### Review

- **Structure and Syntax**
  - `e ::= ··· | let x= e1 in e2`
  - `e1 → e'1`
  - `let x=e1 in e2 → let x=e'1 in e2`
  - `let x=v in e → e[v/x]`
  - `Γ ⊢ e1 : τ'`  
  - `Γ, x : τ' ⊢ e2 : τ`  
  - `Γ ⊢ let x = e1 in e2 : τ`

**Preservation**: Uses Substitution Lemma

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

**Substitution Lemma**: Uses Weakening and Exchange

### 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

### Booleans and Conditionals

- `e ::= ··· | true | false | if e1 e2 e3`
- `v ::= ··· | true | false`
- `τ ::= ··· | bool`

**Preservation**

- \( e1 → e'1 \)
- \( \text{if } e1 e2 e3 \rightarrow e'1 e2 e3 \)

**Progress**

- \( \Gamma ⊢ \text{true} : τ \)
- \( \Gamma ⊢ \text{false} : τ \)

**Extended type system**

- \( \Gamma ⊢ e1 : \text{bool} \)
- \( \Gamma ⊢ e2 : τ \)
- \( \Gamma ⊢ e3 : τ \)

**Extended soundness proof**

- \( \Gamma ⊢ \text{true} : \text{bool} \)
- \( \Gamma ⊢ \text{false} : \text{bool} \)

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

Notes: CBN, new Canonical Forms case, all lemma cases easy

### Derived forms

- **let** seems just like \( \lambda \), so can make it a derived form
  - `let x = e1 in e2` "a macro" / "desugars to" (\( \lambda x. e2 \)) \( e1 \)

- A “derived form”

  (Harder if \( \lambda \) needs explicit type)

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

  - `let x = e1 in e2 → (\( \lambda x. e2 \)) e1`

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

  - \( e1 → e2 → \cdots → e_n \)
  - 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!
Sums

What about ML-style datatypes:

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

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

Pairs (CBV, left-right)

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

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

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

\[
e \rightarrow e'
\]

\[
\tau \rightarrow \tau'
\]

\[
\tau \rightarrow \tau'.1
\]

\[
\tau \rightarrow \tau'.2
\]

\[
e_1 \rightarrow e'_1
\]

\[
e_2 \rightarrow e'_2
\]

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

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

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

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

Pairs continued

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

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

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

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

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

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

\[
e ::= \cdots | \{ l_1 = e_1; \ldots; l_n = e_n \} \ | e.l
\]

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

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

\[
e \rightarrow e'
\]

\[
\tau \rightarrow \tau'
\]

\[
e.l \rightarrow e'.l
\]

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

\[
\Gamma \vdash e : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \quad \Gamma \vdash e_i : \tau_i
\]

\[
\Gamma \vdash e_i : \tau_i \quad \Gamma \vdash e_n : \tau_n \quad \text{labels distinct}
\]

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

Sums syntax and overview

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

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

\[
\tau ::= \cdots \mid \tau_1 \star \tau_2
\]

- 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

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 it
- Reasons: Implementation efficiency, type inference

Return to this topic when we study subtyping
Sums operational semantics

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

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

(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)

\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash A(e) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash B(e) : \tau_1 + \tau_2} \\
\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma, x:\tau_1 \vdash e_1 : \tau} \quad \Gamma, y:\tau_2 \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{match } e \text{ with } Ax. e_1 | By. e_2 : \tau
\]

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 bools and encode with sums:
    - E.g., bool = int + int, true = A(0), false = B(0)

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

\[
\text{type } t = A \text{ of } t_1 | B \text{ of } t_2 | C \text{ of } t_3
\]

\[
\text{match } e \text{ with } A \ x \rightarrow \ldots
\]

One way in C:

```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 Type Safety

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

- Progress for match $v$ with $Ax. e_1 | By. e_2$ follows, as usual, from Canonical Forms
- Preservation for match $v$ with $Ax. e_1 | By. e_2$ 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)
Sums in Java

```
type t = A of t1 | B of t2 | C of t3
```

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 τ1 + τ2 you need a τ1 and a τ2
- To make a τ1 + τ2 you need a τ1 or a τ2
- Given a τ1 + τ2, you can get a τ1 or a τ2 (or both; your "choice")
- Given a τ1 + τ2, you must be prepared for either a τ1 or τ2 (the value’s “choice”)

Base Types and Primitives, in general

What about floats, strings, ...?
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` to `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

Reursion

We 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

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

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:
```
(f x) (f y) (f z)
```

```
(f x) (f y) (f z)
```

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 \(\tau \rightarrow \tau\) for some \(\tau\)
- A particular \(g\) could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type \(\text{int} \rightarrow \text{int}\):
  - \(\lambda x. x + 1\) has no fix-points
  - \(\lambda x. x * 0\) has one fix-point
  - \(\lambda x. \text{abs}(x)\) 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 \((\text{int} \to \text{int}) \to (\text{int} \to \text{int})\), 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
- \(\lambda f. \lambda x. (f x) \ast 0\) (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 \(\text{fix}\ \lambda f. \lambda x. \text{if } (x < 1) 1 (x \ast (f(x - 1)))\) 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 \(\text{fix}\)

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

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 instantiatted 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!

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
- 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

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?”

Termination

Surprising fact: If \(\vdash e : \tau\) 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 \(\Gamma, x,\) and \(\tau\), we cannot derive \(\Gamma \vdash x : \tau\)