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

Derived forms

let seems just like λ, so can make it a derived form

- let x = e₁ in e₂ “a macro” / “desugars to” (λx. e₂) e₁
- A “derived form”
  (Harder if λ needs explicit type)

Or just define the semantics to replace let with λ:

\[
\text{let } x = e₁ \text{ in } e₂ \rightarrow (\lambda x. e₂) e₁
\]

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

\[
\begin{align*}
e & ::= \ldots | \text{true} | \text{false} | \text{if } e₁ e₂ e₃ \\
v & ::= \ldots | \text{true} | \text{false} \\
τ & ::= \ldots | \text{bool} \\
\end{align*}
\]

\[
\begin{align*}
e₁ & \rightarrow e₁' \quad \text{if } e₁ e₂ e₃ \rightarrow e₂ \\
\quad \text{if } e₁ e₂ e₃ \rightarrow e₃ \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e₁ : \text{bool} & \quad \Gamma \vdash e₂ : τ \\
\Gamma \vdash e₂ : τ & \quad \Gamma \vdash e₃ : τ \\
\end{align*}
\]

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

\[
\begin{align*}
e & ::= \ldots \mid (e, e) \mid e.1 \mid e.2 \\
v & ::= \ldots \mid (v, v) \\
\tau & ::= \ldots \mid \tau \times \tau \\
e_1 \rightarrow e'_1 & \\
(\tau_1, \tau_2) & \rightarrow (\tau'_1, \tau'_2) \\
e & \rightarrow e' \\
\frac{e_1 \rightarrow e'_1}{(v_1, v_2).1 \rightarrow v_1} \\
\frac{e_2 \rightarrow e'_2}{(v_1, v_2).2 \rightarrow v_2}
\end{align*}
\]

Small-step can be a pain

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

Records

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

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

\[
\begin{align*}
e & ::= \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l \\
v & ::= \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\} \\
\tau & ::= \ldots \mid l_{1: \tau_1} \ldots l_{n: \tau_n} \\
e_i & \rightarrow e'_i \\
\frac{\{l_1 = v_1; \ldots; l_{i-1} = v_{i-1}; l_i = e_i; \ldots; l_n = e_n\}}{\{l_1 = v_1; \ldots; l_{i-1} = v_{i-1}; l_i = e'_i; \ldots; l_n = e_n\}} \\
\frac{1 \leq i \leq n}{l_i = v_i}
\end{align*}
\]

\(\Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n \)

Return to this topic when we study subtyping

Sums

What about ML-style datatypes:

\[
type\ t = A \mid B of\ int \mid C of\ int \times t
\]

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

\[
\begin{align*}
e & ::= \ldots \mid A(e) \mid B(e) \mid \mathbf{match\ e\ with\ A\ x.\ e\ |\ B\ x.\ e} \\
v & ::= \ldots \mid A(v) \mid B(v) \\
\tau & ::= \ldots \mid \tau_1 \times \tau_2
\end{align*}
\]

- 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

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

- Reasons: Implementation efficiency, type inference

Progress: New cases using Canonical Forms are \(v_1.1\) and \(v_2.2\)

Preservation: For primitive reductions, inversion gives the result directly
Sums operational semantics

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

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

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

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

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 OCaml and related to homework 2

Sums Typing Rules

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

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

Key ideas:

- For constructor-uses, "other side can be anything"
- For 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., 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 \( \text{match } v \text{ with } A.x. e_1 | B.y. e_2 \) follows, as usual, from Canonical Forms

- Preservation for \( \text{match } v \text{ with } A.x. e_1 | B.y. e_2 \) 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

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

match e with A x -> ...

One way in C:

```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(); ...} ...
```

- 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 \( \text{int} + (\text{int} \rightarrow \text{int}) \) with \( \text{int} \ast (\text{int} \rightarrow \text{int}) \)

- 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 \( \tau_1 + \tau_2 \), 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, 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:
  - \( \text{concat} : \text{string} \rightarrow \text{string} \)
  - \( \text{toInt} : \text{float} \rightarrow \text{int} \)
  - “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 assumptions 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

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

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

No new values and no new types

Using fix

To use \text{fix} like \text{let rec}, just pass it a two-argument function where the first argument is for recursion
  - Not shown: \text{fix} and tuples can also encode mutual recursion

Example:

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

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

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

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

\[
\ldots
\]

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 \( \tau \rightarrow \tau \) 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
    - \( \lambda x. x \ast 0 \) has one fix-point
    - \( \lambda x. \text{absolutex}(x) \) has an infinite number of fix-points
    - \( \lambda x. \text{if } (x < 10 \text{ and } x > 0) x \) 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
- \(\lambda f. \lambda x. (f \ x) \times 0\) (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 \(\lambda f. \lambda x. \text{if } (x < 1) 1 (x \times (f (x - 1)))\)?

It turns out there is exactly one (in math): the factorial function!

And \(\text{fix } \lambda f. \lambda x. \text{if } (x < 1) 1 (x \times (f (x - 1)))\) behaves just like the factorial function
- That is, it behaves just like the fix-point of \(\lambda f. \lambda x. \text{if } (x < 1) 1 (x \times (f (x - 1)))\)
- In general, \(\text{fix}\) takes a function-taking-function and returns its fix-point

(Given isn’t necessarily important, but it explains the terminology and shows that programming is deeply connected to mathematics)

Typing \(\text{fix}\)

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

Math explanation: If \(e\) is a function from \(\sigma\) 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 insantiated 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!

General approach
We added let, booleans, pairs, records, sums, and fix
- \(\text{let}\) was syntactic sugar
- \(\text{fix}\) made us Turing-complete by “baking in” self-application
- The others \(\text{added types}\)

Whenever we add a new form of type \(\sigma\) there are:
- Introduction forms (ways to make values of type \(\sigma\))
- Elimination forms (ways to use values of type \(\sigma\))

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 \(\text{unnamed}\) a.k.a. \(\text{structural}\).
Many real PLs have (all or mostly) \(\text{named types}\):
- 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 \(\cdot \vdash e : \sigma\) 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
- Could present it in about an hour if desired

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

Anonymity
We added many forms of types, all \(\text{unnamed}\) a.k.a. \(\text{structural}\).
Many real PLs have (all or mostly) \(\text{named types}\):
- 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