CIS 624: Structure of Programming Languages
Lecture 11 — STLC Extensions and Related Topics

Boyana Norris
2015

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 ::= ... | let x = e₁ in e₂

$e₁ → e’₁$

$let x = e₁ in e₂ → let x = e’₁ in e₂$

$let x = v in e → e[v/x]$

Preservation: If $· ⊢ e : τ$, then $· ⊢ e’ : τ$.
Progress: If $· ⊢ e : τ$, then $e$ is a value or $∃ e’$ such that $e → e’$.

Derived forms

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

- let $x = e₁ in e₂$ "a macro" / “desugars to” $(λx. e₂) e₁$
- A "derived form"

(Harder if $λ$ needs explicit type)

Or just define the semantics to replace let with $λ$:

$let x = e₁ in e₂ → (λx. e₂) e₁$

These 3 semantics are different in the state-sequence sense ($e₁ → e₂ → ... → eₙ$)
- But (totally) equivalent and you could prove it (not hard)

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

Booleans and Conditionals

e ::= ... | true | false | if $e₁ e₂ e₃$

$v ::= ... | true | false$

$τ ::= ... | bool$

$e₁ → e₁’$

if $e₁ e₂ e₃ → e₂$

if $true e₂ e₃ → e₂$

if $false e₂ e₃ → e₃$

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

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

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

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 α-convert

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

\[
\begin{align*}
e_i \to e_i' \\
\{l_1=v_1, \ldots, l_{i-1}=v_{i-1}, l_i=e_i, \ldots, l_n=e_n\} \to \{l_1=v_1, \ldots, l_{i-1}=v_{i-1}, l_i=e_i', \ldots, l_n=e_n\} \\
e.l \to e'.l
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \\
\vdash e_i : \tau_i \\
\vdash e_n : \tau_n
\end{align*}
\]

Labels distinct
\[
\begin{align*}
\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
\quad \Gamma \vdash e.l : \tau_i
\end{align*}
\]

Sums

What about ML-style datatypes:

\[
\text{type } t = A | B \text{ of int } | C \text{ of int } \ast t
\]

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

Sums syntax and overview

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

▶ 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**

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

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

(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 OCaml and related to homework 2

**Sums Type Safety**

Canonical Forms: If \( \vdash v : \tau_1 + \tau_2 \), then there exists a \( v_1 \) such that either \( v = A(v_1) \) and \( \vdash v_1 : \tau_1 \) or \( v = B(v_2) \) and \( \vdash v_2 : \tau_2 \)

- Progress for \( \text{match } v \text{ with } Ax. e_1 | By. e_2 \text{ follows, as usual, from Canonical Forms} \)
- Preservation for \( \text{match } v \text{ with } Ax. e_1 | By. e_2 \text{ follows from the type of the underlying value and the Substitution Lemma} \)
- 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)

**Sums in C**

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

One way in C:

```
struct t {
    enum {A, B, C} tag;
    union {t1 a; t2 b; t3 c;} data;
};
... switch(e->tag){ case A: t1 x=e->data.a; ...
```

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

**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 OCaml does sums (datatypes)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another

**Sums Typing Rules**

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash A(e) : \tau_1 + \tau_2 & \quad \Gamma \vdash B(e) : \tau_1 + \tau_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau_1 + \tau_2 & \\
\Gamma, x: \tau_1 \vdash e_1 : \tau & \\
\Gamma, y: \tau_2 \vdash e_2 : \tau \\
\end{align*}
\]

\[
\Gamma \vdash \text{match } e \text{ with } Ax. e_1 | By. e_2 : \tau
\]

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., bool = int + int, true = A(0), false = B(0)
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 base types and primitives, in general:

- 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”)

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

```
fix e ::= ... | fix e
```

No new values and no new types

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:

- 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 < 10 & & 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
- \(\lambda f. \lambda x. (f \ x) \ast 0\) (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

\[
\Gamma \vdash e : \tau \rightarrow \tau \\
\Gamma \vdash \text{fix } e : \tau
\]

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

Anonymous

We added many forms of types, all unnamed a.k.a. structural.

Many real PLs have (all or mostly) named types:

- Java, C, C++: all record types (or similar) have names
- Omitting them just means compiler makes up a name
- OCaml 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 \(\lambda f. \lambda x. \text{if } (x < 1) 1 (x \ast (f(x - 1)))\)?

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

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

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

Termination

Surprising fact: If \(\vdash e : \tau\) in STLC with all our additions except \(\text{fix}\), 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

Non-proof:

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