CIS 624: Structure of Programming Languages

Lecture 11 — STLC Extensions and Related Topics

Boyana Norris
2017

Booleans and Conditionals

Derived forms

- Let bindings (CBV)

Let seems just like \( \lambda \), so can make it a derived form

- A "derived form"

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

These 3 semantics are different in the state-sequence sense

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!

Booleans and Conditionals

- true

- false

- if

Also extend definition of substitution (will stop writing that)...

Notes: CBN, new Canonical Forms case, all lemma cases easy
Pairs (CBV, left-right)

\[
e ::= \cdots | (e, e) | e.1 | e.2
\]

\[
v ::= \cdots | (v, v)
\]

\[
\tau ::= \cdots | \tau \ast \tau
\]

\[
e_1 \rightarrow e'_1
\]

\[
e_2 \rightarrow e'_2
\]

\[
(\tau_1, \tau_2) \rightarrow (\tau_1', \tau_2')
\]

\[
e \rightarrow e'
\]

\[
e.1 \rightarrow e'.1
\]

\[
(v_1, v_2) \rightarrow (v_1', v_2')
\]

Small-step can be a pain

\begin{itemize}
  \item Large-step needs only 3 rules
  \item Will learn more concise notation later (evaluation contexts)
\end{itemize}

Records

Records are like n-ary tuples except with named fields

\begin{itemize}
  \item Field names are not variables; they do not α-convert
  \item e ::= \cdots | \{l_1 = e_1; \ldots; l_n = e_n\} | e.l
  \item v ::= \cdots | \{l_1 = v_1; \ldots; l_n = v_n\}
  \item \tau ::= \cdots | \tau_1 \ast \tau_2
  \item 1 \leq i \leq n
  \item \{l_1 = v_1, \ldots, l_n = v_n\}.l_i \rightarrow v_i
  \item \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n \quad labels\ distinct
  \item \Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
  \item \Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad 1 \leq i \leq n
  \item \Gamma \vdash e.l_i : \tau_i
\end{itemize}

Sums

What about ML-style datatypes:

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

1. Tagged variants (i.e., discriminated unions)

2. Recursive types

3. Type constructors (e.g., type 'a mylist = ...)

4. Named types

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

\begin{itemize}
  \item (2) is in a later lecture, (3) is straightforward, and (4) we'll discuss informally
\end{itemize}

Pairs continued

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2
\]

\[
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2
\]

Canonical Forms: If \( \vdash v : \tau_1 \ast \tau_2 \), then \( v \) has the form \( (v_1, v_2) \)

Progress: New cases using Canonical Forms are \( v.1 \) and \( v.2 \)

Preservation: For primitive reductions, inversion gives the result directly

Records continued

\begin{itemize}
  \item Should we be allowed to reorder fields?
    \begin{itemize}
      \item \( \vdash \{l_1 = 42; l_2 = \text{true}\} : \{l_2 : \text{bool}; l_1 : \text{int}\} \) ??
      \item Really a question about, "when are two types equal?"
    \end{itemize}
  \end{itemize}

Nothing wrong with this from a type-safety perspective, yet many languages disallow it

\begin{itemize}
  \item Reasons: Implementation efficiency, type inference
\end{itemize}

Return to this topic when we study subtyping

Sums syntax and overview

\[
e ::= \cdots | A(e) | B(e) | \text{match } e \text{ with } A x.e | B x.e
\]

\[
v ::= \cdots | A(v) | B(v)
\]

\[
\tau ::= \cdots | \tau_1 \ast \tau_2
\]

\begin{itemize}
  \item Only two constructors: A and B
  \item All values of any sum type built from these constructors
  \item So \( A(e) \) can have any sum type allowed by \( e \)'s type
  \item No need to declare sum types in advance
  \item Like functions, will "guess the type" in our rules
\end{itemize}
Sums operational semantics

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

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

(Definition of substitution must avoid capture, just like functions)

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

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 Typing Rules

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

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

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

- Progress for match \( v \) with \( A x. e_1 \mid B y. e_2 \) follows, as usual, from Canonical Forms
- Preservation for match \( v \) with \( A x. e_1 \mid B y. e_2 \) 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 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 in C

\[
\text{type } t = A \text{ of } t_1 \mid B \text{ of } t_2 \mid C \text{ of } t_3 \\
\text{match } e \text{ with } A x \rightarrow \ldots
\]

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!
Sums in Java

```
<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
</tr>
</tbody>
</table>
```

One way in Java (t4 is the match-expression’s type):

```
abstract class t {
  abstract t 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

```
base_types & primitives, in general
```

```
Types and assumed steps tell us how to type-check and evaluate p1 v1 ... vn, where p1 is a primitive
```

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
```

```
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:

```
fix f x = e, but we will do something more concise and general but less intuitive
```

```
Recursion
```

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

```
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
```

Why called fix?

```
Boyana Norris
```
Higher types

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

- For what inputs \(f\) of type \(\text{int} \to \text{int}\) 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

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

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

Typing fix

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

Math explanation: If \(e\) is a function from \(\tau\) to \(\tau\), then \(\text{fix } e\), the fixed-point of \(e\), is some \(\tau\) with 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., \(\tau_1 \to \tau_2\), so \(e\) has type \((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)\)

Note: Proving soundness is straightforward!

Anonimy

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

Termination

Surprising fact: If \(\cdot \vdash e : \tau\) in STLC with all our additions except \(\text{fix}\), then there exists a \(v\) such that \(e \to^* v\)

- 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 \(\Gamma, x, \text{ and } \tau\), we cannot derive \(\Gamma \vdash x : \tau\)