Review

\[ e ::= \lambda x. e \mid x \mid ee \mid c \]
\[ v ::= \lambda x. e \mid c \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]
\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[ (\lambda x. e) v \rightarrow e[v/x] \]
\[ e_1 \rightarrow e'_1 \]
\[ e_2 \rightarrow e'_2 \]
\[ v e_2 \rightarrow v e'_2 \]

\( e[e'/x] \): capture-avoiding substitution of \( e' \) for free \( x \) in \( e \)

\[ \Gamma \vdash c : \text{int} \]
\[ \Gamma \vdash x : \Gamma(x) \]
\[ \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \]
\[ \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \]
\[ \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash e_1 \; e_2 : \tau_1 \]

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

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)

\[ e ::= \ldots \mid (e, e) \mid e.1 \mid e.2 \]
\[ v ::= \ldots \mid (v, v) \]
\[ \tau ::= \ldots \mid \tau \ast \tau \]

\[ e_1 \rightarrow e'_1 \]
\[ (e_1, e_2) \rightarrow (e'_1, e_2) \]
\[ e_2 \rightarrow e'_2 \]
\[ (v_1, e_2) \rightarrow (v_1, e'_2) \]
\[ e \rightarrow e' \]
\[ e.1 \rightarrow e'.1 \]
\[ e.2 \rightarrow e'.2 \]
\[ (v_1, v_2).1 \rightarrow v_1 \]
\[ (v_1, v_2).2 \rightarrow v_2 \]

Small-step can be a pain
- Large-step needs only 3 rules
- Will learn more concise notation later (evaluation contexts)
Pairs continued

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

Canonical Forms: If \( \cdot \vdash v : \tau_1 \ast \tau_2 \), 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

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

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

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

\[ e_i \rightarrow e'_i \]
\[ \{ l_1 = v_1, \ldots , l_{i-1} = v_{i-1}, l_i = e_i, \ldots , l_n = e_n \} \rightarrow \{ l_1 = v_1, \ldots , l_{i-1} = v_{i-1}, l_i = e'_i, \ldots , l_n = e_n \} \]

\[ 1 \leq i \leq n \]
\[ \{ l_1 = v_1, \ldots , l_n = v_n \}.l_i \rightarrow v_i \]

\[ \Gamma \vdash l_1 : \tau_1 \ldots \Gamma \vdash l_n : \tau_n \quad \text{labels distinct} \]
\[ \Gamma \vdash \{ l_1 = e_1, \ldots , l_n = e_n \} : \{ l_1 : \tau_1, \ldots , l_n : \tau_n \} \]
\[ \Gamma \vdash e : \{ l_1 : \tau_1, \ldots , l_n : \tau_n \} \quad 1 \leq i \leq n \]
\[ \Gamma \vdash e.l_i : \tau_i \]

Records continued

Should we be allowed to reorder fields?

- \( \cdot \vdash \{ l_1 = 42; l_2 = \text{true} \} : \{ l_2 : \text{bool}; l_1 : \text{int} \} ?? \)

- Really a question about, “when are two types equal?”

\textit{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:

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

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

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

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

\[
\tau := \ldots | \tau_1 + \tau_2
\]

- 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

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 operational semantics

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

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

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

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

\(\text{match}\) has binding occurrences, just like pattern-matching

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

Sums Typing Rules

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

\[
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash A(e) : \tau_1 + \tau_2 \\
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash B(e) : \tau_1 + \tau_2
\]

\[
\Gamma, x: \tau_1 \vdash e_1 : \tau \\
\Gamma, y: \tau_2 \vdash e_2 : \tau
\]

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

Key ideas:

- For constructor-uses, “other side can be anything”
- For \textit{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., \texttt{bool = int + int, true = A(0), false = B(0)}
Sums Type Safety

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

- Progress for match $v$ with $A x. e_1 | B y. e_2$ follows, as usual, from Canonical Forms

- Preservation for match $v$ 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 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 `int + (int → int)` with `int * (int * (int → int))`

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

- To make a $\tau_1 \ast \tau_2$ you need a $\tau_1$ and a $\tau_2$
- To make a $\tau_1 + \tau_2$ you need a $\tau_1$ or a $\tau_2$
- Given a $\tau_1 \ast \tau_2$, you can get a $\tau_1$ or a $\tau_2$ (or both; your “choice”)
- Given a $\tau_1 + \tau_2$, you must be prepared for either a $\tau_1$ or $\tau_2$ (the value’s “choice”)