus on the semantics. This is the power and convenience of semantics-based static analysis framework (Chapter 3 and Chapter 4).

Now, designing a sound static analysis boils down to designing a sound abstract semantics.

8.1.2 An Abstract Semantics

Following the framework of Chapter 4, we define an abstract semantics as

$$\begin{array}{rlcl} \text{abstract domain} & \mathbb{S}^\sharp & = & \mathbb{L} \to \mathbb{M}^\sharp \\ \text{abstract semantic function} & F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp & & \\ & F^\sharp(X^\sharp) & = & \alpha(I) \cup^\sharp \mathit{Step}^\sharp(X^\sharp) \\ & \mathit{Step}^\sharp & = & \wp(\mathrm{id}, \sqcup_M) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp) \end{array}$$

where $\hookrightarrow^\sharp$ is the one-step abstract transition relation $\subseteq \mathbb{S}^\sharp \times \mathbb{S}^\sharp$.

At this level, all things remain the same as being prescribed in Chapter 4. The function π partitions a subset of $\mathbb{L} \times \mathbb{M}^\sharp$ by the labels, returning an element in $\mathbb{L} \to \wp(\mathbb{M}^\sharp)$. The abstract state domain $\mathbb{S}^\sharp$ and the abstract transition relation $\hookrightarrow^\sharp$ needs to satisfy the framework's required conditions:

- We design the set $\mathbb{L} \to \mathbb{M}^\sharp$ of abstract states as a CPO that is Galois-connected with $\wp(\mathbb{L} \times \mathbb{M})$:
$$\wp(\mathbb{L} \times \mathbb{M}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{L} \to \mathbb{M}^\sharp.$$
We design the abstract memory $\mathbb{M}^\sharp$ as a CPO that is Galois-connected with $\wp(\mathbb{M})$:
$$\wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp.$$
- We need to check that the abstract transition relation $\hookrightarrow^\sharp$ as a function satisfies the soundness condition: $\breve{\wp}(\hookrightarrow) \circ \gamma \subseteq \breve{\wp}(\gamma) \circ \hookrightarrow^\sharp$.
- We need to check that the abstract union operator $\cup^\sharp$ satisfies the soundness condition: $\cup \circ (\gamma, \gamma) \subseteq \gamma \circ \cup^\sharp$.

Then for any input program, the algorithms of Section 4.3 soundly approximates the concrete semantics of the program.

Abstract domains. The abstract domains $\mathbb{S}^\sharp$, $\mathbb{M}^\sharp$, $\mathbb{V}^\sharp$, and $\mathbb{A}^\sharp$ are all CPOs that are Galois-connected with the corresponding concrete domains.

$$\wp(\mathbb{L} \times \mathbb{M}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{L} \to \mathbb{M}^\sharp \qquad\qquad \wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp$$

where

$$\begin{array}{rcl} \alpha(S) & = & \{l \mapsto \alpha_M(\{m \mid (l,m) \in S\}) \mid l \in \mathbb{L}\} \\ \gamma(s^\sharp) & = & \{(l,m) \mid m \in \gamma_M(s^\sharp(l)), l \in \mathbb{L}\}. \end{array}$$

Now, the abstract domain $\mathbb{M}^\sharp$ is the parameter.

Example 8.2 (A memory abstract domain) *An example of such a domain $\mathbb{M}^\sharp$ is*

$$\mathbb{M}^\sharp = (\mathbb{X} \cup \mathbb{N}_{site}) \to \mathbb{V}^\sharp$$

given $\mathbb{V}^\sharp$ being a Galois-connected CPO. The $\mathbb{X}$ and $\mathbb{N}_{site}$ are finite sets of, respectively, the variables and the allocation sites in the input program.

The abstraction and concretization functions are:

$$\begin{array}{rcll} \alpha_M(M)(\mathtt{x}) & = & \alpha_V(\{m(\mathtt{x}) \mid m \in M\}) & \text{if } \mathtt{x} \in \mathbb{X} \\ \alpha_M(M)(\mu) & = & \alpha_V(\{m(\mu,n) \mid m \in M, n \in \mathbb{N}\}) & \text{if } \mu \in \mathbb{N}_{site} \\ \gamma_M(M^\sharp) & = & \{m \in \mathbb{M} \mid \forall \mathtt{x} \in \mathbb{X} : m(\mathtt{x}) \in \gamma_V(M^\sharp(\mathtt{x})), & \\ & & \quad \forall \mu \in \mathbb{N}_{site}, \forall n \in \mathbb{N} : m(\mu,n) \in \gamma_V(M^\sharp(\mu))\} & \end{array}$$

Example 8.3 (An abstract value domain) *An example abstract value domain $\mathbb{V}^\sharp$, a CPO that is Galois-connected*

$$\wp(\mathbb{V}) \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \mathbb{V}^\sharp,$$

is achieved by abstracting kind-wise a set of values (integers, labels, and/or locations). We abstract its set of integers into an abstract integer, its set of labels into an abstract label, and a set of locations into an abstract location:

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{A}) \quad \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{A}^\sharp$$

where

$$\begin{array}{rcl} \alpha_V(V) & = & (\alpha_Z(V \cap \mathbb{Z}), \alpha_L(V \cap \mathbb{L}), \alpha_A(V \cap \mathbb{A})) \\ \gamma_V(z^\sharp, l^\sharp, a^\sharp) & = & \gamma_Z(z^\sharp) \cup \gamma_L(l^\sharp) \cup \gamma_A(a^\sharp). \end{array}$$

The abstract integers, abstract labels, and abstract locations need to be Galois-connected CPOs:

$$\wp(\mathbb{Z}) \underset{\alpha_Z}{\overset{\gamma_Z}{\leftrightarrows}} \mathbb{Z}^\sharp \qquad\qquad \wp(\mathbb{L}) \underset{\alpha_L}{\overset{\gamma_L}{\leftrightarrows}} \mathbb{L}^\sharp$$

$$\wp(\mathbb{A} = \mathbb{X} \cup \mathbb{N}_{site} \times \mathbb{N}) \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \mathbb{A}^\sharp = \wp(\mathbb{X} \cup \mathbb{N}_{site}).$$

Note that the set of variables $\mathbb{X}$ and `malloc` *sites $\mathbb{N}_{site}$ are finite for a program; thus, the powersets of those sets can be used as finite abstract domains.*

Abstract transition $\hookrightarrow^\sharp$. Given the above abstract domains, we define the abstract transition relation $\hookrightarrow^\sharp$ as follows.

$$*E_1 := E_2 \quad : \quad (l, M^\sharp) \hookrightarrow^\sharp (\mathtt{next}(l), update^\sharp(M^\sharp, eval^\sharp_{E_1}(M^\sharp), eval^\sharp_{E_2}(M^\sharp)))$$

Note that the abstract transition is identical to the concrete one except that they used abstract versions for the semantic operators (e.g., $update^\sharp$ for $update$).

The types of the semantic operators are:

$$\begin{array}{rl} eval^\sharp_E : & \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\ update^\sharp : & \mathbb{M}^\sharp \times \mathbb{V}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp \\ fetch^\sharp : & \mathbb{M}^\sharp \times \mathbb{V}^\sharp \to \mathbb{V}^\sharp. \end{array}$$

The abstract evaluation operator $eval^\sharp$ can be defined as

$$\begin{array}{rcl} eval^\sharp_{\mathtt{x}}(M^\sharp) & = & fetch^\sharp(M^\sharp, \mathtt{x}) \\ eval^\sharp_{\mathtt{\&x}}(M^\sharp) & = & \{\mathtt{x}\} \\ eval^\sharp_{\mathtt{malloc}_\mu}(M^\sharp) & = & \{\mu\} \\ eval^\sharp_{*E}(M^\sharp) & = & fetch^\sharp(M^\sharp, eval^\sharp_E(M^\sharp)). \end{array}$$

Safe memory operations. If we use the abstract domains of Example 8.2 and Example 8.3, sound memory read or write operations are as follows.

- The memory read operation $fetch^\sharp(M^\sharp, v^\sharp)$ looks up the abstract memory entry at the abstract location $l^\sharp \in \wp(\mathbb{X} \cup \mathbb{N}_{site})$ of $v^\sharp$. Since the abstract location is a set of variables and `malloc` sites, the result is the join of all the entries:

$$\bigsqcup_{a \in l^\sharp} M^\sharp(a).$$

- The memory write operation $update^\sharp(M^\sharp, v_1^\sharp, v_2^\sharp)$ overwrites the memory entry (called *strong update*) when the abstract target location $l^\sharp$ of $v_1^\sharp$ means a single concrete location. Otherwise, the update cannot overwrite the memory. Every entry in the abstract memory that constitutes the target abstract location $l^\sharp \in \wp(\mathbb{X} \cup \mathbb{N}_{site})$ must be joined with the value $v_2^\sharp$ to store (called *weak update*):

$$\bigsqcup_{a \in l^\sharp} M^\sharp[a \mapsto M^\sharp(a) \sqcup v_2^\sharp].$$

Example 8.4 *Consider the program in Figure 8.2 that repeatedly allocates a new memory and overwrite integers to it. The columns named "early", "intermittent", and "stable" show the snapshots of some entries of the abstract memory at each program point during the analysis. The "stable" column show the final result of the analysis.*

We assume that the integer values are abstracted into the integer-interval domain.

First, see the "early" column that captures the abstract memory entries right after the first iteration. The `x` *variable has the abstract address* μ *for all the fresh memories allocated at the* `malloc` *expression. Note the abstract value stored at the* μ *address. The assignment to the abstract address* μ *must be the weak update because* μ *denotes multiple locations. At line 6,* μ *contains* $[0,2]$ *not* $[2,2]$ *as the assigning* $[2,2]$ *to* μ *must be the join with the old value* $[0,0]$.

In the "intermittent" column, see the abstract values at μ *at lines 4 and 6. The assignments does not overwrite but join the new value with the old one. The abstract values stored at* μ *keeps expanding.*

The "stable" column shows the abstract memory entries in the end. The expanding upper bounds of the values at `i` *and* μ *are widened to* $+\infty$ *and become stable there.*

Theorem 8.1 (Safety of $\hookrightarrow^\sharp$ for the pointer language) *Consider the concrete one-step transition of Section 8.1.1 and the abstract transition relation of Section 8.1.2. If the semantic operators satisfy*

		early	intermittent	stable (= early$\bigtriangledown$intermittent)
	`i := 0;`			
1 :		$\mathtt{i} \mapsto [0,0]$	$\mathtt{i} \mapsto [0,1]$	$\mathtt{i} \mapsto [0,+\infty]$
	`while(true){`			
2 :		$\mathtt{i} \mapsto [0,0]$	$\mathtt{i} \mapsto [0,1]$	$\mathtt{i} \mapsto [0,+\infty]$
	`x := malloc`$_\mu$;			
3 :		$\mathtt{x} \mapsto \mu$	$\mathtt{x} \mapsto \mu$	$\mathtt{x} \mapsto \mu$
	`*x := i;`			
4 :		$\mu \mapsto [0,0]$	$\mu \mapsto [0,2]$	$\mu \mapsto [0,+\infty]$
	`i := i + 1;`			
5 :		$\mathtt{i} \mapsto [1,1]$	$\mathtt{i} \mapsto [1,2]$	$\mathtt{i} \mapsto [1,+\infty]$
	`*x := i + 1`			
6 :	}	$\mu \mapsto [0,2]$	$\mu \mapsto [0,3]$	$\mu \mapsto [0,+\infty]$

Figure 8.2
Analysis snapshots of a pointer program

the following soundness properties:

$$\begin{array}{rcl}
\wp(eval_E) \circ \gamma_M & \subseteq & \gamma_V \circ eval_E^\sharp \\
\wp(update) \circ \times \circ (\gamma_M, \gamma_V, \gamma_V) & \subseteq & \gamma_M \circ update^\sharp \\
\wp(fetch) \circ \times \circ (\gamma_M, \gamma_V) & \subseteq & \gamma_V \circ fetch^\sharp \\
\wp(filter_B) \circ \gamma_M & \subseteq & \gamma_M \circ filter_B^\sharp \\
\wp(filter_{\neg B}) \circ \gamma_M & \subseteq & \gamma_M \circ filter_{\neg B}^\sharp
\end{array}$$

then $\breve{\wp}(\hookrightarrow) \circ \gamma \sqsubseteq \breve{\wp}(\gamma) \circ \hookrightarrow^\sharp$. *(The* $\times$ *is the Cartesian product operator of multiple sets.)*

The proof is similarly done as the proof (Appendix B.3) of Theorem 4.4. Note that the definition of $\hookrightarrow^\sharp$ is homomorphic to that of $\hookrightarrow$, that is, the structures of the two definitions are the same except that $\hookrightarrow^\sharp$ uses the abstract versions for the semantic operators. Thus, the soundness of $\hookrightarrow^\sharp$ follows rather straightforwardly from the soundness properties of the abstract semantic operators.

8.2 For a Language with Functions and Recursive Calls

8.2.1 Language and Concrete Semantics

Let us consider an imperative language that allows functions. We extend the imperative language in Chapter 4 (Figure 4.6) with function definitions and recursive calls. See Figure 8.3.

E	::=	$\cdots$	expression, as before (Figure 4.6, page 113)
	|	$\mathtt{f}$	function name
C	::=	$\cdots$	statement, as before (Figure 4.6, page 113)
	|	$E(E)$	function call
	|	**return**	return from call
F	::=	$\mathtt{f}(\mathtt{x}) = C$	function definition
P	::=	$F^+ C$	program, list of function defs followed by a statement

Figure 8.3
Syntax of an example language with functions

The language has only flat function definitions (no nested function definitions) as in the C language. Hence, variables other than function parameters are all global variables. Returning a value from a function is to be simulated by an assignment to a global variable. The function to call is determined at run-time. The function names are first-class values: programmers can store function names in program variables or pass them as parameters to other functions. Hence, like function pointers in C, functions are not necessarily called by their defined names. The function part of the call statement can thus be any expression that evaluates to a function name. We assume the names in a program for the functions and their parameters are all distinct and that every function has one parameter.

Example 8.5 *Consider the following program.*

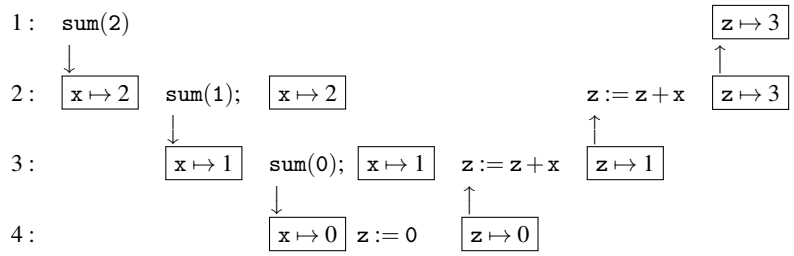
$$\mathtt{sum}(\mathtt{x}) \;=\; \mathbf{if}(\mathtt{x} = 0)\ \{\mathtt{z} := 0;\ \mathbf{return}\}\ \mathbf{else}\ \{\mathtt{sum}(\mathtt{x} - 1);\ \mathtt{z} := \mathtt{z} + \mathtt{x};\ \mathbf{return}\}$$
$$\mathtt{sum}(2)$$

The sum *function is recursive whose result is stored in the* z *variable. After two recursive calls* $\mathtt{sum}(1)$ *and* $\mathtt{sum}(0)$*), the result 3 of* $\mathtt{sum}(2)$ *is stored in* z*.*

The following diagram shows the memory snapshots during the execution. The down-arrows are for calls and up-arrows for returns. Each horizontal line shows a part of the program to evaluate by each call to sum*. Line 2 shows the function body after the initial call* $\mathtt{sum}(2)$*, line 3 the function body after the first recursive call* $\mathtt{sum}(1)$*, and so on.*

The memory entries for the x *and* z *variables are shown inside boxes at the program points. Note that the parameter* x *needs to keep maximum three instances alive simultaneously in the memory (to*

store 2, 1, and 0) during the recursive calls:



Semantic domains. Because of recursive calls, the parameter of a recursive function may have multiple instances alive in the memory at any time during execution. At each recursive call, a new instance of the parameter of the function is allocated in the memory. The function body then uses this new instance as its parameter. The previous instance must be alive and be back to use after the return of the recursive call.

Thus, when defining the semantics of the above language, we need a semantic entity that determines the current instance of function parameters. In semantic jargon, such entity is a table called *environment*. The current environment determines which memory instances of parameters to use.

Also, at function call, the return context must be remembered in order to be recovered on the function return. Each return context consists of the return label (the next label after the function call) and the environment at the function call. Because of recursive calls, we need a stack of remembered contexts, with the most recent context being on the stack top. In semantic jargon, this stack of remembered contexts is called *continuation*.

Thus, we need four components in the state, in addition to the program labels: memory, environment, continuation, and a kind of global counter that we reference to generate fresh instances. The definitions of the semantic domains are shown in Figure 8.4.

Given a program, the sets $\mathbb{X}$, $\mathbb{L}$, and $\mathbb{F}$ are respectively finite sets of variables, labels assigned to each statement, and function names.

An instance $\phi \in \mathbb{I}$ is a time stamp, a kind of global counter that is used to differentiate the instances of formal parameters. We use the global counter ϕ as the instance of the formal parameter at each function call. Since a new instance of a formal parameter is always needed at function calls, the global instance counter ticks at each function call.

Control flow is dynamic. Note that the control flow in the presence of function calls is dynamic; the return from a function call is determined at run-time to flow back to its most-recent call site. Furthermore, a function is not always called by its defined name. Function to call is not syntactically explicit in the program text when the name of a function to call is a value that is stored in a variable or passed as a parameter.

$$
\begin{array}{rrcll}
\langle l,m,\sigma,\kappa,\phi\rangle \in & \mathbb{S} & = & \mathbb{L}\times\mathbb{M}\times\mathbb{E}\times\mathbb{K}\times\mathbb{I} & \\
m \in & \mathbb{M} & = & \mathbb{A}\to\mathbb{V} & \text{memories} \\
\sigma \in & \mathbb{E} & = & \mathbb{X}\to\mathbb{I} & \text{environments} \\
\kappa \in & \mathbb{K} & = & (\mathbb{L}\times\mathbb{E})^* & \text{continuations (stacks of return contexts)} \\
\phi \in & \mathbb{I} & & & \text{instances} \\
a \in & \mathbb{A} & = & \mathbb{X}\times\mathbb{I} & \text{addresses} \\
v \in & \mathbb{V} & = & \mathbb{Z}\cup\mathbb{L}\cup\mathbb{F} & \text{values} \\
 & \mathbb{X} & & & \text{set of variables and parameters} \\
 & \mathbb{L} & & & \text{set of statement labels} \\
 & \mathbb{F} & & & \text{set of function names}
\end{array}
$$

Figure 8.4
Semantic domains for concrete semantics of

Other than the dynamically determined control flow, we let the portion of the control flow that is known beforehand from the program syntax be prepared in $\texttt{next}(l)$, $\texttt{nextTrue}(l)$, and $\texttt{nextFalse}(l)$. They determine the next statement to execute upon completing the execution of each l-labeled statement. Every statement of a given program is uniquely labeled.

Concrete transitional semantics. Concrete semantics of a program for a set I of input states is the least fixpoint

$$\mathbf{lfp}F$$

of monotonic function

$$
\begin{aligned}
&F : \wp(\mathbb{S}) \to \wp(\mathbb{S}) \\
&F(X) = I \cup \wp(\hookrightarrow)(X).
\end{aligned}
$$

The state transition relation $\langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle l',m',\sigma',\kappa',\phi'\rangle$ is defined in Figure 8.5. The transitions from an l-labeled statement are defined by case analysis over the corresponding statement. For each function $\texttt{f}$, let $\mathit{body}(\texttt{f})$ be the label of its body statement and $\mathit{param}(\texttt{f})$ be its formal parameter name.

The expression evaluation function eval_E, the memory update function $\mathit{update}_{\texttt{x}}$, and the filter functions filter_B and $\mathit{filter}_{\neg B}$ are the same as before except that the location of a variable to read or write is the instance of the variable determined by the current environment, hence they need the current environment σ:

$$
\begin{aligned}
\mathit{eval}_E : &\quad \mathbb{M}\times\mathbb{E}\to\mathbb{V} \\
\mathit{update}_{\texttt{x}} : &\quad \mathbb{M}\times\mathbb{V}\times\mathbb{E}\to\mathbb{M} \\
\mathit{filter}_B : &\quad \mathbb{M}\times\mathbb{E}\to\mathbb{M}
\end{aligned}
$$

$$
\begin{array}{rcl}
E_0(E_1) & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle body(\mathtt{f}),\\
& & \quad bind_{\mathtt{x}}(m,\phi',v), \quad \text{parameter binding}\\
& & \quad new\text{-}env_{\mathtt{x}}(\sigma,\phi'), \quad \text{new environment}\\
& & \quad push\text{-}context(\kappa,\mathtt{next}(l),\sigma), \text{ new continuation}\\
& & \quad \phi'\rangle\\
& & \quad \text{where } \mathtt{f} = eval_{E_0}(m,\sigma)\\
& & \qquad \mathtt{x} = param(\mathtt{f})\\
& & \qquad v = eval_{E_1}(m,\sigma)\\
& & \qquad \phi' = tick(\phi)\\
\mathbf{return} & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle l',m,\sigma',\kappa',\phi\rangle\\
& & \quad \text{where } \langle l',\sigma',\kappa'\rangle = pop\text{-}context(\kappa)\\
\mathbf{skip} & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{next}(l),m,\sigma,\kappa,\phi\rangle\\
\mathbf{input}(\mathtt{x}) & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{next}(l),update_{\mathtt{x}}(m,z,\sigma),\sigma,\kappa,\phi\rangle \quad \text{for an input int } z\\
\mathtt{x} := E & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{next}(l),update_{\mathtt{x}}(m,eval_E(m,\sigma),\sigma),\sigma,\kappa,\phi\rangle\\
C_1;C_2 & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{next}(l),m,\sigma,\kappa,\phi\rangle\\
\mathbf{if}(B)\,\{C_1\}\,\mathbf{else}\,\{C_2\} & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{nextTrue}(l),filter_B(m,\sigma),\sigma,\kappa,\phi\rangle\\
& : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{nextFalse}(l),filter_{\neg B}(m,\sigma),\sigma,\kappa,\phi\rangle\\
\mathbf{while}(B)\,\{C\} & : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{nextTrue}(l),filter_B(m,\sigma),\sigma,\kappa,\phi\rangle\\
& : & \langle l,m,\sigma,\kappa,\phi\rangle \hookrightarrow \langle \mathtt{nextFalse}(l),filter_{\neg B}(m,\sigma),\sigma,\kappa,\phi\rangle
\end{array}
$$

Figure 8.5
Concrete transition relation $\hookrightarrow$

Note that a function name `f` is a constant expression. The evaluation $eval_{\mathtt{f}}(m,\sigma)$ is the function name `f` itself.

The other semantic operations are defined as usual. See Figure 8.6.

8.2.2 An Abstract Semantics

Following the framework of Chapter 4, we define an abstract semantics as

$$
\begin{array}{rrcl}
\text{abstract domain} & \mathbb{S}^\sharp & = & \mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp\\
\text{abstract semantic function} & F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp & & \\
& F^\sharp(X^\sharp) & = & I^\sharp \cup^\sharp Step^\sharp(X^\sharp)\\
& Step^\sharp & = & \wp(\mathrm{id},\sqcup_R) \circ \pi \circ \breve{\wp}(\hookrightarrow^\sharp)
\end{array}
$$

where $\hookrightarrow^\sharp$ is the one-step abstract transition relation $\subseteq \mathbb{S}^\sharp \times \mathbb{S}^\sharp$. The function π partitions a set $\subseteq \mathbb{L} \times \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$ by the labels, returning a set $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp)$.

$$\begin{aligned}
\mathit{body}: &\quad \mathbb{F} \to \mathbb{L} \\
\mathit{bind}_{\mathrm{x}}: &\quad \mathbb{M} \times \mathbb{I} \times \mathbb{V} \to \mathbb{M} \\
\mathit{new\text{-}env}_{\mathrm{x}}: &\quad \mathbb{E} \times \mathbb{I} \to \mathbb{E} \\
\mathit{push\text{-}context}: &\quad \mathbb{K} \times \mathbb{L} \times \mathbb{E} \to \mathbb{K} \\
\mathit{pop\text{-}context}: &\quad \mathbb{K} \to \mathbb{L} \times \mathbb{E} \times \mathbb{K} \\
\mathit{tick}: &\quad \mathbb{I} \to \mathbb{I}
\end{aligned}$$

where

$$\begin{aligned}
\mathit{bind}_{\mathrm{x}}(m,\phi,v) &= m[\langle \mathrm{x},\phi\rangle \mapsto v] \\
\mathit{new\text{-}env}_{\mathrm{x}}(\sigma,\phi) &= \sigma[\mathrm{x} \mapsto \phi] \\
\mathit{push\text{-}context}(\kappa,l,\sigma) &= \langle l,\sigma\rangle.\kappa \quad \text{(stack top } \langle l,\sigma\rangle \text{ and the rest } \kappa\text{)} \\
\mathit{pop\text{-}context}(\langle l,\sigma\rangle.\kappa) &= \langle l,\sigma,\kappa\rangle \\
\mathit{tick}(\phi) &= \phi' \qquad \text{(new } \phi'\text{)}
\end{aligned}$$

Figure 8.6
Other semantic operators

The $\sqcup_R$ is the least upper bound operator of $\mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp$. The state abstract domain $\mathbb{S}^\sharp$ and the abstract transition relation $\hookrightarrow^\sharp$ is required to satisfy the framework conditions:

- The abstract state $\mathbb{S}^\sharp$ is to be a CPO, Galois-connected with $\wp(\mathbb{S})$

$$\begin{array}{l}\wp(\mathbb{L} \times \mathbb{M} \times \mathbb{E} \times \mathbb{K} \times \mathbb{I}) \\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{L} \to \mathbb{M}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \times \mathbb{I}^\sharp.\end{array}$$

  where the abstract component domains are also Galois-connected CPOs:

$$\wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp \qquad \wp(\mathbb{E}) \underset{\alpha_E}{\overset{\gamma_E}{\leftrightarrows}} \mathbb{E}^\sharp$$

$$\wp(\mathbb{K}) \underset{\alpha_K}{\overset{\gamma_K}{\leftrightarrows}} \mathbb{K}^\sharp \qquad \wp(\mathbb{I}) \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \mathbb{I}^\sharp$$

- The abstract one-step transition relation $\hookrightarrow^\sharp$ as a function is required to satisfy:

$$\check{\wp}(\hookrightarrow) \circ \gamma \sqsubseteq \check{\wp}(\gamma) \circ \hookrightarrow^\sharp.$$

- The abstract $\cup^\sharp$ in $\mathbb{S}^\sharp$ is required to satisfy: $\cup \circ (\gamma,\gamma) \subseteq \gamma \circ \cup^\sharp$.

Then for any input program, the algorithms (Section 4.3) based on such $F^\sharp$ compute a sound approximation of the concrete semantics of the program.

Memory and environment abstract domains. An example of a memory abstract domain $\mathbb{M}^\sharp$ is a Galois-connected CPO

$$\wp(\mathbb{M}) \underset{\alpha_M}{\overset{\gamma_M}{\leftrightarrows}} \mathbb{M}^\sharp.$$

Example 8.6 (A memory abstract domain) *An example of such a $\mathbb{M}^\sharp$ domain is*

$$\mathbb{M}^\sharp = (\mathbb{X} \times \mathbb{I}^\sharp) \to \mathbb{V}^\sharp$$

given $\mathbb{V}^\sharp$ being a Galois-connected CPO. The set $\mathbb{X}$ is the finite set of variables and parameters in the input program to analyze.

An example of environment abstract domain $\mathbb{E}^\sharp$ is a Galois-connected CPO

$$\wp(\mathbb{E}) \underset{\alpha_E}{\overset{\gamma_E}{\leftrightarrows}} \mathbb{E}^\sharp.$$

Example 8.7 (An abstract environment domain) *An example of such $\mathbb{E}^\sharp$ domain is*

$$\mathbb{E}^\sharp = \mathbb{X} \to \mathbb{I}^\sharp.$$

An example value abstract domain $\mathbb{V}^\sharp$ is a Galois-connected CPO:

$$\wp(\mathbb{V}) \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \mathbb{V}^\sharp$$

Example 8.8 (An abstract value domain) *An example $\mathbb{V}^\sharp$ domain is achieved by abstracting kind-wise a set of values (integers, labels, and/or function names). We abstract its set of integers into an abstract integer, its set of labels into an abstract label, and a set of function names into an abstract function name:*

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{F}) \quad \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{F}^\sharp$$

where the abstract integers, labels, and locations are Galois-connected CPOs:

$$\wp(\mathbb{Z}) \quad \underset{\alpha_Z}{\overset{\gamma_Z}{\leftrightarrows}} \quad \mathbb{Z}^\sharp \qquad \wp(\mathbb{L}) \quad \underset{\alpha_T}{\overset{\gamma_T}{\leftrightarrows}} \quad \mathbb{L}^\sharp \qquad \wp(\mathbb{F}) \quad \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \quad \mathbb{F}$$

Note that the variables $\mathbb{X}$ and the function names $\mathbb{F}$ are finite sets for a program; thus, the powersets of those sets are already finite, eligible for being used as abstract domains in static analysis.

Abstract Transition $\hookrightarrow^\sharp$. The abstract state transition relation for the function call and return statements are in Figure 8.7. Basically, the definition of $\hookrightarrow^\sharp$ is the same as that of $\hookrightarrow$ except that $\hookrightarrow^\sharp$ uses abstract semantic operators for their concrete correspondents.

$$
\begin{array}{lcll}
E_0(E_1) & : & \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp\rangle & \hookrightarrow^\sharp \quad \langle \mathit{body}(\mathtt{f}), \\
 & & & \quad \mathit{bind}^\sharp_{\mathtt{x}}(M^\sharp, \phi'^\sharp, v^\sharp), \quad \text{parameter binding} \\
 & & & \quad \mathit{new\text{-}env}^\sharp_{\mathtt{x}}(\sigma^\sharp, \phi'^\sharp), \quad \text{new environment} \\
 & & & \quad \mathit{push\text{-}context}^\sharp(\kappa^\sharp, \mathtt{next}(l), \sigma^\sharp), \text{ new continuation} \\
 & & & \quad \phi'^\sharp\rangle \\
 & & & \text{where } \mathtt{f} \in \mathit{eval}^\sharp_{E_0}(M^\sharp, \sigma^\sharp) \\
 & & & \qquad \mathtt{x} = \mathit{param}(\mathtt{f}) \\
 & & & \qquad v^\sharp = \mathit{eval}^\sharp_{E_1}(M^\sharp, \sigma^\sharp) \\
 & & & \qquad \phi'^\sharp = \mathit{tick}^\sharp(\phi^\sharp) \\
\mathtt{return} & : & \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp\rangle & \hookrightarrow^\sharp \quad \langle l', M^\sharp, \sigma'^\sharp, \kappa'^\sharp, \phi^\sharp\rangle \\
 & & & \text{where } \langle l'^\sharp, \sigma'^\sharp, \kappa'^\sharp\rangle = \mathit{pop\text{-}context}^\sharp(\kappa^\sharp) \text{ and } l' \in l'^\sharp
\end{array}
$$

Figure 8.7
Abstract transition relation $\hookrightarrow^\sharp$ for function call and return

The abstract semantic operators are homomorphic to their concrete correspondent except that they are defined over abstract domains:

$$
\begin{array}{rl}
\mathit{bind}^\sharp_{\mathtt{x}} : & \mathbb{M}^\sharp \times \mathbb{I}^\sharp \times \mathbb{V}^\sharp \to \mathbb{M}^\sharp \\
\mathit{new\text{-}env}^\sharp_{\mathtt{x}} : & \mathbb{E}^\sharp \times \mathbb{I}^\sharp \to \mathbb{E}^\sharp \\
\mathit{push\text{-}context}^\sharp : & \mathbb{K}^\sharp \times \mathbb{L}^\sharp \times \mathbb{E}^\sharp \to \mathbb{K}^\sharp \\
\mathit{pop\text{-}context}^\sharp : & \mathbb{K}^\sharp \to \mathbb{L}^\sharp \times \mathbb{E}^\sharp \times \mathbb{K}^\sharp \\
\mathit{tick}^\sharp : & \mathbb{I}^\sharp \to \mathbb{I}^\sharp \\
\mathit{eval}^\sharp_E : & \mathbb{M}^\sharp \times \mathbb{E}^\sharp \to \mathbb{V}^\sharp \\
\mathit{update}_{\mathtt{x}}{}^\sharp : & \mathbb{M}^\sharp \times \mathbb{V}^\sharp \times \mathbb{E}^\sharp \to \mathbb{M}^\sharp \\
\mathit{filter}^\sharp_B : & \mathbb{M}^\sharp \times \mathbb{E}^\sharp \to \mathbb{M}^\sharp
\end{array}
$$

Theorem 8.2 (Safety of $\hookrightarrow^\sharp$ for the language with functions) *Consider the concrete one-step transition of Section 8.2.1 and the abstract transition of Section 8.2.2. If the semantic operators*

satisfy the following soundness properties:

$$\begin{array}{rcl}
\wp(\mathit{bind}_{\mathtt{x}}) \circ \times \circ (\gamma_M, \gamma_I, \gamma_V) & \subseteq & \gamma_M \circ \mathit{bind}^\sharp_{\mathtt{x}} \\
\wp(\mathit{new\text{-}env}_{\mathtt{x}}) \circ \times \circ (\gamma_E, \gamma_I) & \subseteq & \gamma_E \circ \mathit{new\text{-}env}^\sharp_{\mathtt{x}} \\
\wp(\mathit{push\text{-}context}) \circ \times \circ (\gamma_K, \gamma_L, \gamma_E) & \subseteq & \gamma_K \circ \mathit{push\text{-}context}^\sharp \\
\wp(\mathit{pop\text{-}context}) \circ \gamma_K & \subseteq & \times \circ (\gamma_L, \gamma_E, \gamma_K) \circ \mathit{pop\text{-}context}^\sharp \\
\wp(\mathit{tick}) \circ \gamma_I & \subseteq & \gamma_I \circ \mathit{tick}^\sharp \\
\wp(\mathit{eval}_E) \circ \times \circ (\gamma_M, \gamma_E) & \subseteq & \gamma_V \circ \mathit{eval}^\sharp_E \\
\wp(\mathit{update}_{\mathtt{x}}) \circ \times \circ (\gamma_M, \gamma_V, \gamma_E) & \subseteq & \gamma_M \circ \mathit{update}^\sharp_{\mathtt{x}} \\
\wp(\mathit{filter}_B) \circ \times \circ (\gamma_M, \gamma_E) & \subseteq & \gamma_M \circ \mathit{filter}^\sharp_B
\end{array}$$

then $\check{\wp}(\hookrightarrow) \circ \gamma \sqsubseteq \check{\wp}(\gamma) \circ \hookrightarrow^\sharp$. *(The* $\times$ *is the Cartesian product operator of multiple sets.)*

The proof is done similarly as the proof (Appendix B.3) of Theorem 4.4. Since the definition of $\hookrightarrow^\sharp$ has the same structure as that of $\hookrightarrow$ except that it uses abstract versions for concrete semantic operators, the soundness of $\hookrightarrow^\sharp$ naturally follows from the soundness of the abstract semantic operators.

Varying the Context Sensitivity. In a concrete execution, every call to a function may happen in a unique machine state. The global variables may have unique values at each call time, and in case of recursive calls the stacked instances of the formal parameter may be different at each call.

If the static analysis abstracts, for each function, all the machine states at the function's multiple calls into a single abstract state, then the analysis is called *context-insensitive*. If the static analysis abstracts the multiple call contexts into a multiple abstract contexts in order to distinguish some differences of the call contexts, then the analysis is called *context-sensitive*.

The context-sensitivity boils down to how we abstract the instance domain $\mathbb{I}$:

$$\wp(\mathbb{I}) \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \mathbb{I}^\sharp$$

If $\mathbb{I}^\sharp$ is a singleton set (that is, if we do not differentiate the parameter instances in abstract semantics), then follows the context-insensitivity.

On the other hand, by using multiple elements for $\mathbb{I}^\sharp$ some kind of context-sensitivity emerges. For example, suppose $\mathbb{I}^\sharp$ uses the set $\mathbb{C}_{site}$ of every call sites of the input program:

$$\mathbb{I}^\sharp = \wp(\mathbb{C}_{site})$$

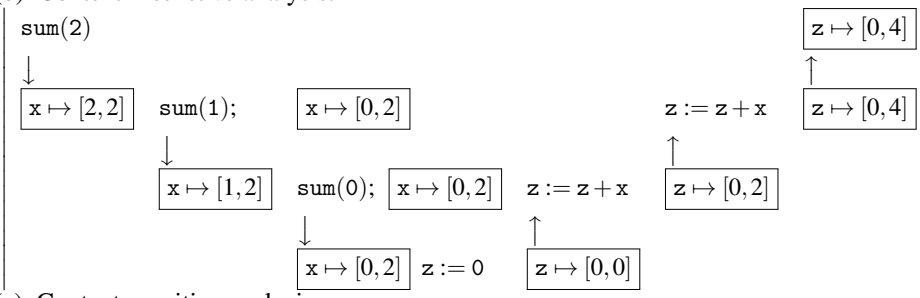
Then multiple abstract instances of each parameter, each of which corresponds to each call site, emerges.

A more elaborate context-sensitivity is possible if we use, say, abstract *call strings*(**?**) for $\mathbb{I}^\sharp$. A call string is an ordered sequence of function names that abstracts the continuation at the moment of a call. The larger the maximum length of the abstract call strings in

(a) Example program:

$$\texttt{sum(x)} \;=\; \textbf{if}(\texttt{x} = 0)\ \{\texttt{z} := 0;\ \textbf{return}\}\ \textbf{else}\ \{\texttt{sum}(\texttt{x}-1);\ \texttt{z} := \texttt{z}+\texttt{x};\ \textbf{return}\}$$
$$\texttt{sum}(2)$$

(b) Context-insenstive analysis:



(c) Context-sensitive analysis:

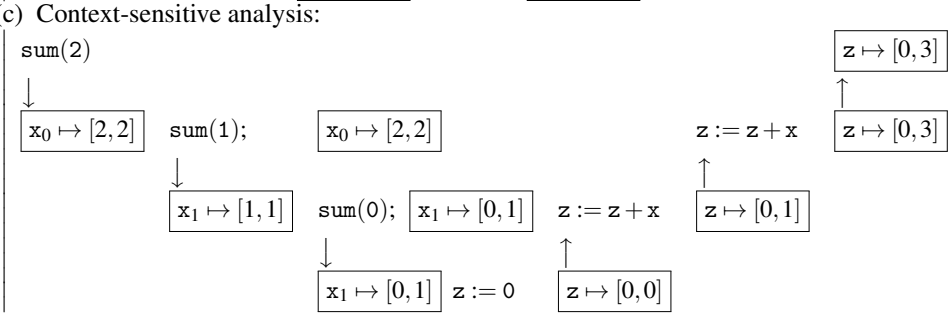


Figure 8.8
Context-insensitive & -senstive analyses of an example program

$\mathbb{I}^\sharp$, the more varied abstract instances are used for a single parameter, hence follows a more elaborate context-sensitive analysis. The precision improvement from the context sensitivity comes with a cost. We can selectively apply the context-sensitivity only to those calls that eventually contribute to the improvement of the analysis precision(**?**).

Example 8.9 *See Figure 8.8. The example program is the same as in Example 8.5 (page 203). Suppose we use the integer-intervals for abstract integers.*

Figure 8.8 (b) and (c) show the abstract memory entries at each program point during analysis. The down-arrows are for calls and up-arrows for returns. Each horizontal line shows a part of the program to analyze by each call to sum.

- *Figure 8.8 (b) shows the abstract memory entries at each program point during the context-insensitive analysis.*

  Recall that the context-insensitivity means that we use a single abstract location for all the instances of each function's parameter. That is, the abstract instance domain $\mathbb{I}^\sharp$ is a singleton set.

In this abstraction, the abstract value $[0,2]$ *of the abstract location* `x` *at the beginning of the function body subsumes all the values passed to* `x`*. The* `z` *variable at the end of the function body also has* $[0,4]$ *that covers all the values stored there.*

- *Figure 8.8 (c) shows the abstract memory entries at each program point during a context-sensitive analysis. Recall that the context-sensitivity boils down to using multiple abstract locations for the instances of each parameter. Suppose we abstract the instances by the call sites. Thus, for the example program we have two abstract instances of* `x`*: one* $\mathtt{x}_0$ *for the initial call of* `sum(2)` *and the other* $\mathtt{x}_1$ *for the recursive calls of* `sum(1)` *and* `sum(0)`.

  Note that the first instance $\mathtt{x}_0$ *of* `x` *for the initial call to* `sum` *has* $[2,2]$*. The second instance* $\mathtt{x}_1$ *has* $[0,1]$ *that covers all the values bound to* `x` *by the two recursive calls. These two call-site-sensitive abstract locations for* `x` *results in a precise analysis for the final abstract value of* `z` *in the end. At the finish of the two recursive calls,* `z` *contains* $[0,1]$*, not* $[0,2]$ *as in the context-insensitive case, because it involves only* $\mathtt{x}_1$*. At the finish of the initial call,* `z` *contains* $[0,3]$ *having the (weak-update) effect of the assignment of the sum of* $[0,1]$ *(*`z`*) and* $[2,2]$ *(the value of* $\mathtt{x}_0$*).*

8.3 Abstractions for Data-Structures

The previous sections presented a general approach to the design of static analysis for programming languages with pointers and with functions. This section and the next one take a complementary step, and study abstractions that are specific to families of data-structures, and functions. Their purpose is to provide readers with a good intuition of what kind of abstraction should be used in any given practical case. On the other hand, they provide a higher level view of the analysis algorithms.

In this section, we study abstractions for data-structures:

- arrays (Section 8.3.1);
- buffers and strings (Section 8.3.2);
- statically allocated pointer structures (Section 8.3.3), and dynamically allocated pointer structures (Section 8.3.4).

8.3.1 Arrays

Arrays are one of the most common data-structures. An array reserves a contiguous memory region of fixed size, an element of which can be accessed using an integer index. Therefore, vectors and matrices are naturally represented as arrays. Similarly, arrays are very adapted to the storage of tables of data. For instance, we encounter such structures in all sorts of applications ranging from user level data-bases (each entry corresponds to one array cell) to Operating System internal states (each process corresponds to a cell in a process table array). There are even some programming languages designed around the notion of arrays, like spreadsheet environments (Excel and similar) or array programming languages (APL, Fortran and similar).

Extension of the language syntax and semantics. To illustrate the issues inherent in the abstraction of arrays, we consider a minimal extension of the language of Chapter 3 with

10 Specialized Static Analysis Frameworks

Goal of the chapter: We discuss three specialized frameworks that are less general but simpler than abstract interpretation.

Specialized frameworks are analogous to domain-specific programming languages as opposed to general-purpose ones. For a limited set of target languages and properties, these specialized frameworks are simple to use yet powerful enough. They can be practical alternatives to the general abstract interpretation framework when the target languages and properties are good fits for them. The burden of soundness proof can be reduced and the special algorithms can outperform the general worklist-based fixpoint iteration algorithms.

Recommended reading: [S], [D]

This chapter is targeted at students and developers, who would like to explore specialized static analysis techniques that can be simple yet powerful enough for specific cases in hand. This chapter is also a reminder to the readers who are already familiar with the specialized static analysis techniques of, if any, limitations and how these techniques can be seen from the general abstract interpretation point of view. This chapter is intended more as a survey of the specialized frameworks than as their in-depth coverages.

10.1 Static Analysis by Equations

From the equations point of view, static analysis comprises equations set-up and equations resolution. For an input program, a set of equations captures all the executions of the program. A solution of this set of equations is the analysis result.

Capturing the dynamics of a program by a set of equations is indeed implicit in the general frameworks of Chapter 3 and Chapter 4. There, the concrete semantics of a program (and usually, its finitely computable abstract version) is defined as a fixpoint of the program's semantic function. Note that a function f's fixpoint, c such that $c = f(c)$, is nothing but a solution of the equation $x = f(x)$.

However, the general frameworks of Chapter 3 and Chapter 4 are sometimes overkills. When the target programming language is simple enough, it is rather straightforward to

set up correct equations for programs without formally defining an abstract semantics that soundly approximates the concrete semantics of the target language (**?**).

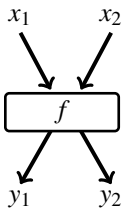
10.1.1 Data-flow Analysis

When is the target programming language simple enough to be suitable for this approach to static analysis? It is when the execution order (control flow) of a program is fixed and explicit from the program text before the execution and when the execution of each construct is simple.

The input programs of such languages are represented by control-flow graphs, and the analysis focuses on the flow of data over the fixed control-flow graphs. By this reason, this approach to static analysis is called *data-flow analysis*.

Program as a graph. Suppose a simple imperative language with assignment, sequence, if-branch, while-loop, and with no pointers and no function calls. The control flows of programs are always fixed and clear from the program source code. The control flow of a program is represented as a directed control-flow graph. The nodes are the atomic statements or conditions of the program. The directed edges determines the execution order (control flow) between the statements. An if-branch statement has two branches in the graph: one with the condition expression node followed by the true branch statements and the other with the node for the negation of the condition expression followed by the false branch statements. Similarly for the while-loop statement.

Equations. The equations are about the states that flow at each edge of the graph. For each node of the form (without loss of generality, we consider two incoming and outgoing edges, respectively):



The edges describe the control flow, and each node f is a state transformation function for the corresponding statement of the program. The function f simulates the statement's semantics: it transforms the incoming machine state (pre-state) into the resulting machine state (post-state). The x_i's are two incoming pre-states and y_i's are the post-states that flow out along the outgoing edges. Then the equations are set up as

$$y_1 = f(x_1 \sqcup x_2)$$
$$y_2 = f(x_1 \sqcup x_2)$$

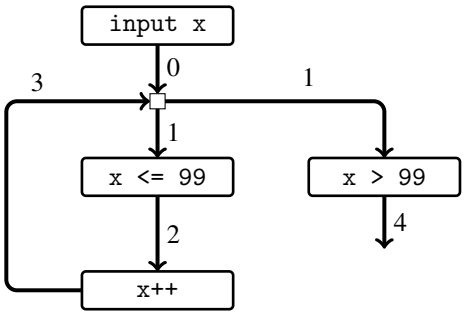


Figure 10.1
An example control flow graph

$$
\begin{aligned}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \sqcap [-\infty, 99] \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty]
\end{aligned}
$$

Figure 10.2
A set of equations for the program in Figure 10.1

The value space that the unknowns (x_i's & y_i's) range over is a lattice whose partial order corresponds to the information subsumption: $a \sqsubseteq b$ means that the set implied by a is also implied by b. An expression of the form $a \sqcup b$ denotes the least upper bound of a and b, the least element that subsumes both a and b. The function f is a monotonic function that describes the operation (semantics) of the node's statement over the lattice of values.

Example 10.1 *Consider the following program.*

```
input (x);
while (x <= 99)
   { x := x+1 }
```

The control flow graph of this program is shown in Figure 10.1. The edges are numbered. The empty node that does nothing corresponds to the identity transformation function.

Suppose we are interested in the value range of variable `x` *on each edge. Suppose we represent value ranges as integer intervals. A set of equations for the values* x_i *of* `x` *at each program label* i *would be, for example, as in Figure 10.2:*

- *The equation for* x_0

$$x_0 = [-\infty, +\infty]$$

describes that `x` *at edge 0 may be any integer because the input is unknown.*

- *The equation for* x_1

$$x_1 = x_0 \sqcup x_3$$

describes the value of `x` *at edge 1 has* `x`*'s value at 0 or 3 (because of the while-loop iterations).*
- *The equation for* x_2

$$x_2 = x_1 \sqcap [-\infty, 99]$$

says the value of `x` *at edge 2 has the value at edge 1 but in case of it less than or equal to 99.*
- *The equation for* x_3

$$x_3 = x_2 \oplus 1$$

says the value of `x` *at edge 3 has the value incremented by 1 to the value at edge 2.*
- *The equation for* x_4

$$x_4 = x_1 \sqcap [100, +\infty]$$

says the value of `x` *at line 4 has the value at edge 1 in case of it greater than 99.*

The definitions of the operators $(\sqcup, \sqcap, \oplus)$ *in the equations are as follows. The soundness of these definitions are easy to see:*

$$\begin{aligned}
[a,b] \sqcup [a',b'] &= [min(a,a'), max(b,b')] \\
[a,b] \sqcap [a',b'] &= \textit{if } max(a,a') > min(b,b') \textit{ then } [\,] \textit{ else } [max(a,a'), min(b,b')] \\
[a,b] \oplus c &= [a+c, b+c]
\end{aligned}$$

It is obvious that the equations in Figure 10.2 with above definitions of the operators capture all the execution cases of the program. Formally proving this correctness is a much ado for the obvious.

Computing a solution of the equations can be done by the general fixpoint iterations such as those in Section 4.3 because all operators are monotonic over the lattice space of the intervals. For finite-height lattice, this naive iterations can reach a fixpoint in finite time.

When the lattice is of infinite-height, as is the interval lattice, the naive fixpoint iterations cannot terminate for some programs. In this case we can use the sound finite computation technique such as the widening operator in Chapter 3 and Chapter 4 to guarantee the termination and the soundness (over-approximation) of the result.

Limitation. Though the above equations are simple so that their correctness is obvious, we cannot rely on the approach for arbitrary languages. First, the dichotomy of control being fixed and data being dynamic does not hold in modern languages. If the target language allows the control-flow as values (for example, jump targets as values, functions as values, or exceptions as values), the control flow graph *a priori* is not possible. The control flow can generally not be represented as a static finite graph. Edges are introduced during program execution. The control flows emerge only during static analysis.

Second, the sound transformation function of each node are not obvious for modern languages. In the above Example 10.1, though the operators "$\sqcup$", "$\sqcap$", and "$\oplus 1$" are straightforward for the example program construct at each node, for complicated languages and value domains the sound definitions can be error-prone. A framework to check the transformation function's soundness (such as Theorem 4.4 of page 117, Theorem 8.1 of page 201, and Theorem 8.2 of page 209) is necessary so that the equation solution guarantees to be sound.

$$
\begin{aligned}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty].
\end{aligned}
$$

Figure 10.3
Another set of sound equations for the program in Figure 10.1

Third issue is about design choices. For a program, the equations are not unique. There are a vast number of ways to set up sound equations that over-approximate all the executions of the program. From this collection of sound equation sets, an analysis chooses one particular way of setting up equations. For example, given the program in the above Example 10.1, different equations sets may be chosen, whose solutions also cover all the possible executions of the program. One such example is in Figure 10.3.

The data-flow analysis lacks a systematic approach to explore various possible choices of sound equations for a given program. For every choice, though simple, we have to argue the soundness of the equations from scratch.

Meanwhile, the semantics-based general frameworks (Chapter 3 and Chapter 4) work for an arbitrary programming language once the concrete semantics of the language is defined in the styles supported by the frameworks. From the point of view of data-flow analysis, the semantic frameworks guide us on what to check (Section 3.4 and Section 4.2.3) to set up sound equations from the programs to analyze. In the semantic frameworks sound and finitely computable static analyses with various accuracies can be systematically defined, only if the required properties of the abstract domains and of the abstract semantic function are ensured.

10.2 Static Analysis by Monotonic Closure

From the monotonic-closure point of view, static analysis comprises setting up initial facts then collecting new facts by a kind of chain reactions. An analysis definition consists of rules for collecting initial facts and rules for inductively generating new facts from existing facts. An analysis in action accumulates facts until no more fact is possible. This collection of facts is an over-approximation of the program behavior.

The initial facts are those that are immediate from the program text. The chain reaction steps generate new facts from existing facts by simulating the program semantics. The analysis terminates when no more facts are generated. For the analysis to terminate, the universe of facts for a given input program must be finite.

For simple target languages and properties, the soundness of the analysis – setting up the initial facts and the chain reactions – are straightforward.

Formally, let R be the set of the chain-reaction rules and X_0 be the initial fact set. Let *Facts* be the set of all possible facts. Notation $X \vdash_R Y$ denotes that new set Y of facts is generated from X by applying all possible rules in R. Then the analysis result is

$$\bigcup_{i \geq 0} Y_i$$

where

$$\begin{aligned} Y_0 &= X_0 \\ Y_{i+1} &= Y \text{ such that } Y_i \vdash_R Y. \end{aligned}$$

Or, equivalently, the analysis result is the least fixpoint

$$\bigcup_{i \geq 0} \phi^i(\emptyset)$$

of monotonic function $\phi : \wp(Facts) \to \wp(Facts)$

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$

10.2.1 Pointer Analysis

Let us consider the following simple C-like imperative language. A program is a collection of assignments.

P	::=	C	program
C	::=		statement
	|	$L := R$	assignment
	|	C ; C	sequence
	|	**while** B C	while-loop
L	::=	$x \mid *x$	target to assign to
R	::=	$n \mid x \mid *x \mid \&x$	value to assign
B			boolean expression

The left- or right-hand-side of an assignment may be a variable(x), the dereference ($*x$) of a variable, or the location ($\&x$) of a variable. Let the semantics of these reference and dereference constructs be the same as in the C language. A variable may store an integer or the location of a variable.

Target property. Suppose we are interested in the set of locations that each pointer variable may store during program execution. Computing such set is reduced to collecting all possible "point-to" facts between two variables: which variable can store the location of which variable. We hence represent each points-to fact by a pair of two variables: `a` $\to$ `b`

denotes that variable a can point to (can have the address of) variable b. The set of such points-to facts is finite for each program because a program has a finite number of variables.

Rules. The analysis globally collects the set of possible points-to facts that can happen during the program execution. We start from an initially empty set. We apply the following rules to add new facts to the global set. This collection (hence the analysis) terminates when no more addition is possible. The rule has the following form

$$\frac{C \quad i_1 \cdots i_k}{j}$$

and it dictates that if the program text has component C and the current solution set has $i_1 \cdots i_k$ then add j to the solution set (**?**). The $i_1 \cdots i_k$ part can be omitted.

The initial facts that are obvious from the program text are collected by this rule:

$$\frac{x := \&y}{x \to y}$$

That is, for each assignment statement of the form $x := \&y$, add the fact that x points to y.

The chain-reaction rules are for other cases of assignments:

$$\frac{x := y \quad y \to z}{x \to z} \qquad \frac{x := *y \quad y \to z \quad z \to w}{x \to w}$$

$$\frac{*x := y \quad x \to w \quad y \to z}{w \to z} \qquad \frac{*x := *y \quad x \to w \quad y \to z \quad z \to v}{w \to v}$$

$$\frac{*x := \&y \quad x \to w}{w \to y}$$

Soundness. It is easy to see that the above rules will collect every possible points-to facts of the input program. The easiness comes from the simple semantics of the target language. The six rules follow, in terms of the points-to information, from the semantics of all six cases of the assignment statements. Consider the last rule, for example. The assignment statement $*x := \&y$ stores the location $\&y$ of y to the location that x points to. Thus, the rule adds $w \to y$ if x points to w. Other rules are similarly straightforward to see their soundness.

Given the global set of facts, applying all the possible rules to the set is a monotonic operation; every rule always adds, if any, new facts to the global set. The analysis result is the least fixpoint of this applying-all-rules function.

The over approximation comes from two sources. First, the rules ignore the conditional execution of the while-loop. Regardless of the while-loop condition the rules are applied to the assignments in the loop body. Second, we collect the points-to facts into a single

global set. This means that a points-to fact from any statement in the input program can trigger a new points-to fact at any statement.

Example 10.2 *Consider the following program.*

```
x := &a ;
y := &x ;
while B
   *y := &b ;
*x := *y
```

Initial facts are from the first two assignments:

$$\texttt{x} \to \texttt{a},\ \texttt{y} \to \texttt{x}.$$

From $\texttt{y} \to \texttt{x}$ *and the assignment body of the while-loop, the analysis can apply the last rule to add*

$$\texttt{x} \to \texttt{b}.$$

For the last assignment and the hitherto collected facts the analysis can apply the second-to-the-last rule and add new facts as follows: from $\texttt{x} \to \texttt{a}$ *and* $\texttt{y} \to \texttt{x}$*, the analysis can add* $\texttt{a} \to \texttt{a}$*; from* $\texttt{x} \to \texttt{b}$ *and* $\texttt{y} \to \texttt{x}$*, the analysis can add* $\texttt{b} \to \texttt{b}$*; from* $\texttt{x} \to \texttt{a}$*,* $\texttt{y} \to \texttt{x}$*, and* $\texttt{x} \to \texttt{b}$*, the analysis can add* $\texttt{a} \to \texttt{b}$*; from* $\texttt{x} \to \texttt{b}$*,* $\texttt{y} \to \texttt{x}$*, and* $\texttt{x} \to \texttt{a}$*, the analysis can add* $\texttt{b} \to \texttt{a}$*.*

Limitation. Note that the above rules do not take the control flow into account. The rules are applied for any assignment statement regardless of where it appears. For example, there is no separate rule for while-loop statement. Regardless of where an assignment appears in the program (e.g., whether an assignment appears within a while-loop body or outside), the analysis blindly collects every possible points-to facts from the collection of the assignment statements in the program.

Similarly, the above analysis is flow-insensitive: the closure rules are oblivious to the statement order in the input program. Regardless of where the assignment statement appears in the program, whenever the premise (numerators) of a rule holds the analysis applies the rule.

For example, for a sequence of statements

```
x := &a ; y := x ; x := &b
```

the analysis concludes $\texttt{y} \to \texttt{a}$ and $\texttt{y} \to \texttt{b}$ because $\texttt{x} \to \texttt{a}$ and $\texttt{x} \to \texttt{b}$ even though y has x before $\texttt{x} \to \texttt{b}$.

This flow-insensitivity can indeed be avoided by pre-processing the input program into a static single assignment (SSA) form where each variable is defined only once. Two writes to a single variable and its reads are transformed into writes to two different variables and their distinctive reads.

10.2.2 Higher-order Control Flow Analysis

As an another example, consider the following higher-order call-by-value functional language. We uniquely label each sub-expression of the program.

P	::=	F	program
F	::=		expression
	|	$\mathtt{x}$	variable
	|	$\lambda \mathtt{x}.E$	a function with argument x and body E
	|	$E\ E$	function application
E	::=	F_l	expression F with label l

A program is defined as an expression that has no free variable. Program executions are defined by the transition steps, each step ($\rightarrow$) of which is by the *beta-reduction* in the call-by-value order. The beta-reduction step is

$$(\lambda \mathtt{x}.e)\ e' \rightarrow \{e'/\mathtt{x}\}e$$

where $\{e'/\mathtt{x}\}e$ denotes the expression obtained by replacing $\mathtt{x}$ by e' in e. We assume that, during execution, every function's argument is uniquely re-named.

For example, the following program

$$(\lambda \mathtt{x}.(\mathtt{x}(\lambda \mathtt{y}.\mathtt{y})))(\lambda \mathtt{z}.\mathtt{z})$$

runs as follows

$$\begin{array}{rc} & (\lambda \mathtt{x}.(\mathtt{x}(\lambda \mathtt{y}.\mathtt{y})))(\lambda \mathtt{z}.\mathtt{z}) \\ \rightarrow & (\lambda \mathtt{z}.\mathtt{z})(\lambda \mathtt{y}.\mathtt{y}) \\ \rightarrow & \lambda \mathtt{y}.\mathtt{y}. \end{array}$$

During the execution, the first step binds $\mathtt{x}$ to $\lambda \mathtt{z}.\mathtt{z}$ and the second step binds $\mathtt{z}$ to $\lambda \mathtt{y}.\mathtt{y}$.

Target property. Suppose we are interested in which functions are called for each application expression. Since functions can be passed to parameters of other functions, such inter-functional control flow is not obvious from the program text. For the target property we need to collect which lambda expression can be bound to which argument during program execution.

Rules. We let an analysis collect facts about which lambda expression "$\lambda \mathtt{x}.e$" a sub-expression may evaluate to. Hence, we represents each fact by a pair $L \ni R$ meaning "L can have value R" where L is either an expression label or a variable, and R is either an

expression label, a variable, or a lambda expression:

$$\begin{array}{lcll} L & ::= & l \mid \mathtt{x} & \text{expression label or variable} \\ R & ::= & l \mid \mathtt{x} \mid v & \\ v & ::= & \lambda \mathtt{x}.E & \end{array}$$

For each program the set of the pairs is finite because a program has a finite number of expressions (l), variables (x), and lambda expressions (v).

From a global set that is initially empty we apply the following rules to add facts of the form $L \ni R$ to the set. The addition (the analysis) terminates when no more addition is possible. Each rule has the following form

$$\frac{C \quad i_1 \cdots i_m}{j_1 \cdots j_n}$$

and it dictates that if the program text has component C and the hitherto collected facts include $i_1 \cdots i_m$ then add $j_1, \cdots, j_n$ to the global set. The $i_1 \cdots i_m$ part may be empty.

The initial fact set-up rules collect facts that are obvious from the program text only:

$$\frac{(\lambda \mathtt{x}.E)_l}{l \ni \lambda \mathtt{x}.E} \qquad \frac{(\mathtt{x})_l}{l \ni \mathtt{x}}$$

A lambda expression is a constant whose value is itself, and a variable expression contains the value of the variable. The value of a variable will be later collected when the variable as the parameter of a function is bound to the actual parameter value at the applications of the function.

The propagation rules that collects new fact by simulating expression evaluations are:

$$\frac{(E_{l_1}\ E_{l_2})_l \quad l_1 \ni \lambda \mathtt{x}.E_{l_3} \quad l_2 \ni v}{l \ni l_3 \quad \mathtt{x} \ni v} \qquad \frac{l_1 \ni l_2 \quad l_2 \ni v}{l_1 \ni v}$$

As in the points-to analysis case, it is easy to see that the above rules will collect every possible values of expressions and variables. For an application expression $(E_{l_1}\ E_{l_2})_l$, if we knew which function can be called $l_1 \ni \lambda \mathtt{x}.E_{l_3}$ with which parameter $l_2 \ni v$, we add two new facts: the first $l \ni l_3$ about the result of the function application and the other $\mathtt{x} \ni v$ about the parameter binding. The second rule propagates the actual value along the chains of collected facts. Formally proving the soundness seems again a much ado for the obvious.

Example 10.3 *Consider the following program. Every lambda expression is identified by its unique parameter name. Each expression of the program is labeled from 0 to 7.*

$$\underbrace{(\lambda \mathtt{x}.\underbrace{(\mathtt{x}_5(\overbrace{\lambda \mathtt{y}.\mathtt{y}_6}^{3}))}_{2})\overbrace{(\lambda \mathtt{z}.\mathtt{z}_7)}^{4}}_{1}{}_{0}$$

During the execution, variable $\mathtt{x}$ *(expression numbered 5) will be bound to* $\lambda \mathtt{z}.\mathtt{z}$ *and variable* $\mathtt{z}$ *(expression numbered 7) to* $\lambda \mathtt{y}.\mathtt{y}$.

Let's see how this information is collected from the above rules. The initial facts are collected from the lambda expressions 1, 3, and 4, and variable expressions 5, 6, and 7:

$$\{1 \ni \lambda \mathtt{x}.(\mathtt{x}(\lambda \mathtt{y}.\mathtt{y})),\quad 3 \ni \lambda \mathtt{y}.\mathtt{y},\quad 4 \ni \lambda \mathtt{z}.\mathtt{z},\quad 5 \ni \mathtt{x},\quad 6 \ni \mathtt{y},\quad 7 \ni \mathtt{z}.\}$$

- *From application expression 0, we add* $\mathtt{x} \ni 4$ *(parameter binding) and* $0 \ni 2$ *(application result) to the above set;*
- *then by the last propagation rule from* $\mathtt{x} \ni 4$ *and* $4 \ni \lambda \mathtt{z}.\mathtt{z}$*, we add* $\mathtt{x} \ni \lambda \mathtt{z}.\mathtt{z}$ *then from* $5 \ni \mathtt{x}$*,* $5 \ni \lambda \mathtt{z}.\mathtt{z}$ *to the above set;*
- *then from application expression 2, we add* $\mathtt{z} \ni 3$ *(parameter binding) and* $2 \ni 7$ *(application result) to the above set;*
- *then by the last propagation rule, we add to the above set* $\mathtt{z} \ni \lambda \mathtt{y}.\mathtt{y}$ *then, from* $7 \ni \mathtt{z}$*, we add* $7 \ni \lambda \mathtt{y}.\mathtt{y}$ *then* $2 \ni \lambda \mathtt{y}.\mathtt{y}$ *then* $0 \ni \lambda \mathtt{y}.\mathtt{y}$.

Limitation. The above analysis uses a crude abstraction for the function values. Note that a function value in the concrete semantics is a pair made of the function code and a table (called *environment*) that determines the values of the function's free variables. The above analysis completely abstracts away the environment part. In the concrete semantics, a function expression $(\lambda x.E)$ in a program may evaluate into distinct values at different contexts (e.g., when the function's free variable is the parameter of a function that is multiply called with different actual parameters). The above analysis collects only the function code part with no distinction for the values of the function's free variables.

In order to employ some degree of context-sensitivity in the abstraction of the function values, the above analysis needs an overhaul, whose design and soundness assurance will be facilitated by semantic frameworks such as Chapter 3 and Chapter 4. See Section 8.2 and Section 8.4.1).

10.3 Static Analysis by Proof Construction

From the proof-construction point of view, static analysis is a proof construction in a finite proof system. A *finite proof system* is a finite set of inference rules for a pre-defined set of judgments. The soundness of the analysis corresponds to the soundness of the proof

system. The soundness property of the proof system expresses that if the input program is provable, then the program satisfies the proven judgment. Thus, for a sound proof system, if a program violates a target judgment then proving the judgment fails for the program.

10.3.1 Type Inference

Let us consider a proof system whose judgment is about the types (**?**) of program expressions. The set of types corresponds to an abstract domain of a static analysis. A type over-approximates the values of an expression. This proof system is called *type system*, and the proof construction called *static type inference* or *static typing*.

Let us consider the following higher-order language with the call-by-value evaluation.

P	::=	E	program
E	::=		expression
	|	n	integer
	|	$\mathtt{x}$	variable
	|	$\lambda \mathtt{x}.E$	function
	|	$E\ E$	function application

In type systems, the judgment that says expression E has type τ is written as

$$\Gamma \vdash E : \tau$$

where Γ is a set of type assumptions for the free variables in E and τ is a type. The judgment means that the evaluation of e with its free variables having values of the types as assumed in Γ is type-safe (runs without a type error) and returns a value of type τ if it terminates.

Simple Type Inference. The above language has two kinds of values, namely integers and functions. Hence, we use following types.

$$\tau ::= \mathit{int} \mid \tau \to \tau$$

Type *int* describes integers, and $\tau_1 \to \tau_2$ the type-safe functions from τ_1 to τ_2 (functions that if given a value of type τ_1 run without a type-error and return a value of type τ_2 if they terminate).

We say that "a program runs without a type error" whenever during the evaluation of the applications expression "$E_1\,E_2$" in the program the value of E_1 is always a function value, not an integer.

The type environment Γ is a finite map from variables to types. Let us write each entry as $x : \tau$. The notation $\Gamma + x : \tau$ denotes the same function as Γ except that its entry for x is τ.

$$\frac{}{\Gamma \vdash n : int} \qquad \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}$$

$$\frac{\Gamma + \mathtt{x} : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda \mathtt{x}.E : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1\ E_2 : \tau_2}$$

Figure 10.4
Proof rules of simple types

The simple type system is shown in Figure 10.4. Each proof rule

$$\frac{J_1 \cdots J_k}{J}$$

is read as: whenever all the premises ($J_1, \cdots J_k$) are provable then the conclusion (J) is provable. A rule with an empty set of premises is an axiom, i.e., its denominator is always provable.

Example 10.4 *For example, program*

$$(\lambda \mathtt{x}.\mathtt{x}\ 1)(\lambda \mathtt{y}.\mathtt{y})$$

is typed int because we can prove

$$\emptyset \vdash (\lambda \mathtt{x}.\mathtt{x}\ 1)(\lambda \mathtt{y}.\mathtt{y}) : int$$

as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{x} : int \to int\ \in \{\mathtt{x} : int \to int\}}{\{\mathtt{x} : int \to int\} \vdash \mathtt{x} : int \to int} \quad \dfrac{}{\{\mathtt{x} : int \to int\} \vdash 1 : int}}{\{\mathtt{x} : int \to int\} \vdash \mathtt{x}\ 1 : int}}{\emptyset \vdash \lambda \mathtt{x}.\mathtt{x}\ 1 : (int \to int) \to int} \quad \dfrac{\dfrac{\mathtt{y} : int\ \in \{\mathtt{y} : int\}}{\{\mathtt{y} : int\} \vdash \mathtt{y} : int}}{\emptyset \vdash \lambda \mathtt{y}.\mathtt{y} : int \to int}}{\emptyset \vdash (\lambda \mathtt{x}.\mathtt{x}\ 1)(\lambda \mathtt{y}.\mathtt{y}) : int}$$

Note that each step of the above proof is an instance of one of the proof rules in Figure 10.4.

The proof rules of Figure 10.4 are sound:

Theorem 10.1 (Soundness of the proof rules) *Let E be a program, an expression without free variables. If $\emptyset \vdash E : \tau$ then the program runs without a type error and returns a value of type τ if it terminates.*

The soundness proof is done by proving two facts, upon defining the program execution in the transitional style as a sequence of steps. The first proof (called *progress lemma*) shows that a typed program, unless it is a value, can always progress one step. The second proof (called *preservation lemma*) shows that typed program after its one step progress still has the same type. By these two facts, a typed program runs without a type error and returns a value of its type if it terminates. The readers can refer to (**?**) for proofs.

Offline Algorithm. The simple type system has a faithful and efficient algorithm for its proof construction. The algorithm is faithful: it succeeds for a judgment if and only if the judgment is provable in the simple type system. The algorithm is efficient: it uses a special operator called *unification* that has no iterative computations (or, just a single iteration) as in the fixpoint algorithms of the general frameworks in Chapter 3 and Chapter 4.

The static typing algorithm can be understood as two steps. First, by scanning the input program we collect equations about the types of every sub-expression of the program. Then we solve the equations by the unification procedure. This offline algorithm can be made online where we solve the type equations while we scan the program.

The following procedure V collects type equations of the form $\tau \doteq \tau$ from the input program. Types in type equations contain type variables α (unknowns) ($\tau ::= \alpha \mid int \mid \tau \to \tau$). Given a program e, an expression without a free variable, the type equations are collected by calling a function $V(\emptyset, E, \alpha)$ with the empty type environment and a fresh type variable α. Procedure $V(\Gamma, e, \tau)$ returns a set of type equations that must hold in order for the e expression to have type τ under assumption Γ:

$$\begin{array}{rcll} V(\Gamma, n, \tau) & = & \{\tau \doteq int\} & \\ V(\Gamma, \mathtt{x}, \tau) & = & \{\tau \doteq \Gamma(\mathtt{x})\} & \\ V(\Gamma, \lambda \mathtt{x}.E, \tau) & = & \{\tau \doteq \alpha_1 \to \alpha_2\} \cup V(\Gamma + \mathtt{x} : \alpha_1, E, \alpha_2) & (\text{new } \alpha_1, \alpha_2) \\ V(\Gamma, E_1\ E_2, \tau) & = & V(\Gamma, E_1, \alpha \to \tau) \cup V(\Gamma, E_2, \alpha) & (\text{new } \alpha) \end{array}$$

Solving the type equations from V is equivalent to proving the given program in the simple type system:

Theorem 10.2 *Let E be a program (an expression without a free variable) and α a fresh type variable. S is a solution for the collection $V(\emptyset, E, \alpha)$ of type equations if and only if judgment $\emptyset \vdash E : S\alpha$ is provable.*

Now, solving the equations is done by the unification procedure (**?**). The unification procedure *unify*, given two types τ_1 and τ_2 of a type equation $\tau_1 \doteq \tau_2$, finds a substitution S (a finite map from type variables to types) whenever possible that make the both sides of the equation literally the same: $S\tau_1 = S\tau_2$. The following *unify* indeed finds the least solution: the most-general unifying substitution. The concatenation S_2S_1 of two substitutions denotes a substitution that applies S_1 then S_2: $(S_2S_1)\tau = S_2(S_1\tau)$.

Procedure $unify(\tau_1, \tau_2)$ is defined as follows, to do the first match from the top:

$$\begin{array}{rcll} unify(\alpha, \tau) & = & \{\alpha \mapsto \tau\} & \text{if } \alpha \notin \tau \\ unify(\tau, \alpha) & = & \{\alpha \mapsto \tau\} & \text{if } \alpha \notin \tau \\ unify(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & let\ S_1 = unify(\tau_1, \tau_1') & \\ & & \quad S_2 = unify(S_1\tau_2, S_1\tau_2') & \\ & & in\ S_2 S_1 & \\ unify(\tau, \tau') & = & failure & \text{other cases} \end{array}$$

Finding a substitution that satisfies all the equations in collection $\{\tau_{1_1} \doteq \tau_{2_1}, \cdots, \tau_{1_k} \doteq \tau_{2_k}\}$ is a simple accumulation of the substitution that unifies each type equation:

$$\begin{array}{rcl} Solve(\{\tau_1 \doteq \tau_2\}) & = & unify(\tau_1, \tau_2) \\ Solve(\{\tau_1 \doteq \tau_2\} \cup rest) & = & let\ S = unify(\tau_1, \tau_2) \\ & & in\ (Solve(S\,rest))S \end{array}$$

where "$S\,rest$" denotes new type equations after applying the S substitution to every type variable in *rest*.

For a program E the simple type inference is thus

$$Solve(V(\emptyset, E, \alpha)).$$

Online Algorithm The above offline algorithm that first collects equations then solves them can be re-phrased as an online algorithm. For a program E the online simple type inference is

$$M(\emptyset, E, \alpha)$$

where

$$\begin{array}{rcll} M(\Gamma, n, \tau) & = & unify(\tau, int) & \\ M(\Gamma, \mathtt{x}, \tau) & = & unify(\tau, \Gamma(\mathtt{x})) & \\ M(\Gamma, \lambda \mathtt{x}.E, \tau) & = & let\ S_1 = unify(\tau, \alpha_1 \to \alpha_2) & (\text{new } \alpha_1, \alpha_2) \\ & & \quad S_2 = M(S_1\Gamma + x : S_1\alpha_1, E, S_1\alpha_2) & \\ & & in\ S_2 S_1 & \\ M(\Gamma, E_1\,E_2, \tau) & = & let\ S_1 = M(\Gamma, E_1, \alpha \to \tau) & (\text{new } \alpha) \\ & & \quad S_2 = M(S_1\Gamma, E_2, S_1\alpha) & \\ & & \text{in } S_2 S_1 & \end{array}$$

The above algorithm is easy to follow since the call to $M(\Gamma, E, \tau)$ computes solutions in order for expression E to have type τ under type assumption Γ for the free variables of E.

Running the M algorithm is equivalent to proving in the type system:

Theorem 10.3 *Let E be a program (an expression without a free variable) and α a fresh type variable. If $M(\emptyset, E, \alpha) = S$ then $\emptyset \vdash E : S\alpha$. Conversely, if $\emptyset \vdash E : \tau$ then $M(\emptyset, E, \alpha) = S_1$ and $\tau = (S_2 S_1)\alpha$ for an additional subsititution S_2.*

Polymorphic Type Inference. Recall that, from the static analysis point of view, the type inferene of the sound simple type system over-approximates the set of values of expressions of the input program. From the soundness point of view, if the analysis succeeds in typing the input program then we can safely conclude that the program will run without a type error. On the other hand, if the analysis fails to type the input program, that means we don't know whether the program will have a type error or not.

The accuracy of the simple type system can be improved by controlling the degree of the over-approximation so that the number of fail cases for type-safe programs should be reduced.

A more accurate system than the simple type system is one called *polymorphic* type system. The polymorphic type system reduces the failure cases for type-safe programs while it is still sound: when the type inference succeeds the input program will run without a type error.

The set of polymorphic types is a finer abstract domain than that of the simple types. The abstract domain is refined because there exist elements (polymorphic types) that represent special sets of values that are not precisely singled out as a separate abstract element in simple types. For example, a polymorphic function type can represent a set of functions that can run regardless of the types of arguments. For example, a polymoprhic type, written as

$$\forall \alpha.\alpha \rightarrow int$$

denotes a set of functions that, whatever the type of the actual argument, run without a type error and return an integer if they terminate.

One polymorphic type inference system is called *let-polymorphic* type system (**?**). This polymorphic type system, which is common in ML-like programming languages, has as for the simple type system sound and complete algorithms (**?**) using the unification procedure. The let-polymorphic type system always computes *principal types* among multiple polymorphic type candidates for each expression. From the static analysis point of view, when principal types exist, it means the existence of best abstractions in their abstract domains.

During typing, a function that can behave independent of its argument types has a generalized, polymorphic types. At different call to the polymorphic function, the polymorphic type of the function is instantiated into an appropriate type. For a function, the polymorphic generalization, if possible, and the distinct instantiations at different calls to the function are analogous to context-sensitive analysis that analyzes functions differently at different call site.

Limitation. For target programming languages that lack a sound static type system we have to invent it. We have to design a finite proof system and prove its soundness. Then we have to find its algorithm and prove its soundness. The burden grows if the algorithm has to solve some constraints that turn out to be unsolvable by the unification procedure.

For languages like ML that already has a sound let-polymorphic static type system, we can instrument its sound type system to carry an extra entity (elements of extra abstract domains) of our interest other than just the conventional type (**?**). However, again, if the system derives some constraints that are unsolvable by the unification procedure, it is our burden to invent a new algorithm and prove its soundness from scratch.