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 (CW1)

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 $1 + \text{true}$ 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
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)

\[ e \mapsto^* e' \] for \( L_{\text{Arith}} \)

\[ \begin{align*}
    e_1 \oplus e_2 & \mapsto^* e'_1 \oplus e'_2 \\
    v_1 \oplus v_2 & \mapsto v_1 \oplus_N v_2 \\
    v_1 \times v_2 & \mapsto v_1 \times_N v_2
\end{align*} \]

- 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:

\[ 1 + (2 \times 3) \mapsto 1 + 6 \mapsto 7 \]

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

\[ \begin{align*}
    e_1 \mapsto e'_1 \\
    \text{let } x = e_1 \text{ in } e_2 \mapsto \text{let } x = e'_1 \text{ in } e_2 \\
    \text{let } x = v_1 \text{ in } e_2 \mapsto e_2[v_1/x]
\end{align*} \]

- If the expression \( e_1 \) is not yet a value, evaluate it one step
- Otherwise, substitute it and proceed

Example:

\[ \begin{align*}
    \text{let } x = 1 + 1 \text{ in } x \times x & \mapsto \text{let } x = 2 \text{ in } x \times x \\
    & \mapsto 2 \times 2 \\
    & \mapsto 4
\end{align*} \]
Small-step semantics: \( L_{\text{Lam}} \)

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

\[
\begin{array}{c}
\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \\
\frac{e_2 \mapsto e_2'}{v_1 e_2 \mapsto v_1 e_2'} \\
(\lambda x. e) \mapsto e[v/x]
\end{array}
\]

- 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

\[ ((\lambda x. \lambda y. x + y) \; 1) \; 2 \mapsto ((\lambda y. 1 + y) \; 2 \mapsto 1 + 2 \mapsto 3 \]

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:

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

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

Meaning of Rules

- A rule of the form:

\[
\frac{Q}{Q}
\]

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

- A rule of the form

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

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 have taken Logic Programming: These rules are a lot like Prolog clauses!)

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

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

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

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

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

---

**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 (Soundness)**

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

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

---

**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)**

  \( e \downarrow v \text{ if and only if } e \mapsto v \).

**Base case.**

If the derivation is of the form

\[
\begin{align*}
  n \downarrow n
\end{align*}
\]

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

---

### Inductive case.

If the derivation is of the form

\[
\begin{align*}
  e_1 \downarrow v_1 & \quad e_2 \downarrow v_2 \\
  e_1 + e_2 \downarrow v_1 + N v_2
\end{align*}
\]

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

\[
\begin{align*}
  e_1 + e_2 \mapsto v_1 + v_2 \mapsto v_1 + N v_2
\end{align*}
\]

- The case for \( \times \) is similar.
Type soundness revisited

- We can decompose soundness into two parts:

**Lemma (Progress)**

\[ \text{If } \vdash e : \tau \then \text{either } e \text{ is a value or for some } e' \text{ we have } e \mapsto e'. \]

**Lemma (Preservation)**

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

- Combining these two, can show:

**Theorem (Soundness)**

\[ \text{If } \vdash e : \tau \text{ and } e \mapsto^* v \then \vdash v : \tau. \]

- 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 +.**

\[ \begin{array}{c}
\vdash e_1 : \text{int} \\
\vdash e_2 : \text{int}
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\vdash e_1 + e_2 : \text{int}
\end{array} \]

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 for L_{if}

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

**Preservation for +.**

\[ \begin{array}{c}
\vdash e_1 : \text{int} \\
\vdash e_2 : \text{int}
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\vdash e_1 + e_2 : \text{int}
\end{array} \]

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
\[
\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau
\]

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 \( \Gamma \vdash e' : \text{bool} \) and \( \Gamma \vdash \text{if } e' \text{ then } e_1 \text{ else } e_2 : \tau \).

- If \( e = \text{true} \) then if \( \text{true} \) then \( e_1 \) else \( e_2 \mapsto e_1 \), so we already know \( \Gamma \vdash e_1 : \tau \).

- The case for if \( e = \text{false} \) then if \( e_1 \) else \( e_2 \mapsto e_2 \) is similar.

Type soundness for \( L_{Rec} \)

**Progress**: If an application term is well-formed:
\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash e_1 \ e_2 : \tau_2
\]

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 \( \text{let} \), 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 proofs from now on.
- Remaining lectures cover cross-cutting language features, which often have subtle interactions with each other

- Next time: Guest lecture by Michel Steuwer on **DSLs and rewrite-based optimizations for performance-portable parallel programming**