IMP Abstract Semantics

CS 260

1 IMP Abstract Syntax

$$n \in \mathbb{Z} \qquad x \in \mathit{Variable}$$

$$\begin{aligned} p \in \mathit{Program} &::= \textbf{decl}\ \vec{x}\ \textbf{in}\ \overrightarrow{s} \\ s \in \mathit{Stmt} &::= x := e \mid \textbf{if}\ e\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2} \mid \textbf{while}\ e\ \vec{s} \\ e \in \mathit{Exp} &::= \overline{n} \mid x \mid e_1 \oplus e_2 \\ \oplus \in \mathit{BinaryOp} &::= + \mid - \mid \times \mid \div \mid < \mid \leq \mid = \mid \neq \end{aligned}$$

This serves as a reminder—the abstract syntax is exactly the same as it was for the concrete semantics.

2 IMP Abstract Semantics

We describe the abstract semantic domains that constitute a state of the abstract transition system (Section 2.1), abstract state transition rules (Section 2.2), and the helper functions used by the transition rules (Section 2.3).

2.1 Abstract Semantic Domains

$$\begin{aligned} \hat{\varsigma} \in \mathit{State}^\sharp &= \mathit{Stmt}^? \times \mathit{Locals}^\sharp \times \mathit{Kont}^\star \\ \hat{\rho} \in \mathit{Locals}^\sharp &= \mathit{Variable} \to \mathbb{Z}^\sharp \\ \kappa \in \mathit{Kont} &= \textbf{stmtK}\ \mathit{Stmt} \uplus \textbf{whileK}\ \mathit{Exp} \times \mathit{Stmt}^\star \end{aligned}$$

The major change is from the concrete *Locals*, mapping variables to integers, to the abstract $\mathit{Locals}^\sharp$, mapping variables to abstractions of integers (where the specific abstraction being used is left unspecified). Because of that change we then need to change *State*, which used concrete *Locals*, to $\mathit{State}^\sharp$, which uses the abstract $\mathit{Locals}^\sharp$. Notice that this definition of $\mathit{State}^\sharp$ (and the following definitions of the abstract transition rules and helper functions) allow us to plug in any integer abstraction we wish without having to change the abstract semantics.

2.2 Abstract Transition Rules

Table 1: The abstract transition relation. Each rule describes how to take one abstract state $(s^?, \hat{\rho}, \vec{\kappa})$ to the next abstract state $(s^?_{new}, \hat{\rho}_{new}, \overrightarrow{\kappa_{new}})$, where $\vec{\kappa} = \kappa \cdot \overrightarrow{\kappa_1}$. The $s^?$ notation means a statement may or may not exist; we use $\bullet$ to indicate that there is no statement. The $\cdot$ operator used for the continuation stack indicates appending sequences; thus κ is the top of the continuation stack in the source state and $\overrightarrow{\kappa_1}$ is the rest of that continuation stack.

no.	$s^?$	**premises**	$s^?_{new}$	$\hat{\rho}_{new}$	$\overrightarrow{\kappa_{new}}$
1	$x := e$	$[\![e]\!]^\sharp = \hat{n}$	$\bullet$	$\hat{\rho}[x \mapsto \hat{n}]$	$\vec{\kappa}$
2	**if** $e\ \overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$n \in \gamma_n([\![e]\!]^\sharp)$, $n \neq 0$	$\bullet$	$\hat{\rho}$	$\texttt{toSK}(\overrightarrow{s_1}) \cdot \vec{\kappa}$
3	**if** $e\ \overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$0 \in \gamma_n([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\texttt{toSK}(\overrightarrow{s_2}) \cdot \vec{\kappa}$
4	**while** $e\ \vec{s}$	$n \in \gamma_n([\![e]\!]^\sharp)$, $n \neq 0$	$\bullet$	$\hat{\rho}$	$\texttt{toSK}(\vec{s}) \cdot \textbf{whileK}(e, \vec{s}) \cdot \vec{\kappa}$
5	**while** $e\ \vec{s}$	$0 \in \gamma_n([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\vec{\kappa}$
6	$\bullet$	$\kappa = \textbf{stmtK}(s_1)$	s_1	$\hat{\rho}$	$\overrightarrow{\kappa_1}$
7	$\bullet$	$\kappa = \textbf{whileK}(e, \vec{s})$, $n \in \gamma_n([\![e]\!]^\sharp)$, $n \neq 0$	$\bullet$	$\hat{\rho}$	$\texttt{toSK}(\vec{s}) \cdot \vec{\kappa}$
8	$\bullet$	$\kappa = \textbf{whileK}(e, \vec{s})$, $0 \in \gamma_n([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\overrightarrow{\kappa_1}$

Notation. We use $[\![e]\!]^\sharp$ as shorthand for $\eta^\sharp(e, \hat{\rho})$ when $\hat{\rho}$ is obvious from context. We use $s^?$ to indicate 0 or 1 statements; $\bullet$ means there is no statement. For any map X, the notation $X[a \mapsto b]$ means a new map that is exactly the same as X except that a maps to b. We use the concretization function $\gamma_n : \mathbb{Z}^\sharp \to \mathcal{P}(\mathbb{Z})$ for the **if** and **while** rules to convert abstract integer values into a set of concrete integer values; we do it this way in the abstract semantics specification because we're leaving the abstract integer domain unspecified, and thus don't know what abstract integer values correspond to "zero" and "non-zero". In the actual abstract semantics implementation we wouldn't use γ_n (which can result in infinite sets of integers) but instead use something specific to the abstract integer domain that we're implementing.

2.3 Abstract Helper Functions

We describe the helper functions used by the transition rules. The functions are listed in alphabetical order.

2.3.1 $\eta^\sharp(e, \hat{\rho})$ **a.k.a.** $[\![e]\!]^\sharp$

This function describes how to evaluate expressions to abstract values. Note that sets of integers are evaluated by using the abstraction function for the particular integer abstract domain being used. Variables are looked up in the abstract locals map; binary operators recursively evaluate the operands and then apply the appropriate abstract operation to the result (the abstract operators are also specific to the abstract integer domain being used).

$\eta^\sharp : \mathit{Exp} \times \mathit{Locals}^\sharp \to \mathbb{Z}^\sharp$

$\eta^\sharp(e, \hat{\rho}) =$

$$\begin{cases} \alpha_n(\overline{n}) & \text{if } e = \overline{n} \\ \hat{\rho}(x) & \text{if } e = x \\ [\![e_1]\!]^\sharp \oplus [\![e_2]\!]^\sharp & \text{if } e = e_1 \oplus e_2 \end{cases}$$

Arithmetic Operators. $\{+, -, \times, \div\} : \mathbb{Z}^\sharp \times \mathbb{Z}^\sharp \to \mathbb{Z}^\sharp$ are specific to a particular abstract integer domain.

Relational Operators. $\{<, \leq, =, \neq\} : \mathbb{Z}^\sharp \times \mathbb{Z}^\sharp \to \mathbb{Z}^\sharp$ are specific to a particular abstract integer domain.

2.3.2 `initstate`

This function takes the program and generates the initial state. We use a given abstract integer domain's abstraction function to create the initial abstract locals map.

$\texttt{initstate} \in \mathit{Program} \to \mathit{State}^\sharp$

$\texttt{initstate}(p) = (\bullet, \hat{\rho}, \vec{\kappa})$ where

$p = \textbf{decl } \vec{x} \textbf{ in } \overrightarrow{s}$

$\hat{\rho} = [\ x_i \mapsto \alpha_n(0)\ \mid\ x_i \in \vec{x}\ ]$

$\vec{\kappa} = \texttt{toSK}(\vec{s})$

2.3.3 `toSK`

This function maps a sequence of statements to a sequence of **stmtK** continuations containing those statements.

$\texttt{toSK} \in \mathit{Stmt}^\star \to \mathit{Kont}^\star$

$\texttt{toSK}(\vec{s}) = \vec{\kappa}$ where $\kappa_i \in \vec{\kappa} = \textbf{stmtK}(s_i)$ for $0 \leq i < |\vec{s}|$