Judgments, Rules, and Induction $_{\rm OOOOO}$

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

Elements of Programming Languages Lecture 13: Small-step semantics and type safety

James Cheney

University of Edinburgh

November 13, 2023

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

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

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

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 ↓ v) and type judgments (Γ ⊢ e : τ)
- and implemented using Scala interpreters

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

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 + 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: LArith



- 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:

$$1+(2\times3)\mapsto1+6\mapsto7$$

Type soundness

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

Small-step semantics: LIf

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

$$\overline{v==v\mapsto ext{true}} \qquad rac{v_1
eq v_2}{v_1==v_2\mapsto ext{false}}$$

 $e\mapsto e'$

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

if true then e_1 else $e_2\mapsto e_1$

if false then e_1 else $e_2\mapsto e_2$

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

if
$$1==2$$
 then 3 else $4 \ \mapsto$ if false then 3 else 4

$$\mapsto$$
 4

Type soundness

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

Small-step semantics: L_{Let}

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

$$e_1\mapsto e_1'$$

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

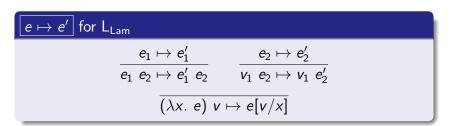
let
$$x = v_1$$
 in $e_2 \mapsto e_2[v_1/x]$

- If the expression e₁ is not yet a value, evaluate it one step
- Otherwise, substitute it and proceed
- Example:

$$\begin{array}{rrrr} \operatorname{let} x = 1 + 1 \text{ in } x \times x & \mapsto & \operatorname{let} x = 2 \text{ in } x \times x \\ & \mapsto & 2 \times 2 \\ & \mapsto & 4 \end{array}$$

Type soundness

Small-step semantics: L_{Lam}



- 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

$$((\lambda x.\lambda y.x + y) 1) 2 \mapsto (\lambda y.1 + y) 2$$
$$\mapsto 1 + 2 \mapsto 3$$

Small-step semantics: L_{Rec}

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

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

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

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

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

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:

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

• 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.
- (If you are familiar with Logic Programming: These rules are a lot like Prolog clauses!)

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

 $\overline{n \Downarrow n}$

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.

▲□▶ ▲□▶ ▲□▶ ▲□▶ □ のQで

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

Inductive case.

If the derivation is of the form

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

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

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

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:

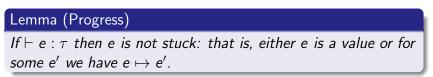
Theorem (Value Soundness)

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 (Preservation)

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

• Combining these two, can show:

Theorem (Soundness)

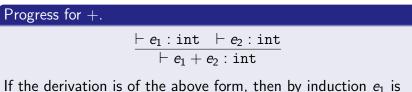
If $\vdash e : \tau$ then e is not stuck and if $e \mapsto^* e'$ then $\vdash e' : \tau$.

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

Type soundness

Progress for L_{If}

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



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 $e_1 + e_2 \mapsto n_1 +_{\mathbb{N}} n_2$.

Type soundness

Progress for L_{If}

Progress for if.

If the derivation is of the form

$$\begin{array}{c|c} -e: \texttt{bool} & \vdash e_1 : \tau & \vdash e_2 : \tau \\ \hline & \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau \end{array}$$

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

• If $e\mapsto e'$ then

if e then e_1 else $e_2 \mapsto \text{if } e'$ then e_1 else e_2 .

• If e is a value, it must be either true or false. In the first case, if true then e_1 else $e_2 \mapsto e_1$, otherwise if false then e_1 else $e_2 \mapsto e_2$.

Type soundness

Preservation for L_{If}

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

Preservation for +.

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

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 $\vdash v_1 +_{\mathbb{N}} v_2$: int.
- If $e_1 + e_2 \mapsto e'_1 + e_2$ where $e_1 \mapsto e'_1$, then since $\vdash e_1$: int, we have $\vdash e'_1$: int, so $\vdash e'_1 + e_2$: int also.
- The case where $e_1 = v_1$ and $v_1 + e_2 \mapsto v_1 + e_2'$ is similar.

Type soundness

Preservation for L_{If}

Preservation for if.

If the derivation is of the form

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

then there are three cases:

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

・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・

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

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

$$\vdash$$
 v₁ : τ' **x**: τ' \vdash **e**₂ : τ

 \vdash let $x = v_1$ in $e_2 : \tau$ let $x = v_1$ 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

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

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

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

(rec $f(x). e) v \mapsto e[rec f(x). e/f, v/x]$

Summary

- Today we have presented
 - Small-step evaluation: a finer-grained semantics
 - Induction on derivations
 - Type soundness (details for L_{lf})
 - 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.