CS152: Programming Languages
Lecture 11 — STLC Extensions and Related Topics

Dan Grossman
Spring 2011

Adding Stuff
Time to use STLC as a foundation for understanding other common language constructs

We will add things via a principled methodology thanks to a proper education

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar), or
  - Direct semantics
- Extend the type system
- Extend soundness proof (new stuck states, proof cases)

In fact, extensions that add new types have even more structure

Derived forms
let seems just like λ, so can make it a derived form

- **let** \( x = e_1 \) in \( e_2 \) “a macro” / “desugars to” \((\lambda x. e_2) \) \( e_1 \)
- A “derived form”

(Harder if \( \lambda \) needs explicit type)

Or just define the semantics to replace let with \( \lambda \):

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow ( \lambda x. e_2 ) \ e_1
\]

These 3 semantics are different in the state-sequence sense
\((e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n)\)
- But (totally) equivalent and you could prove it (not hard).

Note: ML type-checks let and \( \lambda \) differently (later topic)
Note: Don’t desugar early if it hurts error messages!

Let bindings (CBV)

\[
e ::= \ldots | \ let \ x = e_1 \ in \ e_2
\]

\[
e_1 \rightarrow e_1' | \ let \ x = e_1' \ in \ e_2
\]

\[
\text{let } x = v \text{ in } e \rightarrow e[v/x]
\]

\[
\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}
\]

(Also also need to extend definition of substitution...)

Progress: If \( e \) is a let, 1 of the 2 new rules apply (using induction)

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange

Booleans and Conditionals

\[
e ::= \ldots | \ true | \ false | \ if \ e_1 \ e_2 \ e_3
\]

\[
v ::= \ldots | \ true | \ false
\]

\[
\tau ::= \ldots | \ boolean
\]

\[
\frac{e_1 \rightarrow e_1'}{\quad \text{if true } e_2 \ e_3 \rightarrow e_2}
\]

\[
\frac{\quad \text{if false } e_2 \ e_3 \rightarrow e_3}{e_1 \rightarrow e_1' \quad \text{if } e_1 \ e_2 \ e_3}
\]

\[
\frac{\Gamma \vdash \text{true} : \ boolean \quad \Gamma \vdash \text{false} : \ boolean}{\Gamma \vdash \text{if true } e_2 \ e_3 : \tau}
\]

Also extend definition of substitution (will stop writing that)... Notes: CBN, new Canonical Forms case, all lemma cases easy
Pairs (CBV, left-right)

\[
\begin{align*}
    e & ::= \ldots | (e, e) | e.1 | e.2 \\
    v & ::= \ldots | (v, v) \\
    \tau & ::= \ldots | \tau \times \tau \\
    e_1 \to e_1' & \quad (e_1, e_2) \to (e_1', e_2) \\
    e_2 \to e_2' & \quad (v_1, e_2) \to (v_1', e_2') \\
    e \to e' & \quad (v_1, v_2).1 \to v_1 \\
    e.1 \to e'.1 & \quad (v_1, v_2).2 \to v_2
\end{align*}
\]

Small-step can be a pain

- Large-step needs only 3 rules
- Will learn more concise notation later (evaluation contexts)

---

Records

Records are like \(n\)-ary tuples except with named fields

- Field names are not variables; they do not \(\alpha\)-convert

\[
\begin{align*}
    e & ::= \ldots | \{ l_1 = e_1; \ldots; l_n = e_n \} | e.l \\
    v & ::= \ldots | \{ l_1 = v_1; \ldots; l_n = v_n \} \\
    \tau & ::= \ldots | \tau \times \tau \\
    e_i \to e_i' & \quad \{ l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, l_{i+1} = v_{i+1}, \ldots, l_n = v_n \} \\
    e \to e' & \quad \{ l_1 = v_1, \ldots, l_n = v_n \}.d_i \to v_i
\end{align*}
\]

\[1 \leq i \leq n\]

\[
\begin{align*}
    \Gamma \vdash e_1 : \tau_1 & \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \text{labels distinct} \\
    \Gamma \vdash \{ l_1 = e_1, \ldots, l_n = e_n \} : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \\
    \Gamma \vdash e : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \quad 1 \leq i \leq n
\end{align*}
\]

---

Sums syntax and overview

\[
\begin{align*}
    e & ::= \ldots | A(e) | B(e) | \text{match } e \text{ with } A.x. e | B.x. e \\
    v & ::= \ldots | A(v) | B(v) \\
    \tau & ::= \ldots | \tau_1 + \tau_2
\end{align*}
\]

- Only two constructors: \(A\) and \(B\)
- All values of any sum type built from these constructors
- So \(A(e)\) can have any sum type allowed by \(e\)’s type
- No need to declare sum types in advance
- Like functions, will “guess the type” in our rules
Sums operational semantics

\[
\begin{align*}
\text{match } A(v) \text{ with } A. & \quad e_1 & | & \text{By. } e_2 & \rightarrow e_1[v/x] \\
\text{match } B(v) \text{ with } A. & \quad e_1 & | & \text{By. } e_2 & \rightarrow e_2[v/y] \\
& \quad e & \rightarrow e' & A(e) & \rightarrow A(e') \\
& & & B(e) & \rightarrow B(e') \\
& \quad e & \rightarrow e' & \text{match } e \text{ with } A. & \quad e_1 & | & \text{By. } e_2 & \rightarrow e' \text{ with } A. & \quad e_1 & | & \text{By. } e_2
\end{align*}
\]

match has binding occurrences, just like pattern-matching
(Definition of substitution must avoid capture, just like functions)

What is going on

Feel free to think about tagged values in your head:

- A tagged value is a pair of:
  - A tag A or B (or 0 or 1 if you prefer)
  - The (underlying) value
- A match:
  - Checks the tag
  - Binds the variable to the (underlying) value

This much is just like Caml in lecture 1 and related to homework 2

Sums Typing Rules

Inference version (not trivial to infer; can require annotations)

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 & \Gamma & \vdash e : \tau_2 & \Gamma & \vdash A(e) : \tau_1 + \tau_2 & \Gamma & \vdash B(e) : \tau_1 + \tau_2 \\
\Gamma & \vdash e : \tau_1 + \tau_2 & \Gamma, x : \tau_1 & \vdash e_1 : \tau & \Gamma, y : \tau_2 & \vdash e_2 : \tau & \Gamma & \vdash \text{match } e \text{ with } A. & \quad e_1 & | & \text{By. } e_2 & \rightarrow \tau
\end{align*}
\]

Key ideas:

- For constructor-uses, “other side can be anything”
- For match, both sides need same type
  - Don’t know which branch will be taken, just like an if.
  - In fact, can drop explicit booleans and encode with sums:
    - E.g., bool = int + int, true = A(0), false = B(0)

Sums Type Safety

Canonical Forms: If \( \cdot \vdash v : \tau_1 + \tau_2 \), then there exists a \( v_1 \) such that either \( v = A(v_1) \) and \( \cdot \vdash v_1 : \tau_1 \) or \( v = B(v_1) \) and \( \cdot \vdash v_1 : \tau_2 \)

- Progress for \( \text{match } v \text{ with } A. & \quad e_1 & | & \text{By. } e_2 \) follows, as usual, from Canonical Forms
- Preservation for \( \text{match } v \text{ with } A. & \quad e_1 & | & \text{By. } e_2 \) follows from the type of the underlying value and the Substitution Lemma
- The Substitution Lemma has new “hard” cases because we have new binding occurrences
- But that’s all there is to it (plus lots of induction)

What are sums for?

- Pairs, structs, records, aggregates are fundamental data-builders
- Sums are just as fundamental: “this or that not both”
- You have seen how Caml does sums (datatypes)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another

Sums in C

\[
t = \text{of } t1 | \text{of } t2 | \text{of } t3
\]

match e with A x -> ...

One way in C:

\[
\begin{align*}
\text{struct } t \{ \\
\quad \text{enum } \{ A, B, C \} \quad \text{tag;} \\
\quad \text{union } \{ t1 a; t2 b; t3 c; \} \quad \text{data;} \\
\}; \\
\quad \ldots \text{switch(e->tag)}\{ \text{case } A: t1 x=e->data.a; \ldots \\
\}
\end{align*}
\]

- No static checking that tag is obeyed
- As fat as the fattest variant (avoidable with casts)
- Mutation costs us again!
Sums in Java

type t = A of t1 | B of t2 | C of t3
match e with A x -> ...

One way in Java (t4 is the match-expression’s type):
abstract class t {abstract t4 m();}
class A extends t { t1 x; t4 m();...}
class B extends t { t2 x; t4 m();...}
class C extends t { t3 x; t4 m();...}
... e.m() ...

- A new method in t and subclasses for each match expression
- Supports extensibility via new variants (subclasses) instead of extensibility via new operations (match expressions)

Pairs vs. Sums

You need both in your language
- With only pairs, you clumsily use dummy values, waste space, and rely on unchecked tagging conventions
- Example: replace int + (int -> int) with
  int * (int * (int -> int))

Pairs and sums are “logical duals” (more on that later)
- To make a τ1 * τ2 you need a τ1 and a τ2
- To make a τ1 + τ2 you need a τ1 or a τ2
- Given a τ1 * τ2, you can get a τ1 or a τ2 (or both; your “choice”)
- Given a τ1 + τ2, you must be prepared for either a τ1 or τ2 (the value’s “choice”)

Base Types and Primitives, in general

What about floats, string, ...?
Could add them all or do something more general...

Parameterize our language/semantics by a collection of base types \((b_1, \ldots, b_n)\) and primitives \((p_1 : \tau_1, \ldots, p_n : \tau_n)\). Examples:
- concat : string -> string
- toInt : float -> int
- “hello” : string
For each primitive, assume if applied to values of the right types it produces a value of the right type

Together the types and assumed steps tell us how to type-check and evaluate \(p_i v_1 \ldots v_n\) where \(p_i\) is a primitive.

We can prove soundness once and for all given the assumptions

Recursion

We probably won’t prove it, but every extension so far preserves termination

A Turing-complete language needs some sort of loop, but our lambda-calculus encoding won’t type-check, nor will any encoding of equal expressive power
- So instead add an explicit construct for recursion
- You might be thinking let rec \(f\ x = e\), but we will do something more concise and general but less intuitive

\[ e ::= \ldots | \mathrm{fix} \ e \]

\[ \mathrm{fix} \ e \rightarrow \mathrm{fix} \ e' \]
\[ \mathrm{fix} \ \lambda x. \ e \rightarrow e[\mathrm{fix} \ \lambda x. \ e/x] \]

No new values and no new types

Using fix

To use fix like let rec, just pass it a two-argument function where the first argument is for recursion
- Not shown: fix and tuples can also encode mutual recursion

Example:
\[
(\mathrm{fix} \ \lambda f. \ \lambda n. \ \text{if} \ (n < 1) \ 1 \ (n * (f(n - 1)))) \ 5
\rightarrow
(\lambda n. \ \text{if} \ (n < 1) \ 1 \ (n * (f(n - 1))))(n - 1)) \ 5
\rightarrow
\]
\[
(\text{if} \ (5 < 1) \ 1 \ (5 * (f(n - 1))))(5 - 1)
\rightarrow
5 \ 2
\rightarrow
5 \ ((\lambda n. \ \text{if} \ (n < 1) \ 1 \ (n * (f(n - 1))))(n - 1)) \ 4
\rightarrow
...
\]

Why called fix?

In math, a fix-point of a function \(g\) is an \(x\) such that \(g(x) = x\).
- This makes sense only if \(g\) has type \(\tau \rightarrow \tau\) for some \(\tau\)
- A particular \(g\) could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type \(\text{int} \rightarrow \text{int}\):
  - \(\lambda x. \ x + 1\) has no fix-points
  - \(\lambda x. \ x * 0\) has one fix-point
  - \(\lambda x. \ \text{absolute_value}(x)\) has an infinite number of fix-points
  - \(\lambda x. \ \text{if} \ (x < 10 \ \&\& \ x > 0) \ x \ 0\) has 10 fix-points
Higher types

At higher types like $\text{(int → int)} → \text{(int → int)}$, the notion of fix-point is exactly the same (but harder to think about)

- For what inputs $f$ of type $\text{int → int}$ is $g(f) = f$

Examples:

- $\lambda f. \lambda x. (f \, x) + 1$ has no fix-points
- $\lambda f. \lambda x. (f \, x) \ast 0$ (or just $\lambda f. \lambda x. 0$) has 1 fix-point
  - The function that always returns 0
  - In math, there is exactly one such function (cf. equivalence)
- $\lambda f. \lambda x. \text{absolute_value}(f \, x)$ has an infinite number of fix-points: Any function that never returns a negative result

Typing fix

\[
\begin{array}{c}
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash \text{fix } e : \tau
\end{array}
\]

Math explanation: If $e$ is a function from $\tau$ to $\tau$, then fix $e$, the fixed-point of $e$, is some $\tau$ with the fixed-point property.

- So it’s something with type $\tau$.

Operational explanation: fix $\lambda x. e'$ becomes $e'[\text{fix } \lambda x. e'/x]$

- The substitution means $x$ and fix $\lambda x. e'$ need the same type
- The result means $e'$ and fix $\lambda x. e'$ need the same type

Note: The $\tau$ in the typing rule is usually insatianted with a function type

- e.g., $\tau_1 \rightarrow \tau_2$, so $e$ has type $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$

Note: Proving soundness is straightforward!

Back to factorial

Now, what are the fix-points of $\lambda f. \lambda x. \text{if } (x < 1) 1 \ast (f(x - 1))$?

It turns out there is exactly one (in math): the factorial function!

And fix $\lambda f. \lambda x. \text{if } (x < 1) 1 \ast (f(x - 1))$ behaves just like the factorial function

- That is, it behaves just like the fix-point of $\lambda f. \lambda x. \text{if } (x < 1) 1 \ast (f(x - 1))$
- In general, fix takes a function-taking-function and returns its fix-point

(This isn’t really important, but I like explaining terminology and showing that programming is deeply connected to mathematics)

General approach

We added let, boolean, pairs, records, sums, and fix

- let was syntactic sugar
- fix made us Turing-complete by “baking in” self-application
- The others added types

Whenever we add a new form of type $\tau$ there are:

- Introduction forms (ways to make values of type $\tau$)
- Elimination forms (ways to use values of type $\tau$)

What are these forms for functions? Pairs? Sums?

When you add a new type, think “what are the intro and elim forms”?

Termination

Surprising fact: If $\cdot \vdash e : \tau$ in STLC with all our additions except fix, then there exists a $\nu$ such that $e \rightarrow^* \nu$

- That is, all programs terminate

So termination is trivially decidable (the constant “yes” function), so our language is not Turing-complete

The proof requires more advanced techniques than we have learned so far because the size of expressions and typing derivations does not decrease with each program step

- Might teach the proof in a future lecture, but more likely point you toward references if you’re interested

Non-proof:

- Recursion in $\lambda$ calculus requires some sort of self-application
- Easy fact: For all $\Gamma$, $x$, and $\tau$, we cannot derive $\Gamma \vdash x : \tau$