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)

<table>
<thead>
<tr>
<th>e ::= ...</th>
<th>let x = e₁ in e₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>e₁ → e'₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>let x = e₁ in e₂ = let x = e'₁ in e₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ e₁ : τ'</td>
</tr>
<tr>
<td>Γ, x : τ' ⊢ e₂ : τ</td>
</tr>
<tr>
<td>Γ ⊢ let x = e₁ in e₂ : τ</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>let x = e₁ in e₂ = (λx. e₂) e₁</th>
</tr>
</thead>
</table>

These 3 semantics are **different** in the state-sequence sense (e₁ → e₂ → ... → eₙ)

- But (totally) **equivalent** and you could prove it (not hard)

Note: ML type-checks let and λ differently (later topic)

Note: Don’t desugar early if it hurts error messages!

Booleans and Conditionals

<table>
<thead>
<tr>
<th>e ::= ...</th>
<th>true</th>
<th>false</th>
<th>if e₁ e₂ e₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>v ::= ...</td>
<td>true</td>
<td>false</td>
<td>-------------</td>
</tr>
<tr>
<td>τ ::= ...</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| e₁ → e'₁ |

<table>
<thead>
<tr>
<th>if true e₂ e₃ → e₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>if false e₂ e₃ → e₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Γ ⊢ e₁ : bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ e₂ : τ</td>
</tr>
<tr>
<td>Γ ⊢ e₃ : τ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Γ ⊢ true : bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ false : bool</td>
</tr>
</tbody>
</table>

Also extend definition of substitution (will stop writing that)... Notes: CBN, new Canonical Forms case, all lemma cases easy
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

Pairs (CBV, left-right)

```
e ::= ... | (e, e) | e.1 | e.2
v ::= ... | (v, v)
τ ::= ... | τ * τ
e1 → e'
1
(e1, e2) ... only 3 rules
▶ Will learn more concise notation later (evaluation contexts)
```

Pairs continued

```
Γ ⊢ e1 : τ1  Γ ⊢ e2 : τ2
Γ ⊢ (e1, e2) : τ1 ∗ τ2

Γ ⊢ e : τ1 ∗ τ2
Γ ⊢ e : τ1 : τ2

Canonical Forms: If ⊢ v : τ1 ∗ τ2, then v has the form (v1, v2)
```

Progress: New cases using Canonical Forms are v.1 and v.2

Preservation: For primitive reductions, inversion gives the result directly

Small-step can be a pain

▶ Large-step needs only 3 rules

Records

Records are like n-ary tuples except with named fields

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

```
e ::= ... | {l1 = e1; ...; ln = en} | e.l
v ::= ... | {l1 = v1; ...; ln = vn} | v.l
τ ::= ... | {l1 : τ1; ...; ln : τn}
```

```
e1 → e'
e1.l → e'.l
Γ ⊢ e1 : τ1 ... Γ ⊢ en : τn  labels distinct
Γ ⊢ {l1 = e1; ...; ln = en} : {l1 : τ1; ...; ln : τn}
Γ ⊢ e : {l1 : τ1; ...; ln : τn}  1 ≤ i ≤ n
Γ ⊢ e.l : τl
```

Records continued

Should we be allowed to reorder fields?

▶ l ⊢ {l1 = 42; l2 = true} : {l2 : bool; l1 : 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 syntax and overview

```
e ::= ... | A(e) | B(e) | match e with Ax. e | Bx. e
v ::= ... | A(v) | B(v)
τ ::= ... | τ1 + τ2
```

▶ 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
Sums operational semantics

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

\[
e \to e' \\
A(e) \to A(e') \\
B(e) \to B(e') \\
\]

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

(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

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 Typing Rules

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

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

Key ideas:
- For constructor-uses, “other side can be anything”
- For \text{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., \text{bool} = \text{int + int}, \text{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 \( v = A(v_1) \) and \( \cdot \vdash v_1 : \tau_1 \) or \( v = B(v_1) \) and \( \cdot \vdash v_1 : \tau_2 \)

- Progress for \text{match } v \text{ with } Ax. e_1 | By. e_2 follows, as usual, from Canonical Forms
- Preservation for \text{match } v \text{ 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 C

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

match e with A x -> ...

One way in C:

\[
\begin{align*}
\text{struct } t \{ \\
\text{enum } \{ A, B, C \} \text{ tag; } \\
\text{union } \{ t_1 a; t_2 b; t_3 c; \} \text{ data; } \\
\}; \\
\end{align*}
\]

\[
\begin{align*}
\ldots \text{ switch}(e->tag){ \text{ case } A: t_1 x=e->data.a; \ldots}
\end{align*}
\]

- 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)

**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 → 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 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`

\[
\begin{aligned}
& e ::= \ldots \mid \text{fix } e \\
& e \rightarrow e' \\
& \text{fix } e \rightarrow \text{fix } e' \\
& \text{fix } \lambda x. e \rightarrow e[\text{fix } \lambda x. e/x]
\end{aligned}
\]

No new values and no new types

**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 \(\text{int} + (\text{int} → \text{int})\) with \(\text{int} * (\text{int} * (\text{int} → \text{int}))\)

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

- To make a \(\tau_1 + \tau_2\) you need a \(\tau_1 \text{ and } \tau_2\)
- To make a \(\tau_1 + \tau_2\) you need a \(\tau_1 \text{ or } \tau_2\)
- Given a \(\tau_1 \text{ or } \tau_2\), you can get a \(\tau_1 \text{ or } \tau_2\) (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”)

**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:

\[
\begin{aligned}
& (\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n * (f(n-1)))) \ 5 \\
& \rightarrow (\lambda n. \text{if } (n < 1) 1 (n * (f(n-1))))(5-1)) 5 \\
& \rightarrow (5 * (\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n * (f(n-1))))(5-1)) \\
& \rightarrow 2^2 \\
& 5 * (\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n * (f(n-1))))(5-1)) \\
& \rightarrow 2^2 \\
& 5 * (\text{fix } \lambda f. \lambda n. \text{if } (n < 1) 1 (n * (f(n-1))))(5-1)) 4 \\
& \rightarrow ...
\end{aligned}
\]

**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
  - \(\lambda x. x \times 0\) 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

Boyana Norris  
CIS 624 2013, Lecture 11  
20
General approach

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 fix \( e \), the fixed-point of \( e \), is some \( \tau \) with the fixed-point property

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!

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

Termination

Surprising fact: If \( \cdot \vdash e : \tau \) in STLC 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

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 \( \Gamma \vdash x : \tau \)