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

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

Refresher

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

- 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: ** \( L_{If} \)

\[ e \mapsto e' \] for \( L_{If} \)

- If the conditional test is not a value, evaluate it one step
- Otherwise, evaluate the corresponding branch
- Example: 
  \[ \text{let } x = 1 + 1 \text{ in } x \times x \mapsto \text{let } x = 2 \text{ in } x \times x \mapsto 2 \times 2 \mapsto 4 \]

**Small-step semantics: ** \( L_{Let} \)

\[ e \mapsto e' \] for \( L_{Let} \)

- If the expression \( e_1 \) is not yet a value, evaluate it one step
- Otherwise, substitute it and proceed
- Example: 
  \[ \text{let } x = 1 + 1 \text{ in } x \times x \mapsto \text{let } x = 2 \text{ in } x \times x \mapsto 2 \times 2 \mapsto 4 \]
Small-step semantics: \( L_{\text{Lam}} \)

### for \( L_{\text{Lam}} \)

\[
\begin{align*}
\text{If the function part is not a value, evaluate it one step} \\
\text{If the function is a value and the argument isn't, evaluate it one step} \\
\text{If both function and argument are values, substitute and proceed}
\end{align*}
\]

\[
\begin{array}{c}
(\lambda x. e) v \mapsto e[v/x] \\
((\lambda x. \lambda y. x + y) 1) 2 \mapsto (\lambda y. 1 + y) 2 \\
\mapsto 1 + 2 \mapsto 3
\end{array}
\]

Small-step semantics: \( L_{\text{Rec}} \)

### for \( L_{\text{Rec}} \)

\[
\text{Same rules for evaluation inside application} \\
\text{Note that we need to substitute } \text{rec } f(x). e \text{ for } f.
\]

Suppose \( \text{fact} \) is the factorial function:

\[
\begin{align*}
\text{fact } 2 &\mapsto \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact}(2-1) \\
&\mapsto \text{if false then } 1 \text{ else } 2 \times \text{fact}(2-1) \\
&\mapsto 2 \times \text{fact}(2-1) \\
&\mapsto 2 \times (\text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact}(1-1)) \\
&\mapsto 2 \times (\text{if false then } 1 \text{ else } 1 \times \text{fact}(1-1)) \\
&\mapsto 2 \times (1 \times \text{fact}(1-1)) \mapsto 2 \times (1 \times \text{fact}(0)) \\
&\mapsto \ast \ 2 \times (1 \times 1) \mapsto 2 \times 1 \mapsto 2
\end{align*}
\]

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:

\[
\frac{P_1 \cdot \cdot \cdot P_n}{Q} \quad P_1 \cdot \cdot \cdot P_n
\]

where \( P_1, \ldots, P_n \) and \( Q \) are judgments.

Meaning of Rules

- A rule of the form:

\[
\frac{Q}{P_1 \ \cdot \cdot \cdot \ P_n}
\]

is called an **axiom**. It says that \( Q \) is always derivable.

- A rule of the form:

\[
\frac{P_1 \ \cdot \cdot \cdot \ P_n}{Q}
\]

says that judgment \( Q \) is derivable if \( P_1, \ldots, P_n \) are derivable.
- Symbols like \( e, v, \tau \) 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 $P(e_1 + e_2, v_1 +_N v_2)$
- If $P(e_1, v_1)$ and $P(e_2, v_2)$ then $P(e_1 \times e_2, v_1 \times_N v_2)$

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

$$n \Downarrow n$$

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.

**Inductive case.**

If the derivation is of the form

$\begin{align*} e_1 \Downarrow v_1 \\
 e_2 \Downarrow v_2 \\
 e_1 + e_2 \Downarrow v_1 +_N v_2 \end{align*}$

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

$$e_1 + e_2 \mapsto^* v_1 + v_2 \mapsto v_1 +_N v_2$$

- 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)**

\[ \text{If } \vdash e : \tau \text{ then } e \text{ is not stuck: that is, either } e \text{ is a value or for some } e' \text{ we have } e \mapsto e'. \]

**Lemma (Preservation)**

\[ \text{If } \vdash e : \tau \text{ and } e \mapsto e' \text{ then } \vdash e' : \tau \]

- Combining these two, can show:

**Theorem (Soundness)**

\[ \text{If } \vdash e : \tau \text{ then } e \text{ is not stuck and if } e \mapsto^* e' \text{ then } \vdash e' : \tau. \]

- We will sketch these properties for L_<text>if</text> (leaving out a lot of formal detail)

Progress for L_<text>if</text>

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

**Progress for +.**

\[
\begin{align*}
\vdash e_1 : \text{int} & \quad \vdash e_2 : \text{int} \\
\hline
\vdash e_1 + e_2 : \text{int}
\end{align*}
\]

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 \( e_1 + e_2 \mapsto n_1 + n_2 \).

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

**Preservation for +.**

\[
\begin{align*}
\vdash e_1 : \text{int} & \quad \vdash e_2 : \text{int} \\
\hline
\vdash e_1 + e_2 : \text{int}
\end{align*}
\]

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

- If \( e_1 = v_1 \) and \( v_1 + v_2 \mapsto v_1 + v_2 \) then obviously \( \vdash v_1 + v_2 : \text{int} \).
- If \( e_1 + e_2 \mapsto e'_1 + e_2 \) where \( e_1 \mapsto e'_1 \), then since \( \vdash e_1 : \text{int} \), we have \( \vdash e'_1 : \text{int} \), so \( \vdash e'_1 + e_2 : \text{int} \) also.
- The case where \( e_1 = v_1 \) and \( v_1 + e_2 \mapsto v_1 + e'_2 \) is similar.
Preservation for L_{if}

If the derivation is of the form
\[ \vdash e : \text{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau \]
\[ \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \]
then there are three cases:
- If \( \text{if } e \text{ then } e_1 \text{ else } e_2 \mapsto \text{if } e' \text{ then } e_1 \text{ else } e_2 \) where \( e \mapsto e' \), then by induction we can show that \( \vdash e' : \text{bool} \) and \( \vdash \text{if } e' \text{ then } e_1 \text{ else } e_2 : \tau \).
- If \( e = \text{true} \) then \( \text{if true then } e_1 \text{ else } e_2 \mapsto e_1 \), so we already know \( \vdash e_1 : \tau \).
- The case for \( \text{if false then } e_1 \text{ else } e_2 \mapsto e_2 \) is similar.

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

- Progress: straightforward (a “let” can always take a step)
- Preservation: Suppose we have
\[ \vdash v_1 : \tau' \quad x : \tau' \vdash e_2 : \tau \]
\[ \vdash \text{let } x = v_1 \text{ in } e_2 : \tau \]
\[ \text{let } x = v_1 \text{ in } e_2 \mapsto e_2[v_1/x] \]

We need to show that \( \vdash e_2[v_1/x] : \tau \)
- For this we need a substitution lemma

Lemma (Substitution)

\( \text{If } \Gamma, x : \tau' \vdash e : \tau \text{ and } \Gamma \vdash e' : \tau' \text{ then } \Gamma \vdash e[e'/x] : \tau \)

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

- Progress: If an application term is well-formed:
\[ \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \vdash e_2 : \tau_1 \]
\[ \vdash e_1 \ e_2 : \tau_2 \]
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
\( (\lambda x. \ e)\ v \mapsto e[v/x] \)
\( \text{rec } f(x). \ e\ v \mapsto e[\text{rec } f(x). \ e/f, v/x] \)

Summary

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