Elements of Programming Languages

Lecture 13: Small-step semantics and type safety

James Cheney

University of Edinburgh

November 7, 2024

Overview

- For the remaining lectures we consider some *cross-cutting* considerations for programming language design.
  - Last time: Imperative programming
- Today:
  - Finer-grained (small-step) evaluation
  - Type safety

Refresher

- In the first 6 lectures we covered:
  - Basic arithmetic (L_{Arith})
  - Conditionals and booleans (L_{If})
  - Variables and let-binding (L_{Let})
  - Functions and recursion (L_{Rec})
  - Data structures (L_{Data})
- formalized using big-step evaluation ($e \Downarrow v$) and type judgments ($\Gamma \vdash e : \tau$)
- and implemented using Scala interpreters

Limitations of big-step semantics

- Big-step semantics is convenient, but also limited
- It says how to evaluate the "whole program" (expression) to its "final value"
- *But what if there is no final value?*
    - Expressions like $1 + \texttt{true}$ simply don't evaluate
    - Nonterminating programs don't evaluate either, but for a different reason!
- As we will see in later lectures, it is also difficult to deal with other features, like exceptions, using big-step semantics

Small-step semantics

- We will now consider an alternative: *small-step semantics*

$$e \mapsto e'$$

- which says how to evaluate an expression "one step at a time"
- If $e_0 \mapsto \cdots \mapsto e_n$ then we write $e_0 \mapsto^* e_n$. (in particular, for $n = 0$ we have $e_0 \mapsto^* e_0$)
- We want it to be the case that $e \mapsto^* v$ if and only if $e \Downarrow v$.
- But $\mapsto$ provides more detail about how this happens.
- It also allows expressions to "go wrong" (get stuck before reaching a value)

Small-step semantics: L_{Arith}

$e \mapsto e'$ for L_{Arith}

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1 \oplus e_2 \mapsto v_1 \oplus e_2'}$$

$$\frac{}{v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{v_1 \times v_2 \mapsto v_1 \times_{\mathbb{N}} v_2}$$

- If the first subexpression of $\oplus$ can take a step, apply it
- If the first subexpression is a value and the second can take a step, apply it
- If both sides are values, perform the operation
- Example:

$$1 + (2 \times 3) \mapsto 1 + 6 \mapsto 7$$

Small-step semantics: $\mathrm{L_{If}}$

$e \mapsto e'$ for $\mathrm{L_{If}}$

$$\frac{}{v == v \mapsto \mathtt{true}} \qquad \frac{v_1 \neq v_2}{v_1 == v_2 \mapsto \mathtt{false}}$$

$$\frac{e \mapsto e'}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto \mathtt{if}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}$$

$$\frac{}{\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_1}$$

$$\frac{}{\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_2}$$

- If the conditional test is not a value, evaluate it one step
- Otherwise, evaluate the corresponding branch

$$\begin{aligned} \mathtt{if}\ 1 == 2\ \mathtt{then}\ 3\ \mathtt{else}\ 4 &\mapsto \mathtt{if\ false\ then}\ 3\ \mathtt{else}\ 4 \\ &\mapsto 4 \end{aligned}$$

Small-step semantics: $\mathsf{L}_{\mathsf{Let}}$

$e \mapsto e'$ for $\mathsf{L}_{\mathsf{Let}}$

$$\frac{e_1 \mapsto e_1'}{\texttt{let } x = e_1 \texttt{ in } e_2 \mapsto \texttt{let } x = e_1' \texttt{ in } e_2}$$

$$\frac{}{\texttt{let } x = v_1 \texttt{ in } e_2 \mapsto e_2[v_1/x]}$$

- If the expression e_1 is not yet a value, evaluate it one step
- Otherwise, substitute it and proceed
- Example:

$$\begin{aligned} \texttt{let } x = 1 + 1 \texttt{ in } x \times x &\mapsto \texttt{let } x = 2 \texttt{ in } x \times x \\ &\mapsto 2 \times 2 \\ &\mapsto 4 \end{aligned}$$

Small-step semantics: $\mathsf{L}_{\mathsf{Lam}}$

$e \mapsto e'$ for $\mathsf{L}_{\mathsf{Lam}}$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}$$

$$\frac{}{(\lambda x.\ e)\ v \mapsto e[v/x]}$$

- If the function part is not a value, evaluate it one step
- If the function is a value and the argument isn't, evaluate it one step
- If both function and argument are values, substitute and proceed

$$\begin{aligned} ((\lambda x.\lambda y.x + y)\ 1)\ 2 &\mapsto (\lambda y.1 + y)\ 2 \\ &\mapsto 1 + 2 \mapsto 3 \end{aligned}$$

Small-step semantics: L_{Rec}

$e \mapsto e'$ for L_{Rec}

$$\frac{}{(\texttt{rec}\ f(x).\ e)\ v \mapsto e[\texttt{rec}\ f(x).e/f, v/x]}$$

- Same rules for evaluation inside application
- Note that we need to substitute $\texttt{rec}\ f(x).e$ for f.
- Suppose *fact* is the factorial function:

$$\begin{array}{rcl}
\mathit{fact}\ 2 & \mapsto & \texttt{if}\ 2 == 0\ \texttt{then}\ 1\ \texttt{else}\ 2 \times \mathit{fact}(2-1) \\
& \mapsto & \texttt{if}\ \texttt{false}\ \texttt{then}\ 1\ \texttt{else}\ 2 \times \mathit{fact}(2-1) \\
& \mapsto & 2 \times \mathit{fact}(2-1) \mapsto 2 \times \mathit{fact}(1) \\
& \mapsto & 2 \times (\texttt{if}\ 1 == 0\ \texttt{then}\ 1\ \texttt{else}\ 1 \times \mathit{fact}(1-1)) \\
& \mapsto & 2 \times (\texttt{if}\ \texttt{false}\ \texttt{then}\ 1\ \texttt{else}\ 1 \times \mathit{fact}(1-1)) \\
& \mapsto & 2 \times (1 \times \mathit{fact}(1-1)) \mapsto 2 \times (1 \times \mathit{fact}(0)) \\
& \mapsto^* & 2 \times (1 \times 1) \mapsto 2 \times 1 \mapsto 2
\end{array}$$

Judgments and Rules, in general

- A *judgment* is a relation among one or more abstract syntax trees.
- Examples so far: $e \Downarrow v$, $\Gamma \vdash e : \tau$, $e \mapsto e'$
- We have been defining judgments using *rules* of the form:

$$\frac{}{Q} \qquad \frac{P_1 \quad \cdots \quad P_n}{Q}$$

- where $P_1, \ldots, P_n$ and Q are judgments.

Meaning of Rules

- A rule of the form:

$$\overline{Q}$$

  is called an *axiom*. It says that Q is always derivable.
- A rule of the form

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

  says that judgment Q is derivable if $P_1, \ldots, P_n$ are derivable.
- Symbols like e, v, τ in rules stand for arbitrary expressions, values, or types.
- (Similar rules are a general basis for programming in Logic Programming languages like Prolog)

Rule induction

Induction on derivations of $e \Downarrow v$

Suppose $P(-,-)$ is a predicate over pairs of expressions and values. If:

- $P(v, v)$ holds for all values v
- If $P(e_1, v_1)$ and $P(e_2, v_2)$ then $P(e_1 + e_2, v_1 +_{\mathbb{N}} v_2)$
- If $P(e_1, v_1)$ and $P(e_2, v_2)$ then $P(e_1 \times e_2, v_1 \times_{\mathbb{N}} v_2)$

then $e \Downarrow v$ implies $P(e, v)$.

- Rule induction can be derived from mathematical induction on the size (or height) of the derivation tree.
- (Much like structural induction.)
- We won't formally prove this.

Example: $e \Downarrow v$ implies $e \mapsto^* v$

- As an example, we'll show a few cases of the forward direction of:

Theorem (Equivalence of big-step and small-step evaluation)

$e \Downarrow v$ if and only if $e \mapsto^* v$.

Base case.

If the derivation is of the form

$$\frac{}{n \Downarrow n}$$

for some number n, then $e = n$ is already a value $v = n$, so no steps are needed to evaluate it, i.e. $n \mapsto^* n$ in zero steps. □

Example: $e \Downarrow v$ implies $e \mapsto^* v$

Inductive case.

If the derivation is of the form

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2}$$

then by induction, we know $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$. Using the small-step rules, we can then show

$$e_1 + e_2 \mapsto^* v_1 + e_2 \mapsto^* v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2$$

□

- The case for $\times$ is similar.

Type soundness

- The central property of a type system is *soundness*.
- Roughly speaking, soundness means "well-typed programs don't go wrong" [Milner].
- But what exactly does "go wrong" mean?
  - For large-step: hard to say
  - For small-step: "go wrong" means "stuck" expression e that is not a value and cannot take a step.
- We could show something like:

Theorem (Value Soundness)

If $\vdash e : \tau$ *and* $e \mapsto^* v$ *then* $\vdash v : \tau$*.*

- This says that if an expression evaluates to a value, then the value has the right type.

Type soundness revisited

- We can decompose soundness into two parts:

Lemma (Progress)

If $\vdash e : \tau$ then e is not stuck: that is, either e is a value or for some e' we have $e \mapsto e'$.

Lemma (Preservation)

If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$

- Combining these two, can show:

Theorem (Soundness)

If $\vdash e : \tau$ then e is not stuck and if $e \mapsto^* e'$ then $\vdash e' : \tau$.

- We will *sketch* these properties for L_{If} (leaving out a lot of formal detail)

Progress for L_{If}

Progress is proved by induction on $\vdash e : \tau$ derivations. We show some representative cases.

Progress for $+$.

$$\frac{\vdash e_1 : \texttt{int} \quad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

If the derivation is of the above form, then by induction e_1 is either a value or can take a step, and likewise for e_2. There are three cases.

- If $e_1 \mapsto e_1'$ then $e_1 + e_2 \mapsto e_1' + e_2$.
- If e_1 is a value v_1 and $e_2 \mapsto e_2'$, then $v_1 + e_2 \mapsto v_1 + e_2'$.
- If both e_1 and e_2 are values then they must both be numbers $n_1, n_2 \in \mathbb{N}$, so $e_1 + e_2 \mapsto n_1 +_{\mathbb{N}} n_2$.

□

Progress for L_{If}

Progress for `if`.

If the derivation is of the form

$$\frac{\vdash e : \texttt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

then by induction, either e is a value or can take a step. There are two cases:

- If $e \mapsto e'$ then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$.
- If e is a value, it must be either `true` or `false`. In the first case, $\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mapsto e_1$, otherwise $\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \mapsto e_2$.

□

Preservation for L_{If}

Preservation is proved by induction on the structure of $\vdash e : \tau$. We'll consider some representative cases:

Preservation for $+$.

$$\frac{\vdash e_1 : \texttt{int} \quad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

If the derivation is of the above form, there are three cases.

- If $e_i = v_i$ and $v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2$ then obviously $\vdash v_1 +_{\mathbb{N}} v_2 : \texttt{int}$.
- If $e_1 + e_2 \mapsto e_1' + e_2$ where $e_1 \mapsto e_1'$, then since $\vdash e_1 : \texttt{int}$, we have $\vdash e_1' : \texttt{int}$, so $\vdash e_1' + e_2 : \texttt{int}$ also.
- The case where $e_1 = v_1$ and $v_1 + e_2 \mapsto v_1 + e_2'$ is similar.

$\square$

Preservation for L_{If}

Preservation for `if`.

If the derivation is of the form

$$\frac{\vdash e : \texttt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

then there are three cases:

- If $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ where $e \mapsto e'$, then by induction we can show that $\vdash e' : \texttt{bool}$ and $\vdash \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau$.
- If $e = \texttt{true}$ then $\texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto e_1$, so we already know $\vdash e_1 : \tau$.
- The case for $\texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto e_2$ is similar.

□

Type soundness for $\mathsf{L}_{\mathsf{Let}}$ [non-examinable]

- Progress: straightforward (a "let" can always take a step)
- Preservation: Suppose we have

$$\frac{\vdash v_1 : \tau' \quad x{:}\tau' \vdash e_2 : \tau}{\vdash \mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 : \tau} \qquad \frac{}{\mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 \mapsto e_2[v_1/x]}$$

We need to show that $\vdash e_2[v_1/x] : \tau$

- For this we need a *substitution lemma*

Lemma (Substitution)

If $\Gamma, x{:}\tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x] : \tau$

Type soundness for L_{Rec} [non-examinable]

- Progress: If an application term is well-formed:

$$\frac{\vdash e_1 : \tau_1 \to \tau_2 \quad \vdash e_2 : \tau_1}{\vdash e_1\ e_2 : \tau_2}$$

  then by induction, e_1 is either a value or $e_1 \mapsto e_1'$ for some e_1'. If it is a value, it must be either a lambda-expression or a recursive function, so $e_1\ e_2$ can take a step. Otherwise, $e_1\ e_2 \mapsto e_1'\ e_2$.
- Preservation: Similar to `let`, using substitution lemma for the cases

$$\begin{array}{rcl} (\lambda x.\ e)\ v & \mapsto & e[v/x] \\ (\mathtt{rec}\ f(x).\ e)\ v & \mapsto & e[\mathtt{rec}\ f(x).\ e/f, v/x] \end{array}$$

Summary

- Today we have presented
  - Small-step evaluation: a finer-grained semantics
  - Induction on derivations
  - Type soundness (details for L_{If})
  - Sketch of type soundness for L_{Rec} [Non-examinable]
- Deep breath: No more induction proofs from now on.
- Remaining lectures cover cross-cutting language features, which often have subtle interactions with each other
  - Next time: Imperative programming revisited: references, arrays and other resources.