CSE 5095: Types and Programming Languages

IMP: Modeling Imperative Languages

Stefan Muller, partially based on material by Jim Sasaki

Lecture 19

1 IMP Syntax

We've spent much of the class discussing models of functional programming languages. Now, we'll switch to a model of imperative programming languages, IMP. We'll take our E language from much earlier in the semester, without let binding (many real imperative languages also include functions and structured data in their expression languages, but for simplicity, we won't) and add another layer of syntax, *statements*. Below, we'll use e for expressions.

Expressions	e	$::=$	x	Variables
		$\mid$	$\overline{n}$	Numbers
		$\mid$	"s"	Strings
		$\mid$	$e + e$	Addition
		$\mid$	$e \mathbin{\hat{}} e$	Concatenation
		$\mid$	$\lvert e \rvert$	String Length
		$\mid$	x	Variables
Types	τ	$::=$	$\mathsf{int} \mid \mathsf{string}$	
Statements	s	$::=$	$x := e$	Assignment
		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{fi}$	Conditional
		$\mid$	$\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$	Loop
		$\mid$	$s; s$	Sequence
		$\mid$	skip	Skip

A few notes on the syntax:

- We don't have let binding in expressions, but we still have variables in expressions; these are given values by assignment statements rather than let expressions.
- We can sequence statements together using semicolons: e.g. $s_1; s_2$. We can continue this with multiple statements—while it rarely matters semantically, we'll say that sequencing is right-associative, i.e. $s_1; s_2; s_3 \equiv s_1; (s_2; s_3)$.
- This makes a sequence of statements look almost like in C/C++/Java, where a semicolon ends a statement. Technically though, we shouldn't have a semicolon at the end (e.g., $s_1; s_2;$ is not syntactically valid).
- skip is a no-op statement. Why is this useful? Maybe you don't want an "else" branch of a conditional:

$$\mathsf{if}\ x < 0\ \mathsf{then}\ x := x + \overline{1}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}$$

- $x := e$ evaluates e and assigns the value to variable x.
- If and while statements work like in most major programming languages. Since we don't have Booleans, we'll treat positive integers as "true" and zero and negative integers as "false".

- As with expressions, we've omitted a lot in the name of keeping the language simple, but there's a lot of room for syntactic sugar. For example, want for loops? We can represent `for (x = 0; x < n; x++) { s }` as:

$$x := \overline{0}; \mathsf{while}\ x < n\ \mathsf{do}\ s; x := x + \overline{1}\ \mathsf{od}$$

Examples.

- The following program computes the Factorial of n (assuming n is positive): at the end of the program, $n \geq 0 \rightarrow r = n!$.

```
r := 1̄;
while (n ≥ 1)
do
   r := r ∗ n;
   n := n − 1
od
```

- We don't have assignment expressions (like in C, where `y = x++` assigns `x + 1` to `x` and returns the old `x` which is assigned to `y`. However, we can express this as

$$y := x; x := x + \overline{1}$$

2 Small-step Operational Semantics of Programs

We have a small-step semantics for expressions already, so today we'll define one for statements. We'll actually define the semantics not just over statements, but over "configurations", which are a pair of a statement and a state, which we'll write as $\langle S, \sigma \rangle$. A state σ is a mapping from variables to their values. We'll write a state as a set of variable-value pairs, e.g., $\sigma = \{x = 1, y = \text{"Hello"}\}$. Like with sets, we'll write the empty state as $\emptyset$. We'll write $\sigma(x)$ to mean the value of x in state σ. For the σ above, $\sigma(x) = 1$. We'll also use the following syntax to *update* or *extend* a state with a new value (or an updated value for a variable that's already in the state:

$$\sigma[x \mapsto e]$$

Note this doesn't actually change σ in any way. If σ is the state above, then $(\sigma[x \mapsto 2])(x) = 2$, but $\sigma(x)$ is still 1.

We'll write

$$\langle s_1, \sigma_1 \rangle \mapsto \langle s_2, \sigma_2 \rangle$$

to mean that in one step, if we're in state σ_1 and we execute s_1, the program changes to s_2 (this is a bit confusing; it'll become clearer in a bit) and the state changes to σ_2.

As an example,

$$\langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\} \rangle \mapsto \langle y := \overline{2}, \{x = 1, y = 0\} \rangle$$

(technically, this will actually be two steps with the rules we'll give, but this is just to give an idea).

We can keep going:

$$\langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\} \rangle \mapsto \langle y := \overline{2}, \{x = 1, y = 0\} \rangle \mapsto \langle \mathsf{skip}, \{x = 1, y = 2\} \rangle$$

We also have a judgment corresponding to e val for configurations: $\langle s, \sigma \rangle$ final means that the configuration has finished evaluating. Finally, we need to add the state σ to the semantics for expressions, since we'll need it to evaluate variables:

$$e \mapsto_\sigma e$$

We don't need configurations on both sides because evaluating expressions doesn't change the state. Nothing's stopping us from putting them, this is just less writing. We now need a step rule for variables, since they won't have been substituted for:

$$\frac{}{x \mapsto_\sigma \sigma(x)}\ (\textsc{StepVar})$$

All of the other small step rules for expressions are the same as before, they just thread the state through. Now for the rules for statements.

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle\ \mathsf{final}}\ (\textsc{FinalSkip}) \qquad \frac{e \mapsto_\sigma e'}{\langle x := e, \sigma\rangle \mapsto \langle x := e', \sigma\rangle}\ (\textsc{SStepSearchAssign})$$

$$\frac{e\ \mathsf{val}}{\langle x := e, \sigma\rangle \mapsto \langle \mathsf{skip}, \sigma[x \mapsto e]\rangle}\ (\textsc{SStepAssign}) \qquad \frac{\langle s_1, \sigma\rangle \mapsto \langle s_1', \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \mapsto \langle s_1'; s_2, \sigma'\rangle}\ (\textsc{SStepSearchSeq})$$

$$\frac{}{\langle \mathsf{skip}; s, \sigma\rangle \mapsto \langle s, \sigma\rangle}\ (\textsc{SStepSeq})$$

$$\frac{e \mapsto_\sigma e'}{\langle \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}, \sigma\rangle \mapsto \langle \mathsf{if}\ e'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}, \sigma\rangle}\ (\textsc{SStepSearchIf})$$

$$\frac{n > 0}{\langle \mathsf{if}\ \overline{n}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}, \sigma\rangle \mapsto \langle s_1, \sigma\rangle}\ (\textsc{SStepIfTrue}) \qquad \frac{n \leq 0}{\langle \mathsf{if}\ \overline{n}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}, \sigma\rangle \mapsto \langle s_2, \sigma\rangle}\ (\textsc{SStepIfFalse})$$

$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}, \sigma\rangle \mapsto \langle \mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, \sigma\rangle}\ (\textsc{SStepWhile})$$

Example 1 Let's formally apply the rules to the example above.

$$\begin{array}{ll} & \langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\}\rangle \\ \mapsto & \langle \mathsf{skip}; y := \overline{2}, \{x = 0, y = 0\}[x \mapsto 1]\rangle \\ \mapsto & \langle y := \overline{2}, \{x = 1, y = 0\}\rangle \\ \mapsto & \langle \mathsf{skip}, \{x = 1, y = 2\}\rangle \end{array}$$

Note that we write $\{x = 0, y = 0\}[x \mapsto 1]$ for clarity. This *doesn't* step to $\{x = 1, y = 0\}$, it is *equal to* $\{x = 1, y = 0\}$. We could just as easily have written it this way instead.

As we see, the first assignment actually steps to skip and then we use another rule (and another step) to remove that skip. This will get a little tiresome, so we can also write

$$\begin{array}{ll} & \langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\}\rangle \\ \mapsto^2 & \langle y := \overline{2}, \{x = 1, y = 0\}\rangle \\ \mapsto & \langle \mathsf{skip}, \{x = 1, y = 2\}\rangle \end{array}$$

where, as in E, $\mapsto^2$ means that this is actually 2 steps. We can use this to skip over "boring" steps. How do we know if a step is "boring"? There's no hard and fast rule, but if a step doesn't update the state, check a condition or do something else of interest like that, it's probably safe to skip. When in doubt, write out the steps.

Example 2 Below, let $W = \textsf{while}\ x\ \textsf{do}\ x := x - \overline{1}\ \textsf{od}$.

$$\begin{array}{ll}
 & \langle W, \{x = 1\}\rangle \\
\mapsto & \langle \textsf{if}\ x\ \textsf{then}\ x := x - \overline{1}; W\ \textsf{else skip fi}, \{x = 1\}\rangle \\
\mapsto & \langle \textsf{if}\ \overline{1}\ \textsf{then}\ x := x - \overline{1}; W\ \textsf{else skip fi}, \{x = 1\}\rangle \\
\mapsto & \langle x := x - \overline{1}; W, \{x = 1\}\rangle \\
\mapsto & \langle x := \overline{1} - \overline{1}; W, \{x = 1\}\rangle \\
\mapsto & \langle x := \overline{0}; W, \{x = 1\}\rangle \\
\mapsto^2 & \langle W, \{x = 1\}[x \mapsto 0]\rangle \\
\mapsto & \langle \textsf{if}\ x\ \textsf{then}\ x := x - \overline{1}; W\ \textsf{else skip fi}, \{x = 0\}\rangle \\
\mapsto & \langle \textsf{if}\ \overline{0}\ \textsf{then}\ x := x - \overline{1}; W\ \textsf{else skip fi}, \{x = 0\}\rangle \\
\mapsto & \langle \textsf{skip}, \{x = 0\}\rangle
\end{array}$$

3 Typing rules for IMP

We'll now present a type system for IMP. The type system for expressions is the same as before. Statements don't really have a type, but we still want to make sure they "make sense", e.g., if x has type $\textsf{int}$, we don't want to do $x :=$ "Hello". We'll use the judgment $\Gamma \vdash s\ \textsf{ok}$ to say that s is well-typed under the context Γ.

$$\frac{}{\Gamma \vdash \textsf{skip ok}}\ (\text{OKSkip}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma, x : \tau \vdash x := e\ \textsf{ok}}\ (\text{OKAssign})$$

$$\frac{\Gamma \vdash e : \textsf{int} \qquad \Gamma \vdash s_1\ \textsf{ok} \qquad \Gamma \vdash s_2\ \textsf{ok}}{\Gamma \vdash \textsf{if}\ e\ \textsf{then}\ s_1\ \textsf{else}\ s_2\ \textsf{fi ok}}\ (\text{OKIf}) \qquad \frac{\Gamma \vdash e : \textsf{int} \qquad \Gamma \vdash s\ \textsf{ok}}{\Gamma \vdash \textsf{while}\ e\ \textsf{do}\ s\ \textsf{od ok}}\ (\text{OKWhile})$$

$$\frac{\Gamma \vdash s_1\ \textsf{ok} \qquad \Gamma \vdash s_2\ \textsf{ok}}{\Gamma \vdash s_1; s_2\ \textsf{ok}}\ (\text{OKSeq})$$

We also need to know that a state is well-typed, i.e., that all the values have the right types. We'll say that $\Gamma \vdash \sigma$ if for all $x : \tau \in \Gamma$, we have $x = e \in \sigma$ for some e and $\cdot \vdash e : \tau$. Note that we don't need a context to type these values because they shouldn't have any variables in them (we say they are *closed*). So, for example,

$$x : \textsf{int}, y : \textsf{string} \vdash \{x = 5, y = \text{"Hello"}\}$$

but the following *is not* true:

$$x : \textsf{int}, y : \textsf{string} \vdash \{x = 5\}$$

4 Type Safety

Progress and preservation for expressions are mostly unchanged other than the variable case (because we changed the small-step rule for variables), and both also need the additional assumption that the state is well-typed.

We also need a lemma that values are well-typed under an empty context.

Lemma 1. *If* $\Gamma \vdash e : \tau$ *and* $e\ \textsf{val}$ *then* $\cdot \vdash e : \tau$.

Proof. By induction on the derivation of $e\ \textsf{val}$.

- ValNum. Then $e = \overline{n}$. By inversion, $\tau = \textsf{int}$. Apply TypeNum.
- ValString. Then $e =$ "s". By inversion, $\tau = \textsf{string}$. Apply TypeString.

□

Lemma 2 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \sigma$ *and* $e \mapsto_\sigma e'$ *then* $\Gamma \vdash e' : \tau$.

- StepVar. Then $e = x$ and $e' = \sigma(x)$. By inversion on TypeVar, $\Gamma(x) = \tau$. By the definition of $\Gamma \vdash \sigma$, we know $\cdot \vdash \sigma(x) : \tau$. By weakening, $\Gamma \vdash \sigma(x) : \tau$.

Lemma 3 (Progress). *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash \sigma$ then either e val or there exists e' such that $e \mapsto_\sigma e'$.*

- TypeVar. Then $e = x$ and $\Gamma(x) = \tau$. By the definition of $\Gamma \vdash \sigma$, we know that there exists some e' such that $\sigma(x) = e'$. By StepVar, $e \mapsto_\sigma e'$.

We've set things up so that type safety now also implies there are no unbound variables at runtime (assuming we start with a well-typed state)!

We need preservation and progress results for statements too. They'll use the progress and preservation results for expressions from above.

Lemma 4 (Preservation for Statements). *If $\Gamma \vdash s$ ok and $\Gamma \vdash \sigma$ and $\langle s, \sigma \rangle \mapsto \langle s', \sigma' \rangle$ then $\Gamma \vdash s'$ ok and $\Gamma \vdash \sigma'$*

Proof. By induction on the derivation of $\langle s, \sigma \rangle \mapsto \langle s', \sigma' \rangle$.

- SStepSearchAssign. Then $s = x := e$ and $s' = x := e'$ and $e \mapsto_\sigma e'$ and $\sigma' = \sigma$. By inversion, $x : \tau \in \Gamma$ and $\Gamma \vdash e : \tau$. By Lemma 2, $\Gamma \vdash e' : \tau$. Apply OKAssign. We have $\Gamma \vdash \sigma'$ because $\sigma' = \sigma$.

- SStepAssign. Then $s = x := e$ and e val and $s' = \mathsf{skip}$ and $\sigma' = \sigma[x \mapsto e]$. By inversion, $x : \tau \in \Gamma$ and $\Gamma \vdash e : \tau$. By Lemma 1, we have $\cdot \vdash e : \tau$. We therefore have $\Gamma \vdash \sigma'$. We have $\Gamma \vdash \mathsf{skip}$ ok by OKSkip.

- SStepSearchSeq. Then $s = s_1; s_2$ and $s = s_1'; s_2$ and $\langle s_1, \sigma \rangle \mapsto \langle s_1', \sigma' \rangle$. By inversion on TypeSeq, $\Gamma \vdash s_1$ ok and $\Gamma \vdash s_2$ ok. By induction, $\Gamma \vdash s_1'$ ok and $\Gamma \vdash \sigma'$. Apply rule TypeSeq.

- SStepSeq. Then $s = \mathsf{skip}; s'$ and $\sigma' = \sigma$, so $\Gamma \vdash \sigma'$. By inversion on TypeSeq, $\Gamma \vdash s'$ ok.

- SStepSearchIf. Then $s = \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}$ and $s' = \mathsf{if}\ e'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}$ and $e \mapsto_\sigma e'$ and $\sigma' = \sigma$. By inversion, $\Gamma \vdash e : \mathsf{int}$ and $\Gamma \vdash s_1$ ok and $\Gamma \vdash s_2$ ok. By Lemma 2, $\Gamma \vdash e' : \mathsf{int}$. Apply TypeIf.

- SStepIfTrue. Then $s = \mathsf{if}\ \overline{n}\ \mathsf{then}\ s'\ \mathsf{else}\ s_2\ \mathsf{fi}$ and $\sigma' = \sigma$. By inversion, $\Gamma \vdash s'$ ok.

- SStepIfFalse. Similar to above.

- SStepWhile. Then $s = \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$ and $\sigma' = \sigma$. and $s' = \mathsf{if}\ e\ \mathsf{then}\ (s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od})\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}$. By inversion, $\Gamma \vdash e : \mathsf{int}$ and $\Gamma \vdash s$ ok. By assumption, $\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$ ok. By TypeSeq, $\Gamma \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$ ok. By TypeSkip, $\Gamma \vdash \mathsf{skip}$ ok. By TypeIf, $\Gamma \vdash s'$ ok.

□

Lemma 5 (Progress for statements). *If $\Gamma \vdash s$ ok and $\Gamma \vdash \sigma$, then either $\langle s, \sigma \rangle$ final or there exist s' and σ' such that $\langle s, \sigma \rangle \mapsto \langle s', \sigma' \rangle$.*

Proof. By induction on the derivation of $\Gamma \vdash s$ ok.

- TypeSkip. Then $s = \mathsf{skip}$. By FinalSkip, $\langle s, \sigma \rangle$ final.

- TypeAssign. Then $s = x := e$ and $\Gamma = \Gamma', x : \tau$ and $\Gamma \vdash e : \tau$. By Lemma 3, either e val or $e \mapsto_\sigma e'$.
  - If e val, then $\langle s, \sigma \rangle \mapsto \langle \mathsf{skip}, \sigma[x \mapsto e] \rangle$ by SStepAssign.
  - If $e \mapsto_\sigma e'$, then $\langle s, \sigma \rangle \mapsto \langle x := e', \sigma \rangle$ by SStepSearchAssign.

- TypeIf. Then $s = \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}$ and $\Gamma \vdash e : \mathsf{int}$ and $\Gamma \vdash s_1$ ok and $\Gamma \vdash s_2$ ok. By Lemma 3, either e val or $e \mapsto_\sigma e'$.
  - If e val, then $\langle s, \sigma \rangle$ steps by SStepIfTrue or SStepIfFalse depending on the value of n.
  - If $e \mapsto_\sigma e'$, then $\langle s, \sigma \rangle \mapsto \langle \mathsf{if}\ e'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}, \sigma \rangle$ by SStepSearchIf.

- TypeWhile. Then $s = \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$, and $\langle s, \sigma \rangle$ steps by SStepWhile.
- TypeSeq. Then $s = s_1; s_2$ and $\Gamma \vdash s_1\ \mathsf{ok}$ and $\Gamma \vdash s_2\ \mathsf{ok}$. By induction, either $\langle s_1, \sigma \rangle\ \mathsf{final}$ or $\langle s_1, \sigma \rangle \mapsto \langle s_1', \sigma \rangle$.
  - If $\langle s_1, \sigma \rangle\ \mathsf{final}$, then by inversion on FinalSkip, $s_1 = \mathsf{skip}$. By SStepSeq, $\langle s, \sigma \rangle \mapsto \langle s_2, \sigma \rangle$.
  - If $\langle s_1, \sigma \rangle \mapsto \langle s_1', \sigma' \rangle$, then $\langle s, \sigma \rangle \mapsto \langle s_1'; s_2, \sigma' \rangle$ by SStepSearchSeq.

$\square$