Outline

- Continue extending ST\(\lambda\)C – data structures, recursion
- Discussion of “anonymous” types
- Consider termination informally
- Next time: Curry-Howard Isomorphism, Evaluation Contexts, Abstract Machines
Review

\[ e ::= \lambda x. \, e \mid x \mid e \, e \mid c \quad v ::= \lambda x. \, e \mid c \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \quad \Gamma ::= \cdot \mid \Gamma, x : \tau \]

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

\[ e[e'/x] : \text{capture-avoiding substitution of } e' \text{ for free } x \text{ in } e \]

\[ \Gamma \vdash c : \text{int} \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \]

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

\[ \Gamma \vdash e_1 \, e_2 : \tau_1 \]

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 \( e \rightarrow e' \).
Pairs (CBV, left-right)

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

\[
v ::= \ldots \mid (v, v)
\]

\[
\tau ::= \ldots \mid \tau \ast \tau
\]

\[
e_1 \rightarrow e_1'
\]

\[
\frac{}{(e_1, e_2) \rightarrow (e_1', e_2)}
\]

\[
e_2 \rightarrow e_2'
\]

\[
\frac{}{(v_1, e_2) \rightarrow (v_1, e_2')}
\]

\[
e \rightarrow e'
\]

\[
\frac{}{e.1 \rightarrow e'.1}
\]

\[
e \rightarrow e'
\]

\[
\frac{}{e.2 \rightarrow e'.2}
\]

\[
(v_1, v_2).1 \rightarrow v_1
\]

\[
(v_1, v_2).2 \rightarrow v_2
\]

Small-step can be a pain (more concise notation next lecture)
Pairs continued

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

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

\[
\Gamma \vdash e : \tau_1 \ast \tau_2
\]

\[
\Gamma \vdash e.1 : \tau_1 \quad \Gamma \vdash e.2 : \tau_2
\]

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

Progress: New cases using C.F. are \( v.1 \) and \( v.2 \).

Preservation: For primitive reductions, inversion gives the result \textit{directly}.
Records

Records seem like pairs with *named fields*

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

Fields do *not* $\alpha$-convert.

Names might let us reorder fields, e.g.,
\[
\begin{array}{c}
\cdot \vdash \{l_1 = 42; l_2 = \text{true}\} : \{l_2 : \text{bool}; l_1 : \text{int}\}.
\end{array}
\]

*Nothing wrong with this*, but many languages disallow it. (Why? Run-time efficiency and/or type inference)

(Caml has only *named* record types with *disjoint* fields.)

More on this when we study *subtyping*
Sums

What about ML-style datatypes:

```ml
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. Names the type

Today we'll model just (1) with (anonymous) sum types...
Sum syntax and overview

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

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

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

- 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
Sum semantics

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

\[
\begin{align*}
e & \rightarrow e' \\
A(e) & \rightarrow A(e') \\
B(e) & \rightarrow B(e')
\end{align*}
\]

\[
e \rightarrow e' \\
\text{match } e \text{ with } Ax. e_1 \mid By. e_2 & \rightarrow \text{match } e' \text{ with } Ax. e_1 \mid By. e_2
\]

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) and the value
- A match checks the tag and binds the variable to the value

This much is just like Caml in lecture 1 and related to homework 2.

Sums in other guises:

- C: use an *enum* and a *union*
  - More space than ML, but supports in-place mutation
- OOP: use an abstract superclass and subclasses
Sum Type-checking

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

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

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

Key ideas:

- For constructor-uses, “other side can be anything”
- For match, both sides need same type since don’t know which branch will be taken, just like an if.

Can encode booleans with sums. E.g., \texttt{bool} = \texttt{int} + \texttt{int}, \texttt{true} = A(0), \texttt{false} = B(0).
Type Safety

Canonical Forms: If $\cdot \vdash v : \tau_1 + \tau_2$, then either $v$ has the form $A(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or the form $B(v_1)$ and $\cdot \vdash v_1 : \tau_2$.

The rest is induction and substitution...
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 \rightarrow int)` with `int \ast (int \ast (int \rightarrow int))`

- “logical duals” (as we’ll see soon and the typing rules show)
  - 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”).
Base Types, in general

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

Parameterize our language/semantics by a collection of base types $(b_1, \ldots, b_n)$ and primitives $(c_1 : \tau_1, \ldots, c_n : \tau_n)$.

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 $c_i \ v_1 \ldots v_n$ where $c_i$ 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. What we add won’t be encodable in ST\(\lambda\)C.

E.g., \texttt{let rec f x = e}

Do typed recursive functions need to be bound to variables or can they be anonymous?

In Caml, you need variables, but it’s unnecessary:

\[
e ::= \ldots \mid \text{fix } e
\]

\[
\frac{e \rightarrow e'}{\text{fix } e \rightarrow \text{fix } e'} \quad \frac{\text{fix } \lambda x. \ e \rightarrow e[\text{fix } \lambda x. \ e/x]}{}
\]
Using fix

It works just like let rec, e.g.,

\[
\text{fix } \lambda f. \lambda n. \text{ if } n < 1 \text{ then } 1 \text{ else } n * (f(n - 1))
\]

Note: You can use it for mutual recursion too.
Pseudo-math digression

Why is it called fix? In math, a fixed-point of a function $g$ is an $x$ such that $g(x) = x$.

Let $g$ be $\lambda f. \lambda n. \text{if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1))$.

If $g$ is applied to a function that computes factorial for arguments $\leq m$, then $g$ returns a function that computes factorial for arguments $\leq m + 1$.

Now $g$ has type $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$. The fix-point of $g$ is the function that computes factorial for all natural numbers.

And $\text{fix } g$ is equivalent to that function. That is, $\text{fix } g$ is the fix-point of $g$. 

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 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 \( \text{fix } \lambda x. e' \) better have the same type. And the result means \( e' \) and \( \text{fix } \lambda x. e' \) better have the same type.

Note: Proving soundness is straightforward!
General approach

We added lets, booleans, pairs, records, sums, and fix. Let was syntactic sugar. Fix made us Turing-complete by “baking in” self-application. The others 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”? 
Anonymity

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)
- Caml sum-types have names.

A never-ending debate:

- Structural types allow more code reuse, which is good.
- Named types allow less code reuse, which is good.
- Structural types allow generic type-based code, which is good.
- Named types allow type-based code to distinguish names, which is good.

The theory is often easier and simpler with structural types.
Termination

Surprising fact: If $\cdot \vdash e : \tau$ in the ST$\lambda$C with all our additions except 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.

Proof is in the book. It requires cleverness because the size of expressions does not “go down” as programs run.

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