CSE505: Graduate Programming Languages

Lecture 11 — STLC Extensions and Related Topics

Dan Grossman
Winter 2012

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

Let bindings (CBV)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]

\[ e_1 \rightarrow e_1' \]

\[ \text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2 \]

\[ \Gamma \vdash e_1 : \tau' \mid \Gamma, x : \tau' \vdash e_2 : \tau \]

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

(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

Derived forms

\textbf{let} seems just like \( \lambda \), so can make it a derived form

- \textbf{let } \( x = e_1 \text{ in } e_2 \) “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 \textbf{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 \textbf{let} and \( \lambda \) differently (later topic)

Note: Don’t desugar early if it hurts error messages!

Booleans and Conditionals

\[ e ::= \ldots \mid \text{true} \mid \text{false} \mid \text{if } e_1 e_2 e_3 \]

\[ v ::= \ldots \mid \text{true} \mid \text{false} \]

\[ \tau ::= \ldots \mid \text{bool} \]

\[ e_1 \rightarrow e_1' \]

\[ \text{if true } e_2 e_3 \rightarrow e_2 \]

\[ \text{if false } e_2 e_3 \rightarrow e_3 \]

\[ \Gamma \vdash e_1 : \text{bool} \]

\[ \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash e_3 : \tau \]

\[ \Gamma \vdash \text{true} : \text{bool} \]

\[ \Gamma \vdash \text{false} : \text{bool} \]

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

\[
\begin{align*}
e & ::= \ldots | (e, e) | e.1 | e.2 \\
v & ::= \ldots | (v, v) \\
\tau & ::= \ldots | \tau * \tau
\end{align*}
\]

\[
\begin{align*}
e_1 \rightarrow e'_1 & \quad (e_1, e_2) \rightarrow (e'_1, e_2) \\
e_2 \rightarrow e'_2 & \quad (v_1, e_2) \rightarrow (v_1, e'_2) \\
e \rightarrow e' & \quad e.1 \rightarrow e'.1 \\
\end{align*}
\]

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

\[
\begin{align*}
e & ::= \ldots | \{ l_1 = e_1; \ldots ; l_n = e_n \} | e.1 \\
v & ::= \ldots | \{ l_1 = v_1; \ldots ; l_n = v_n \} \\
\tau & ::= \ldots | \{ l_1 : \tau_1; \ldots ; l_n : \tau_n \}
\end{align*}
\]

\[
\begin{align*}
el \rightarrow e' & \quad \{ l_1 = v_1; \ldots ; l_{i-1} = v_{i-1}; l_i = e_i; \ldots ; l_n = e_n \} \\
& \quad \rightarrow \{ l_1 = v_1; \ldots ; l_{i-1} = v_{i-1}; l_i = e_i'; \ldots ; l_n = e_n \} \\
1 \leq i \leq n \\
\Gamma \vdash e_1 : \tau_1 & \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \text{labels distinct} \\
\Gamma \vdash \{ l_1 = e_1; \ldots ; l_n = e_n \} : \{ l_1 : \tau_1; \ldots ; l_n : \tau_n \} \\
\Gamma \vdash e : \{ l_1 : \tau_1; \ldots ; l_n : \tau_n \} & \quad 1 \leq i \leq n \\
\Gamma \vdash e.1 : \tau_1 & \\
\end{align*}
\]

Sums

What about ML-style datatypes:

\[
\begin{align*}
t & = A | B \text{ of } \text{int} | C \text{ of } \text{int * t}
\end{align*}
\]

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., type ‘a mylist = \ldots)
4. Named types

For now, just model (1) with (anonymous) sum types

- (2) is in a later lecture, (3) is straightforward, and (4) we'll discuss informally

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

- 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 syntax and overview

- Can have any size
- No need to separate constructors
- Can use \(\text{Name}\) rather than \(\text{Tag}\)
Sums operational semantics

\[
\text{match } A(v) \text{ with } A.x. e_1 | B.y. e_2 \rightarrow e_1[v/x]
\]

\[
\text{match } B(v) \text{ with } A.x. e_1 | B.y. e_2 \rightarrow e_2[v/y]
\]

\[
e \rightarrow e'
\]

\[
\frac{\Gamma \vdash A(e) \rightarrow A'(e')} {\Gamma \vdash B(e) \rightarrow B'(e')}
\]

\[
e \rightarrow e'
\]

\[
\text{match } e \text{ with } A.x. e_1 | B.y. e_2 \rightarrow \text{match } e' \text{ with } A.x. e_1 | B.y. e_2.
\]

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 and related to homework 2

Sums Typing Rules

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

\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash A(e) : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash B(e) : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \text{match } e \text{ with } A.x. e_1 | B.y. e_2 : \tau}
\]

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., \(\text{bool} = \text{int} + \text{int}\), \(\text{true} = A(0)\), \(\text{false} = B(0)\)

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 \(\text{match } v \text{ with } A.x. e_1 | B.y. e_2\) follows, as usual, from Canonical Forms
- Preservation for \(\text{match } v \text{ with } A.x. e_1 | B.y. e_2\) 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

\[
t \equiv A \text{ of } t1 | B \text{ of } t2 | C \text{ of } t3
\]

match \(e\) with \(A x \rightarrow \ldots\)

One way in C:

\[
\text{struct } t \{
    \text{enum } \{A, B, C\} \tag;
    \text{union } \{t1 a; t2 b; t3 c;\} \text{ data; }
\};
\]

\[
\ldots \text{ switch}(e->tag)\{ \text{ case } A: t1 x=e->data.a; \ldots
\]

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

```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):

```java
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
  ```java
  int * (int * (int -> int))
  ```

Pairs and sums are “logical duals” (more on that later)
- To make a `t1 * t2` you need a `t1` and a `t2`
- To make a `t1 + t2` you need a `t1` or a `t2`
- Given a `t1 * t2`, you can get a `t1` or a `t2` (or both; your “choice”)
- Given a `t1 + t2`, you must be prepared for either a `t1` or `t2` (the value’s “choice”)

Base Types and Primitives, in general

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

Parameterize our language/semantics by a collection of base types (b1, ..., bn) and primitives (p1 : τ1, ..., pn : τn). Examples:
- `concat` : string→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 p1 v1 ... vn where p1 is a primitive

We can prove soundness once and for all given the assumptions

Recursion

We 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 ::= ... | fix e

fix e -> e'

fix λx. e -> e[fix λx. 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:

```java
(fix λf. λn. if (n<1) 1 (n * (f(n-1)))) 5

→

(λn. if (n<1) 1 (n * ((λf. λn. if (n<1) 1 (n * (f(n-1))))(n-1)))) 5

→

if (5<1) 1 (5 * ((λf. λn. if (n<1) 1 (n * (f(n-1))))(5-1))

→

if (5<1) 1 (5 * ((λf. λn. if (n<1) 1 (n * (f(n-1))))(5-1))

→

5 * ((λn. if (n<1) 1 (n * ((λf. λn. if (n<1) 1 (n * (f(n-1))))(n-1)))) 4)

→

...
```

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 τ → τ for some τ
- A particular g could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type `int → int`:
  ```java
  λx. x + 1 has no fix-points
  λx. x * 0 has one fix-point
  λx. absolute_value(x) has an infinite number of fix-points
  λx. if (x < 0) x 0 has 10 fix-points
  ```
Higher types

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

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

Examples:

- \(\lambda f. \lambda x. (f\ x) + 1\) has no fix-points
- \(\lambda f. \lambda x. (f\ x) \times 0\) (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 \(\text{fix}\)

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\Gamma &\vdash \text{fix} e : \tau
\end{align*}
\]

Math explanation: If \(e\) is a function from \(\tau\) to \(\tau\), then \(\text{fix} e\), the fixed-point of \(e\), is some \(\tau\) with the fixed-point property

- So it’s something with type \(\tau\)

Operational explanation: \(\text{fix} \lambda x. e’\) becomes \(e’[\text{fix} \lambda x. e’/x]\)

- The substitution means \(x\) and \(\text{fix} \lambda x. e’\) need the same type
- The result means \(e’\) and \(\text{fix} \lambda x. e’\) need the same type

Note: The \(\tau\) in the typing rule is usually insantiated 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!

Anonymity

We added many forms of types, all \textit{unnamed} a.k.a. \textit{structural}.

Many real PLs have (all or mostly) \textit{named} types:

- Java, C, C++: all record types (or similar) have names
- Omitting them just means compiler makes up a name
- Caml sum types and record types have names

A never-ending debate:

- Structural types allow more code reuse: good
- Named types allow less code reuse: good
- Structural types allow generic type-based code: good
- Named types let type-based code distinguish names: good

The theory is often easier and simpler with structural types

Back to factorial

Now, what are the fix-points of

\[\lambda f. \lambda x. (x < 1) 1 (x * (f(x - 1)))\]

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

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

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

(This isn’t necessarily important, but it explains the terminology and shows that programming is deeply connected to mathematics)

General approach

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

- \text{let} was syntactic sugar
- \text{fix} made us Turing-complete by “baking in” self-application
- The others \textit{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 \textit{fix}, then there exists a \(v\) such that \(e \rightarrow^* v\)

- 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

- Could present it in about an hour if desired

Non-proof:

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