CSE 505: Programming Languages

Lecture 13 — Safely Extending STLC: Sums, Products, Bools

Zach Tatlock
Fall 2013

Review

$$
\begin{array}{rcl}
e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\
v & ::= & \lambda x.\ e \mid c
\end{array}
\qquad
\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid \tau \to \tau \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

$$
\frac{}{(\lambda x.\ e)\ v \to e[v/x]}
\qquad
\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}
\qquad
\frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}
$$

$e[e'/x]$: capture-avoiding substitution of e' for free x in e

$$
\frac{}{\Gamma \vdash c : \mathsf{int}}
\qquad
\frac{}{\Gamma \vdash x : \Gamma(x)}
\qquad
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}
$$

Preservation: If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.
Progress: If $\cdot \vdash e : \tau$, then e is a value or $\exists\ e'$ such that $e \to e'$.

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

Pairs (CBV, left-right)

$$\begin{array}{lcl} e & ::= & \ldots \mid (e, e) \mid e.1 \mid e.2 \\ v & ::= & \ldots \mid (v, v) \\ \tau & ::= & \ldots \mid \tau * \tau \end{array}$$

$$\frac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)} \qquad \frac{e_2 \rightarrow e_2'}{(v_1, e_2) \rightarrow (v_1, e_2')}$$

$$\frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{}{(v_1, v_2).1 \rightarrow v_1} \qquad \frac{}{(v_1, v_2).2 \rightarrow v_2}$$

Small-step can be a pain

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

Pairs continued

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad\qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

Canonical Forms: If $\cdot \vdash v : \tau_1 * \tau_2$, 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

Records are like n-ary tuples except with *named fields*

- Field names are *not* variables; they do *not* α-convert

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l \\
v & ::= & \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\} \\
\tau & ::= & \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}
\end{array}
$$

$$
\frac{e_i \to e_i'}{\begin{array}{l}\{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i, \ldots, l_n{=}e_n\} \\ \to \{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i', \ldots, l_n{=}e_n\}\end{array}}
\qquad
\frac{e \to e'}{e.l \to e'.l}
$$

$$
\frac{1 \le i \le n}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n \qquad \text{labels distinct}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}
$$

$$
\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad 1 \le i \le n}{\Gamma \vdash e.l_i : \tau_i}
$$

Records continued

Should we be allowed to reorder fields?

- $\cdot \vdash \{l_1 = 42; l_2 = \textbf{true}\} : \{l_2 : \textbf{bool}; l_1 : \textbf{int}\}$??
- Really a question about, "when are two types equal?"

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

- Reasons: Implementation efficiency, type inference

Return to this topic when we study *subtyping*

Sums

What about ML-style datatypes:

```
type t = A | B of int | C of int * 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

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

Sums syntax and overview

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with } \mathsf{A}x.\ e \mid \mathsf{B}x.\ e \\
v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \\
\tau & ::= & \ldots \mid \tau_1 + \tau_2
\end{array}
$$

- Only two constructors: **A** and **B**
- All values of any sum type built from these constructors
- So $\mathsf{A}(e)$ 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

$$\frac{}{\textbf{match } \textbf{A}(v) \textbf{ with } \textbf{A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow e_1[v/x]}$$

$$\frac{}{\textbf{match } \textbf{B}(v) \textbf{ with } \textbf{A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow e_2[v/y]}$$

$$\frac{e \rightarrow e'}{\textbf{A}(e) \rightarrow \textbf{A}(e')} \qquad \frac{e \rightarrow e'}{\textbf{B}(e) \rightarrow \textbf{B}(e')}$$

$$\frac{e \rightarrow e'}{\textbf{match } e \textbf{ with } \textbf{A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow \textbf{match } e' \textbf{ with } \textbf{A}x.\ e_1 \mid \textbf{B}y.\ e_2}$$

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 OCaml and related to homework 2

Sums Typing Rules

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

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{match } e \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 : \tau}$$

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., **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 is $\mathbf{A}(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or v is $\mathbf{B}(v_1)$ and $\cdot \vdash v_1 : \tau_2$

- Progress for **match** v **with** $\mathbf{A}x.\ e_1 \mid \mathbf{B}y.\ e_2$ follows, as usual, from Canonical Forms
- Preservation for **match** v **with** $\mathbf{A}x.\ e_1 \mid \mathbf{B}y.\ e_2$ 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

```
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!

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 $\textbf{int} + (\textbf{int} \rightarrow \textbf{int})$ with $\textbf{int} * (\textbf{int} * (\textbf{int} \rightarrow \textbf{int}))$

Pairs and sums are "logical duals" (more on that later)

- To make a $\tau_1 * \tau_2$ you need a τ_1 *and* a τ_2
- To make a $\tau_1 + \tau_2$ you need a τ_1 *or* a τ_2
- Given a $\tau_1 * \tau_2$, you can get a τ_1 or a τ_2 (or both; your "choice")
- Given a $\tau_1 + \tau_2$, you must be prepared for either a τ_1 or τ_2 (the value's "choice")