Programming Language Theory

Simply-Typed Lambda Calculus Extensions
Looking back, Looking forward

- Simply-Typed Lambda Calculus
  - syntax, small-step operational semantics, type system
  - proof of type soundness
- Today: Extend STLC (pairs, sums, recursion, . . .)
- Further ahead: References, exceptions, polymorphism, . . .
Simply-Typed Lambda Calculus (with constants)

\[ e ::= c \mid x \mid \lambda x.\ e \mid e\ e \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]
\[ v ::= c \mid \lambda x.\ e \]
\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[ e \rightarrow_{\text{cbv}} e' \]

\[ \text{E-Apply} \]
\[ (\lambda x.\ e_b)\ v_a \rightarrow_{\text{cbv}} e_b[v_a/x] \]

\[ \Gamma \vdash e : \tau \]

\[ \text{T-Const} \]
\[ \Gamma \vdash c : \text{int} \]

\[ \text{T-Var} \]
\[ \Gamma @ x \leadsto \tau \]

\[ \Gamma \vdash x : \tau \]

\[ \text{T-Lam} \]
\[ \Gamma, x : \tau_a \vdash e_b : \tau_r \]
\[ \Gamma \vdash \lambda x.\ e_b : \tau_a \rightarrow \tau_r \]

\[ \text{T-App} \]
\[ \Gamma \vdash e_f : \tau_a \rightarrow \tau_r \]
\[ \Gamma \vdash e_a : \tau_a \]
\[ \Gamma \vdash e_f\ e_a : \tau_r \]
Type Soundness: Main Theorem and Lemmas

A program that type checks does not get stuck.

Theorem (Type Safety): If $\cdot \vdash e : \tau$ and $e \rightarrow_{cbv}^* e'$, then either $e'$ is a value or there exists $e''$ such that $e' \rightarrow_{cbv} e''$.

Follows from two key lemmas:

- Lemma (Progress): If $\cdot \vdash e' : \tau$, then either $e'$ is a value or there exists an $e''$ such that $e' \rightarrow_{cbv} e''$.
- Lemma (Preservation): If $\cdot \vdash e : \tau$ and $e \rightarrow_{cbv} e^\dagger$, then $\cdot \vdash e^\dagger : \tau$.

Proof of Type Safety given Progress and Preservation:

- By induction on (the derivation) $e \rightarrow_{cbv}^* e'$.
  - $e \rightarrow_{cbv}^* e' \equiv e \rightarrow_{cbv}^* e$ : By Progress.
  - $e \rightarrow_{cbv}^* e' \equiv e \rightarrow_{cbv}^* e^\dagger \rightarrow_{cbv} e' :$ By Preservation and IH.
Type Soundness: Auxiliary Lemmas

Lemma (Canonical Forms): If $\cdot \vdash v : \tau$, then
1. if $\tau = \text{int}$, then $v = c$ (for some $c$)
2. if $\tau = \tau_1 \rightarrow \tau_2$, then $v = \lambda x. e$ (for some $\lambda x. e$)

Lemma (Substitution): If $\Gamma, x : \tau_x \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau_x$, then $\Gamma \vdash e_1[e_2/x] : \tau$.

Lemma (Exchange): If $\Gamma, x : \tau_x, y : \tau_y \vdash e : \tau$ and $x \neq y$, then $\Gamma, y : \tau_y, x : \tau_x \vdash e : \tau$.

Lemma (Weakening): If $\Gamma \vdash e : \tau$ and $x \notin \text{Dom}(\Gamma)$, then $\Gamma, x : \tau_x \vdash e : \tau$. 
Type Safety Proof Hierarchy

Type Safety: Well-typed programs never get stuck.

- Progress: Well-typed programs are values or can take a step.
  By induction on $\cdot \vdash e : \tau$.
  - Canonical Forms: “If it’s a duck, then it has feathers.”
    By inspection of typing rules.

- Preservation: Evaluation preserves the type.
  By induction on $\cdot \vdash e : \tau$, with cases on $e \rightarrow_{\text{cbv}} e'$.
  - Substitution: Things stay well-typed after substitution.
    - Exchange: Reordering variables in context is ok.
    - Weakening: Adding unused variables to context is ok.
Extending the Simply-Typed Lambda Calculus

Use STLC as a foundation for understanding other common language constructs.

Add things via a principled methodology:

- Extend Syntax: $e, v, \tau, \ldots$
  - derived forms (syntactic sugar)
- Extend Operational Semantics: $e \xrightarrow{\text{cbv}} e'$
  - direct semantics
- Extend Type System: $\Gamma \vdash e : \tau$
- Extend Proofs:
  - Progress, Canonical Forms, Preservation, Substitution

In fact, extensions that add new types have even more structure.
Let bindings (CBV)

\[ e ::= \cdots | \text{let } x = e_1 \text{ in } e_2 \]

\[ e_x \rightarrow_{\text{cbv}} e'_x \]

\[ \text{let } x = e_x \text{ in } e_b \rightarrow_{\text{cbv}} \text{let } x = e'_x \text{ in } e_b \]

\[ \text{let } x = v \text{ in } e_b \rightarrow_{\text{cbv}} e_b[v/x] \]

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

Progress: If \( e \) is a \texttt{let}, then one of the two rules above applies (using induction).

Preservation: Uses Substitution Lemma.

Derived forms

\texttt{let} seems very similar to \( \lambda \), so make it a derived form:

\begin{itemize}
  \item \texttt{let } \texttt{x = } \texttt{e}_x \texttt{ in } \texttt{e}_b \ “desugars to / macro expands to” \ (\lambda x. \ e_b) \ e_x
\end{itemize}

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

Or define the operational semantics to replace \texttt{let} with \( \lambda \):

\[
\texttt{let } x = e_x \texttt{ in } e_b \rightarrow \text{cbv} \ (\lambda x. \ e_b) \ e_x
\]

These 3 semantics are \textit{different} in the sequence of machine states:

\begin{itemize}
  \item \( e_1 \rightarrow_{\text{cbv}} e_2 \rightarrow_{\text{cbv}} \cdots \rightarrow_{\text{cbv}} e_n \)
\end{itemize}

But (totally) \textit{equivalent} and you could prove it (not hard).

Note: SML type-checks \texttt{let} and \( \lambda \) differently.
Note: Don’t desugar early if it hurts error messages!
Booleans and Conditionals

\[ e ::= \cdots \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \]

\[ \nu ::= \cdots \mid \text{true} \mid \text{false} \]

\[ \tau ::= \cdots \mid \text{bool} \]

\[
\frac{e_b \rightarrow_{\text{cbv}} e_b'}{\text{if } e_b \text{ then } e_t \text{ else } e_f \rightarrow_{\text{cbv}} \text{if } e_b' \text{ then } e_t \text{ else } e_f}
\]

\[
\frac{\text{if true then } e_t \text{ else } e_f \rightarrow_{\text{cbv}} e_t}{\text{if false then } e_t \text{ else } e_f \rightarrow_{\text{cbv}} e_f}
\]

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

\[ \Gamma \vdash e_b : \text{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau \]

\[ \Gamma \vdash \text{if } e_b \text{ then } e_t \text{ else } e_f : \tau \]

Notes: new Canonical Forms case; all lemma cases easy
Pairs

\[
\begin{align*}
e & ::= \cdots | (e, e) | e.1 | e.2 \\
\nu & ::= \cdots | (\nu, \nu) \\
\tau & ::= \cdots | \tau \ast \tau
\end{align*}
\]

\[
\begin{align*}
e_1 & \rightarrow_{\text{cbv}} e'_1 \\
& \quad \Rightarrow (e_1, e_2) \rightarrow_{\text{cbv}} (e'_1, e_2)
\end{align*}
\]

\[
\begin{align*}
e_2 & \rightarrow_{\text{cbv}} e'_2 \\
& \quad \Rightarrow (\nu_1, e_2) \rightarrow_{\text{cbv}} (\nu_1, e'_2)
\end{align*}
\]

\[
\begin{align*}
e_p & \rightarrow_{\text{cbv}} e'_p \\
& \quad \Rightarrow e_p.1 \rightarrow_{\text{cbv}} e'_p.1
\end{align*}
\]

\[
\begin{align*}
e_p & \rightarrow_{\text{cbv}} e'_p \\
& \quad \Rightarrow e_p.2 \rightarrow_{\text{cbv}} e'_p.2
\end{align*}
\]

\[
\begin{align*}
(\nu_1, \nu_2).1 & \rightarrow_{\text{cbv}} \nu_1
\end{align*}
\]

\[
\begin{align*}
(\nu_1, \nu_2).2 & \rightarrow_{\text{cbv}} \nu_2
\end{align*}
\]

Small-step has 6 rules; large-step needs only 3 rules.
Will learn more concise notation later (evaluation contexts).
Pairs (continued)

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

\[
\Gamma \vdash e_p : \tau_1 \ast \tau_2 \\
\Gamma \vdash e_p.1 : \tau_1 \\
\Gamma \vdash e_p.2 : \tau_2
\]

Canonical Forms: If \( \cdot \vdash v : \tau_1 \ast \tau_2 \), then \( v = (v_1, v_2) \) (for some \( v_1 \) and \( v_2 \)).

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

Preservation: For primitive reductions, use inversion.
Records

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

\[
e ::= \cdots | \{l_1 = e_1; \cdots ; l_n = e_n\} | e.l
\]
\[
v ::= \cdots | \{l_1 = v_1; \cdots ; l_n = v_n\}
\]
\[
\tau ::= \cdots | \{l_1: \tau_1; \cdots ; l_n: \tau_n\}
\]

\[
\frac{e \rightarrow_{cbv} e'}{e.l \rightarrow_{cbv} e'.l}
\]
\[
\frac{\{l_1 = v_1; \cdots ; l_i = v_i; \cdots ; l_n = v_n\}.l_i \rightarrow_{cbv} v_i}{\{l_1 = v_1; \cdots ; l_i = v_i; \cdots ; l_n = v_n\}.l_i \rightarrow_{cbv} v_i}
\]
\[
\frac{e_i \rightarrow_{cbv} e_i'}{\{l_1 = v_1; \cdots ; l_{i-1} = v_{i-1}; l_i = e_i; \cdots ; l_n = e_n\} \rightarrow_{cbv} \{l_1 = v_1; \cdots ; l_{i-1} = v_{i-1}; l_i = e_i'; \cdots ; l_n = e_n\}}
\]

\[
\Gamma \vdash e : \{l_1: \tau_1; \cdots ; l_n: \tau_n\} \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n
\]
\[
\Gamma \vdash e.l_i : \tau_i
\]
\[
\Gamma \vdash \{l_1 = e_1; \cdots ; l_n = e_n\} : \{l_1: \tau_1; \cdots ; l_n: \tau_n\}
\]

Fields in a record or record type should be distinct.
Fields do \textit{not} \( \alpha \)-convert.
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, but many languages disallow it.

- (Why? Run-time efficiency and/or type inference.)

More on records when we study subtyping.
Sums

What about ML-style datatypes:

```plaintext
datatype t = A | B of int | C of int * t
```

Combine many features in one:

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., `datatype 'a mylist = ···`)
4. Named types

For now, just model (1) with (anonymous) sum types.

- (2) in a later lecture
- (3) has an easy way (macros) and a hard way (higher-order polymorphism)
- (4) to be discussed informally
Sums syntax and overview

\[
e ::= \cdots | L(e) | R(e) | \text{case } e \text{ of } L(x) \Rightarrow e | R(x) \Rightarrow e
\]

\[
v ::= \cdots | L(v) | R(v)
\]

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

- Only two constructors: \( L \) and \( R \)
- All values of any sum type built from these constructors
- So \( L(e) \) 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

\[
\text{case } L(v) \text{ of } L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \rightarrow_{\text{cbv}} e_1[v/x]
\]

\[
\text{case } R(v) \text{ of } L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \rightarrow_{\text{cbv}} e_2[v/y]
\]

\[
e_s \rightarrow_{\text{cbv}} e'_s
\]

\[
\text{case } e_s \text{ of } L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \rightarrow_{\text{cbv}} \text{case } e'_s \text{ of } L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2
\]

\[
e \rightarrow_{\text{cbv}} e'
\]

\[
L(e) \rightarrow_{\text{cbv}} L(e')
\]

\[
e \rightarrow_{\text{cbv}} e'
\]

\[
R(e) \rightarrow_{\text{cbv}} R(e')
\]

Note: \text{case} has binding occurrences, just like pattern-matching.

(Definition of substitution must avoid capture, just like functions.)
Sums operational semantics

Feel free to think about tagged values in your head:

- A tagged value is a pair of
  - a tag (L or R (or 0 or 1, if you prefer))
  - the (underlying) value

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

This much is just like SML.
Sums type system

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \quad & \Gamma \vdash e : \tau_2 \\
\Gamma \vdash L(e) : \tau_1 + \tau_2 & \quad & \Gamma \vdash R(e) : \tau_1 + \tau_2 \\
\Gamma \vdash e : \tau_1 + \tau_2 & \quad & \Gamma, x:\tau_1 \vdash e_1 : \tau \quad \Gamma, y:\tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case } e \text{ of } L(x) => e_1 \mid R(y) => e_2 : \tau
\end{align*}
\]

Key ideas:

- For constructors, “other side can be anything”
  - Like functions, “guess” the other type
  - Not trivial to infer; can require annotations

- For match, both sides need same type
  - don’t know which branch will be taken, just like an if.

Can encode booleans with sums:

- \texttt{bool} = \texttt{int} + \texttt{int}, \texttt{true} = L(0), \texttt{false} = R(0).
Sums type safety

Canonical Forms:
If \( \cdot \vdash v : \tau_1 + \tau_2 \), then either \( v = L(v') \) (for some \( v' \)) or \( v = R(v') \) (for some \( v' \)).

- Progress for case \( e \) of \( L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \) follows, as usual, from Canonical Forms.
- Preservation for case \( e \) of \( L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \) follows from the type of the underlying value and from Substitution.
- Substitution has new “hard” cases.
  because case \( e \) of \( L(x) \Rightarrow e_1 \mid R(y) \Rightarrow e_2 \) has new binding occurrences.
- That’s all (plus lots of induction).
What are sums for?

- Pairs, structs, records, aggregates are fundamental data-builders
- Sums are just as fundamental: “this or that (and not both)”
- We have seen how SML does sums (datatype)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another
Sums in C

datatype \texttt{t} = A of \texttt{t1} \mid B of \texttt{t2} \mid C of \texttt{t3}

case \texttt{e} of A \texttt{x} => ... 

One way in C:

\begin{verbatim}
struct \texttt{t} {
  enum \{A, B, C\} \texttt{tag};
  union \{\texttt{t1 a; t2 b; t3 c;}\} \texttt{data};
};
... switch(\texttt{e->tag}) { case A: \texttt{t1 x=\texttt{e->data.a;}} ...
\end{verbatim}

▶ No static checking that tag is obeyed
▶ As fat as the fattest variant (avoidable with casts)
  ▶ Mutation costs us again!
Sums in Java

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

```
case e of 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 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 \( \text{int} \times (\text{int} \times (\text{int} \rightarrow \text{int})) \)

Pairs and sums are “logical duals” (as the typing rules show and we’ll see in a later lecture).

- To make a \( \tau_1 \times \tau_2 \), need a \( \tau_1 \) and a \( \tau_2 \).
- To make a \( \tau_1 + \tau_2 \), need a \( \tau_1 \) or a \( \tau_2 \).
- Given a \( \tau_1 \times \tau_2 \), can get a \( \tau_1 \) or a \( \tau_2 \) (your “choice”).
- Given a \( \tau_1 + \tau_2 \), 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 \((b_1, \cdots, b_n)\) and
▶ primitives \((p_1 : \tau_1, \cdots, p_n : \tau_n)\)

Examples:
▶ \text{concat} : \text{string} \rightarrow \text{string} \rightarrow \text{string}
▶ \text{floatToInt} : \text{float} \rightarrow \text{int}
▶ "hello" : \text{string}
Base Types and Primitives, in general

For each primitive, assume that if it is applied to values of the right types, then it produces a value of the right type.

> Not always a valid assumption: division? file I/O?

Together the types and assumed steps tell us how to type-check and evaluate $p_i \ v_1 \cdots v_n$ 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 recursion. But, our Lambda-Calculus encoding of recursion won’t type-check.

▶ Instead, add an explicit construct for recursion.
▶ Might consider `fun f x = e;`

Instead, introduce something more concise and general but less intuitive.

Note: No new values and no new types.
Recursion

We won’t prove it, but every extension so far preserves termination.

A Turing-complete language needs some sort of recursion. But, our Lambda-Calculus encoding of recursion won’t type-check.

Instead, add an explicit construct for recursion.

Instead, introduce something more concise and general but less intuitive.

\[
\begin{align*}
  e ::= & \cdots | \text{fix } e \\
  e \rightarrow_{\text{cbv}} e' \quad & \text{fix } e \rightarrow_{\text{cbv}} \text{fix } e' \\
  \text{fix } (\lambda x. e) \rightarrow_{\text{cbv}} e[\text{fix } (\lambda x. e)/x]
\end{align*}
\]

Note: No new values and no new types.
Using \texttt{fix}

It works just like \texttt{fun}, e.g.,

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

Note: Use \texttt{fix} and tuples to encode mutual recursion.
Why called \texttt{fix}?

In math, the 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, 42, or infinity fix-points.
- Examples for functions of type \texttt{int} \( \rightarrow \) \texttt{int}:
  - \( \lambda x. x + 1 \) has no fix-points
  - \( \lambda x. x * 0 \) has one fix-point
  - \( \lambda x. \text{abs\_val}(x) \) has an infinite number of fix-points
  - \( \lambda x. \text{if } x < 10 \text{ && } x > 0 \text{ then } x \text{ else } 0 \) 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)

▶ \(\lambda f. \lambda x. \text{abs\_val}(f \ x)\) has an infinite number of fix-points
  ▶ Any function that never returns a negative result
Back to factorial

So, what are the fix-points of

\[ \lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x \ast (f(x - 1)) \]

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

And \( \text{fix} (\lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x \ast (f(x - 1))) \)
behaves just like the factorial function.

\( \Rightarrow \) i.e., it behaves just like the fix-point of

\[ \lambda f. \lambda x. \text{if } x < 1 \text{ then } 1 \text{ else } x \ast (f(x - 1)) \].

(This isn’t really important, but good to explain terminology and show that programming is deeply connected to mathematics.)
Recursion type system and type soundness

Recall: fix-point only makes sense for functions of type $\tau \rightarrow \tau$ (for some $\tau$).

$$
\begin{align*}
\Gamma \vdash e : \tau \rightarrow \tau \\
\Gamma \vdash \text{fix } e : \tau
\end{align*}
$$

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: The $\tau$ in the typing rule is usually insantiated 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!
Recursion redux

Because `fix` most useful for defining functions:

\[
e ::= \cdots \mid \text{rec } f \ x. \ e
\]

\[
\text{rec } f \ x. \ e \rightarrow_{\text{cbv}} \lambda x. \ e[\text{rec } f \ x. \ e/f]
\]

\[
\Gamma, f : \tau_a \rightarrow \tau_r, x : \tau \vdash e : \tau_r \\
\Gamma \vdash \text{rec } f \ x. \ e : \tau_a \rightarrow \tau_r
\]

Equivalently, make it a derived form:

- `\text{rec } f \ x. \ e` “desugars to / macro expands to” `fix (\lambda f. \lambda x. \ e)`
Lists

\[
\begin{align*}
e & ::= \cdots \mid \text{nil} \mid e::e \mid \text{nil}?\ e \mid \text{hd}\ e \mid \text{tl}\ e \\
\nu & ::= \cdots \mid \text{nil} \mid \nu::\nu \\
\tau & ::= \cdots \mid \text{list}\ \tau
\end{align*}
\]

\[
\begin{align*}
e_h & \to_{\text{cbv}} e'_h \\
\frac{e_h :: e_t \to_{\text{cbv}} e'_h :: e_t}{e_h :: e_t \to_{\text{cbv}} e'_h :: e_t} & \quad \frac{e_t \to_{\text{cbv}} e'_t}{v_h :: e_t \to_{\text{cbv}} v_h :: e'_t}
\end{align*}
\]

\[
\begin{align*}
e_l & \to_{\text{cbv}} e'_l \\
\frac{\text{nil}\ ?\ e_l \to_{\text{cbv}} \text{nil}\ ?\ e'_l}{\text{nil}\ ?\ e_l \to_{\text{cbv}} \text{nil}\ ?\ e'_l} & \quad \frac{e_l \to_{\text{cbv}} e'_l}{\text{hd}\ e_l \to_{\text{cbv}} \text{hd}\ e'_l} & \quad \frac{e_l \to_{\text{cbv}} e'_l}{\text{tl}\ e_l \to_{\text{cbv}} \text{tl}\ e'_l}
\end{align*}
\]

\[
\begin{align*}
\frac{\text{nil}\ ?\ \text{nil} \to_{\text{cbv}} \text{true}}{\text{nil}\ ?\ \text{nil} \to_{\text{cbv}} \text{true}} & \quad \frac{\text{nil}\ ?\ (v_h :: v_t) \to_{\text{cbv}} \text{false}}{\text{nil}\ ?\ (v_h :: v_t) \to_{\text{cbv}} \text{false}}
\end{align*}
\]

\[
\begin{align*}
\frac{\text{hd}\ (v_h :: v_t) \to_{\text{cbv}} v_h}{\text{hd}\ (v_h :: v_t) \to_{\text{cbv}} v_h} & \quad \frac{\text{tl}\ (v_h :: v_t) \to_{\text{cbv}} v_t}{\text{tl}\ (v_h :: v_t) \to_{\text{cbv}} v_t}
\end{align*}
\]
Lists (continued)

\[
\begin{align*}
\Gamma \vdash \text{nil} : \text{list } \tau \\
\Gamma \vdash e_h : \tau & \quad \Gamma \vdash e_t : \text{list } \tau \\
\quad \quad \quad \Gamma \vdash e_h :: e_t : \text{list } \tau \\
\Gamma \vdash e_l : \text{list } \tau \\
\Gamma \vdash \text{nil? } e_l : \text{bool} \\
\Gamma \vdash \text{hd } e_l : \tau \\
\Gamma \vdash \text{tl } e_l : \text{list } \tau
\end{align*}
\]

Progress and Preservation: Straightforward
Lists (continued)

\[
\Gamma \vdash \text{nil} : \text{list } \tau \\
\Gamma \vdash e_h : \tau \\
\Gamma \vdash e_t : \text{list } \tau \\
\Gamma \vdash e_h :: e_t : \text{list } \tau \\
\Gamma \vdash e_l : \text{list } \tau \\
\Gamma \vdash \text{nil? } e_l : \text{bool} \\
\Gamma \vdash \text{hd } e_l : \tau \\
\Gamma \vdash \text{tl } e_l : \text{list } \tau
\]

Progress and Preservation: Straightforward

Oops: \text{hd nil} is well-typed, but stuck!
Lists (corrected)

\[
e ::= \cdots | \text{nil} | e :: e | \text{unlist } e
\]

\[
v ::= \cdots | \text{nil} | v :: v
\]

\[
\tau ::= \cdots | \text{list } \tau
\]

\[
\frac{e_h \rightarrow_{\text{cbv}} e'_h}{e_h :: e_t \rightarrow_{\text{cbv}} e'_h :: e_t}
\]

\[
\frac{e_t \rightarrow_{\text{cbv}} e'_t}{v_h :: e_t \rightarrow_{\text{cbv}} v_h :: e'_t}
\]

\[
\frac{e_l \rightarrow_{\text{cbv}} e'_l}{\text{unlist } e_l \rightarrow_{\text{cbv}} \text{unlist } e'_l}
\]

\[
\text{unlist } \text{nil} \rightarrow_{\text{cbv}} L(0)
\]

\[
\text{unlist } (v_h :: v_t) \rightarrow_{\text{cbv}} R((v_h, v_t))
\]

\[
\frac{\Gamma \vdash e_h : \tau \quad \Gamma \vdash e_t : \text{list } \tau}{\Gamma \vdash e_h :: e_t : \text{list } \tau}
\]

\[
\frac{\Gamma \vdash e_l : \text{list } \tau}{\Gamma \vdash \text{unlist } e_l : \text{int} + (\tau * \text{list } \tau)}
\]

Progress and Preservation: Straightforward
Extensions: the general approach

We added `let`, booleans, pairs, records, sums, `fix`, lists.

- `let` was syntactic sugar.
- `fix` “baked in self-application” (and made lang. Turing-complete).
- 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? for pairs? for sums? for lists?

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:
- C, C++, Java: all record types (or similar) have names
  - (omitting them just means compiler makes up a name)
- SML sum-types have names.
  - SML record-types are unnamed.

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 allow type-based code to distinguish names: good.

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

Surprising fact: If $\cdot \vdash e : \tau$ in the STLC with all additions except fix, then there exists a $v$ such that $e \rightarrow^* v$.

That is, all programs terminate.

Termination is trivially decidable (the constant “yes” function), and language is not Turing-complete.

Proof is in the book. 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 $\Gamma \vdash x \ x : \tau$. 