Review

\[ e ::= \lambda x. e \mid x \mid e \ e \mid c \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

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

\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[ (\lambda x. e) \ v \to e[v/x] \]

\[ e_1 \to e'_1 \]

\[ e_2 \to e'_2 \]

\[ e_1 \ e_2 \to e'_1 \ e_2 \]

\[ v \ e_2 \to v \ e'_2 \]

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

\[ \Gamma \vdash c : \text{int} \]

\[ \Gamma \vdash x : \Gamma(x) \]

\[ \Gamma \vdash \lambda x. e : \tau_1 \to \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_2 \to \tau_1 \]

\[ \Gamma \vdash e_2 : \tau_2 \]

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

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
Let bindings (CBV)

\[
e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2
\]

- \(e_1 \to e'_1\)
- \(\text{let } x = e_1 \text{ in } e_2 \to \text{let } x = e'_1 \text{ in } e_2\)
- \(\text{let } x = v \text{ in } e \to e[v/x]\)

\[
\Gamma \vdash e_1 : \tau' \\
\Gamma, x : \tau' \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\]

(Also need to extend definition of substitution...)

Progress: If \(e\) is a let, 1 of the 2 new rules apply (using induction)

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange
Derived forms

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

- **let** $x = e_1 \text{ in } e_2$ “a macro” / “desugars to” $(\lambda x. \ e_2) \ e_1$
- A “derived form”

(Harder if $\lambda$ needs explicit type)

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

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow (\lambda x. \ e_2) \ e_1
\]

These 3 semantics are *different* in the state-sequence sense

$(e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n)$

- 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

\[ e ::= \ldots | \text{true} | \text{false} | \text{if } e_1 \ e_2 \ e_3 \]

\[ v ::= \ldots | \text{true} | \text{false} \]

\[ \tau ::= \ldots | \text{bool} \]

\[ e_1 \rightarrow e'_1 \]

\[ \text{if } e_1 \ e_2 \ e_3 \rightarrow \text{if } e'_1 \ e_2 \ e_3 \]

\[ \text{if true } e_2 \ e_3 \rightarrow e_2 \]

\[ \text{if false } e_2 \ e_3 \rightarrow e_3 \]

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \]

\[ \Gamma \vdash \text{if } e_1 \ e_2 \ e_3 : \tau \]

\[ \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool} \]

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

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

\[ e ::= \ldots | (e, e) | e.1 | e.2 \]
\[ v ::= \ldots | (v, v) \]
\[ \tau ::= \ldots | \tau \times \tau \]

\[
\begin{align*}
  e_1 &\rightarrow e'_1 \\
  (e_1, e_2) &\rightarrow (e'_1, e_2)
\end{align*}
\]

\[
\begin{align*}
  e_2 &\rightarrow e'_2 \\
  (v_1, e_2) &\rightarrow (v_1, e'_2)
\end{align*}
\]

\[
\begin{align*}
  e &\rightarrow e' \\
  e.1 &\rightarrow e'.1
\end{align*}
\]

\[
\begin{align*}
  e &\rightarrow e' \\
  e.2 &\rightarrow e'.2
\end{align*}
\]

\[
\begin{align*}
  (v_1, v_2).1 &\rightarrow v_1 \\
  (v_1, v_2).2 &\rightarrow v_2
\end{align*}
\]

Small-step can be a pain

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

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1}
\]

\[
\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}
\]

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 \textit{directly}
Records

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

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

\[

e ::= \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l
\]

\[
v ::= \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\}
\]

\[
\tau ::= \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}
\]

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

\[
1 \leq i \leq n
\]

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

\[
\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \text{labels distinct}
\]

\[
\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\]

\[
\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad 1 \leq i \leq n}{\Gamma \vdash e.l_i : \tau_i}
\]
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?”

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

- Reasons: Implementation efficiency, type inference

Return to this topic when we study subtyping
Sums

What about ML-style datatypes:

```ocaml
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

- We’ll do (2) in a couple weeks, (3) is straightforward, and (4) we’ll discuss informally
Sums syntax and overview

\[ e ::= \ldots | A(e) | B(e) | \text{match } e \text{ with } Ax. \ e | Bx. \ e \]
\[ v ::= \ldots | A(v) | B(v) \]
\[ \tau ::= \ldots | \tau_1 + \tau_2 \]

- 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
match $A(v)$ with $Ax. \ e_1 \mid By. \ e_2 \rightarrow e_1[v/x]$

match $B(v)$ with $Ax. \ e_1 \mid By. \ e_2 \rightarrow e_2[v/y]$

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

match $e$ with $Ax. \ e_1 \mid By. \ e_2 \rightarrow$ match $e'$ 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)
  - The (underlying) value

- A match:
  - Checks the tag
  - Binds the variable to the (underlying) value

This much is just like Caml in lecture 1 and related to homework 2
Sums Typing Rules

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

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 + \tau_2 & \quad \Gamma, x:\tau_1 \vdash e_1 : \tau \\
\Gamma, y:\tau_2 \vdash e_2 : \tau & \quad \Gamma \vdash \text{match } e \text{ with } A x . \ e_1 \mid B y . \ e_2 : \tau
\end{align*}
\]

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., \texttt{bool} = \texttt{int + int}, \texttt{true} = A(0), \texttt{false} = B(0)
Sums Type Safety

Canonical Forms: If $\vdash v : \tau_1 \perp \tau_2$, 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 $\text{match } v \text{ with } Ax. \ e_1 \mid By. \ e_2$ follows, as usual, from Canonical Forms

- Preservation for $\text{match } v \text{ with } Ax. \ e_1 \mid By. \ e_2$ 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 Caml 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 \( \text{int} + (\text{int} \to \text{int}) \) with \( \text{int} \times (\text{int} \times (\text{int} \to \text{int})) \)

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

- To make a \( \tau_1 \times \tau_2 \) 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 \( \tau_1 \times \tau_2 \), 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 and Primitives, in general

What about floats, string, ...?
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 $(p_1 : \tau_1, \ldots, p_n : \tau_n)$. 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 $p_i \ v_1 \ldots v_n$ where $p_i$ is a primitive.

We can prove soundness once and for all given the assumptions
Recursion

We probably 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
Recursion

We probably 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 \texttt{let rec } \( f \ x = e \), but we will do something more concise and general but less intuitive.

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

\[ e \rightarrow e' \]

\[ \text{fix } e \rightarrow \text{fix } e' \]

\[ \text{fix } \lambda x. \ e \rightarrow e[\text{fix } \lambda x. \ e/x] \]

No new values and no new types.
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:

\[
(\text{fix } \lambda f. \lambda n. \text{ if } (n < 1) 1 (n \times (f(n - 1))))\ 5
\]
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:

\[(\text{fix } \lambda f. \lambda n. \text{if } (n<1) 1 (n \times (f(n - 1)))) 5\]

\[\rightarrow\]

\[(\lambda n. \text{if } (n<1) 1 (n \times ((\text{fix } \lambda f. \lambda n. \text{if } (n<1) 1 (n \times (f(n - 1))))(n - 1)))) 5\]
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:

\[
\text{\( (\text{fix} \ \lambda f. \ \lambda n. \ \text{if} \ (n < 1) \ 1 \ (n \ast (f(n - 1)))) \ 5 \) }
\]

\[
\rightarrow \\
\text{\( (\lambda n. \ \text{if} \ (n < 1) \ 1 \ (n \ast ((\text{fix} \ \lambda f. \ \lambda n. \ \text{if} \ (n < 1) \ 1 \ (n \ast (f(n - 1))))(n - 1)))) \ 5 \) }
\]

\[
\rightarrow \\
\text{\( \text{if} \ (5 < 1) \ 1 \ (5 \ast ((\text{fix} \ \lambda f. \ \lambda n. \ \text{if} \ (n < 1) \ 1 \ (n \ast (f(n - 1))))(5 - 1)) \) }
\]
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:

\[(\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n \ast (f(n - 1)))) \ 5\]

\[\rightarrow\]

\[(\lambda n. \text{if } (n < 1) 1 (n \ast ((\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n \ast (f(n - 1))))(n - 1)))) \ 5\]

\[\rightarrow\]

\[\text{if } (5 < 1) 1 (5 \ast ((\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n \ast (f(n - 1))))(5 - 1))\]

\[\rightarrow2\]

\[5 \ast ((\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n \ast (f(n - 1))))(5 - 1))\]
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:

\[(\text{fix } \lambda f. \lambda n. \text{ if } (n<1) 1 (n * (f(n - 1)))) 5\]

\[\rightarrow\]

\[(\lambda n. \text{ if } (n<1) 1 (n * ((\text{fix } \lambda f. \lambda n. \text{ if } (n<1) 1 (n * (f(n - 1))))(n - 1)))) 5\]

\[\rightarrow\]

\[\text{if } (5<1) 1 (5 * ((\text{fix } \lambda f. \lambda n. \text{ if } (n<1) 1 (n * (f(n - 1))))(5 - 1))\]

\[\rightarrow^2\]

\[5 * ((\text{fix } \lambda f. \lambda n. \text{ if } (n<1) 1 (n * (f(n - 1))))(5 - 1))\]

\[\rightarrow^2\]

\[5 * ((\lambda n. \text{ if } (n<1) 1 (n * ((\text{fix } \lambda f. \lambda n. \text{ if } (n<1) 1 (n * (f(n - 1))))(n - 1))))(n - 1)))) 4)\]

\[\rightarrow\]

\[\ldots\]
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 $\tau \to \tau$ for some $\tau$
- A particular $g$ could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type $\text{int} \to \text{int}$:
  - $\lambda x. x + 1$ has no fix-points
  - $\lambda x. x * 0$ has one fix-point
  - $\lambda x. \text{absolute\_value}(x)$ has an infinite number of fix-points
  - $\lambda x. \text{if} (x < 10 \&\& x > 0) x 0$ 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
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 \textbf{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, \textbf{fix} takes a function-taking-function and returns its fix-point

(This isn't really important, but I like explaining terminology and showing that programming is deeply connected to mathematics)
Typing \textbf{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 \(\text{fix } \lambda x. e'\) 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 instantiated 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!
General approach

We added let, 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 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 $\texttt{fix}$, 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

- Might teach the proof in a future lecture, but more likely point you toward references if you’re interested

Non-proof:

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