Wrapping up Polymorphism
We saw that in System F has explicit type abstraction and application:

\[ \Theta, \alpha; \Gamma \vdash e : B \]

\[ \Theta; \Gamma \vdash \lambda \alpha. e : \forall \alpha. B \]

\[ \Theta; \Gamma \vdash e : \forall \alpha. B \]

\[ \Theta \vdash A \text{ type} \]

\[ \Theta; \Gamma \vdash e A : [A/\alpha]B \]

This is fine in theory, but what do programs look like in practice?
Suppose we have a map functional and an isEven function:

\[
\begin{align*}
map &: \forall \alpha. \forall \beta. (\alpha \to \beta) \to \text{list } \alpha \to \text{list } \beta \\
isEven &: \mathbb{N} \to \text{bool}
\end{align*}
\]

A function taking a list of numbers and applying isEven to it:

\[
\begin{align*}
\text{map } \mathbb{N} \text{ bool } \text{isEven} &: \text{list } \mathbb{N} \to \text{list bool}
\end{align*}
\]

If you have a list of lists of natural numbers:

\[
\begin{align*}
\text{map } (\text{list } \mathbb{N}) (\text{list bool}) (\text{map } \mathbb{N} \text{ bool } \text{isEven}) &: \text{list } (\text{list } \mathbb{N}) \to \text{list } (\text{list bool})
\end{align*}
\]

The type arguments overwhelm everything else!
• Luckily, ML and Haskell have type inference
• Explicit type applications are omitted – we write
  `map isEven` instead of `map \mathbb{N} \text{bool} isEven` 
• Constraint propagation via the *unification algorithm*
  figures out what the applications should have been

Example:

\[
\begin{align*}
\text{map } ?a \ ?b \ & \text{ isEven} \\
\text{map } ?a \ ?b & : (\mathbb{N} \rightarrow ?b) \rightarrow \text{list } ?a \rightarrow \text{list } ?b \\
\text{isEven} : \mathbb{N} \rightarrow \text{bool} & \quad \text{So } ?a \rightarrow ?b \text{ must equal } \mathbb{N} \rightarrow \text{bool} \\
?a = \mathbb{N}, \ ?b = \text{bool} & \quad \text{Only choice that makes } ?a \rightarrow ?b = \mathbb{N} \rightarrow \text{bool}
\end{align*}
\]
Effects
The Story so Far...

- We introduced the simply-typed lambda calculus
- ...and its double life as constructive propositional logic
- We extended it to the polymorphic lambda calculus
- ...and its double life as second-order logic

This is a story of pure, total functional programming
Effects

• Sometimes, we write programs that takes an input and computes an answer:
  • Physics simulations
  • Compiling programs
  • Ray-tracing software

• Other times, we write programs to do things:
  • communicate with the world via I/O and networking
  • update and modify physical state (eg, file systems)
  • build interactive systems like GUIs
  • control physical systems (eg, robots)
  • generate random numbers

• PL jargon: pure vs effectful code
Two Paradigms of Effects

- From the POV of type theory, two main classes of effects:
  1. State:
     - Mutable data structures (hash tables, arrays)
     - References/pointers
  2. Control:
     - Exceptions
     - Coroutines/generators
     - Nondeterminism

- Other effects (eg, I/O and concurrency/multithreading) can be modelled in terms of state and control effects

- In this lecture, we will focus on state and how to model it
State

```ocaml
# let r = ref 5;;
val r : int ref = {contents = 5}
# !r;;
- : int = 0
# r := !r + 15;;
- : unit = ()
# !r;;
- : int = 20

- We can create fresh reference with ref e
- We can read a reference with !e
- We can update a reference with e := e'
```
### A Type System for State

<table>
<thead>
<tr>
<th>Category</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td>( X ::= 1 \mid \mathbb{N} \mid X \rightarrow Y \mid \text{ref} X )</td>
</tr>
<tr>
<td><strong>Terms</strong></td>
<td>( e ::= \langle \rangle \mid n \mid \lambda x : X. e \mid ee' )</td>
</tr>
<tr>
<td></td>
<td>( \mid \text{new} e \mid !e \mid e ::= e' \mid l )</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>( v ::= \langle \rangle \mid n \mid \lambda x : X. e \mid l )</td>
</tr>
<tr>
<td><strong>Stores</strong></td>
<td>( \sigma ::= \cdot \mid \sigma, l : v )</td>
</tr>
<tr>
<td><strong>Contexts</strong></td>
<td>( \Gamma ::= \cdot \mid \Gamma, x : X )</td>
</tr>
<tr>
<td><strong>Store Typings</strong></td>
<td>( \Sigma ::= \cdot \mid \Sigma, l : X )</td>
</tr>
</tbody>
</table>
Operational Semantics

\[
\begin{align*}
\langle \sigma; e_0 \rangle & \leadsto \langle \sigma'; e_0' \rangle \\
\langle \sigma; e_0 e_1 \rangle & \leadsto \langle \sigma'; e_0' e_1 \rangle \\
\langle \sigma; e_1 \rangle & \leadsto \langle \sigma'; e_1' \rangle \\
\langle \sigma; v_0 e_1 \rangle & \leadsto \langle \sigma'; v_0 e_1' \rangle
\end{align*}
\]

\[
\langle \sigma; (\lambda x : X. e) \rangle \leadsto \langle \sigma; [v/x] e \rangle
\]

- Similar to the basic STLC operational rules
- Threads a store \( \sigma \) through each transition
Operational Semantics

\[
\begin{align*}
\langle \sigma; e \rangle & \leadsto \langle \sigma'; e' \rangle \\
\langle \sigma; \text{new } e \rangle & \leadsto \langle \sigma'; \text{new } e' \rangle \\
\langle \sigma; e \rangle & \leadsto \langle \sigma'; e' \rangle \\
\langle \sigma; \text{!}e \rangle & \leadsto \langle \sigma'; \text{!}e' \rangle \\
\langle \sigma; e_0 \rangle & \leadsto \langle \sigma'; e'_0 \rangle \\
\langle \sigma; e_0 := e_1 \rangle & \leadsto \langle \sigma'; e'_0 := e_1 \rangle \\
\langle \sigma; l \notin \text{dom}(\sigma) \rangle & \leadsto \langle (\sigma, l : v); l \rangle \\
\langle \sigma; \text{new } v \rangle & \leadsto \langle (\sigma, l : v); l \rangle \\
\langle \sigma; l : v \in \sigma \rangle & \leadsto \langle \sigma; l \rangle \\
\langle \sigma; \text{!}l \rangle & \leadsto \langle \sigma; v \rangle \\
\langle \sigma; e_1 \rangle & \leadsto \langle \sigma'; e'_1 \rangle \\
\langle \sigma; v_0 := e_1 \rangle & \leadsto \langle \sigma'; v_0 := e'_1 \rangle \\
\langle (\sigma, l : v, \sigma'); l := v' \rangle & \leadsto \langle (\sigma, l : v', \sigma'); \langle \rangle \rangle
\end{align*}
\]
Typing for Terms

\[ \Sigma; \Gamma \vdash e : X \]

\[ \begin{array}{c}
\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X} & \text{HYP} & \frac{1 : 1}{\Sigma; \Gamma \vdash \langle \rangle : 1} & \frac{n : \mathbb{N}}{\Sigma; \Gamma \vdash n : \mathbb{N}}
\end{array} \]

\[ \Sigma; \Gamma, x : X \vdash e : Y \]

\[ \frac{\Sigma; \Gamma \vdash \lambda x : X. e : X \rightarrow Y}{\Sigma; \Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow \text{l} \]

\[ \begin{array}{c}
\frac{\Sigma; \Gamma \vdash e : X \rightarrow Y}{\Sigma; \Gamma \vdash e : X \rightarrow Y} & \frac{\Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e' : X} & \frac{\Sigma; \Gamma \vdash ee' : Y}{\Sigma; \Gamma \vdash ee' : Y} \rightarrow \text{E}
\end{array} \]

- Similar to STLC rules + thread \( \Sigma \) through all judgements
Typing for Imperative Terms

\[ \Sigma; \Gamma \vdash e : X \]

\[
\begin{align*}
\Sigma; \Gamma \vdash e : X & \quad \text{REFL} \quad \Sigma; \Gamma \vdash \text{new } e : \text{ref } X \\
\Sigma; \Gamma \vdash e : \text{ref } X & \quad \Sigma; \Gamma \vdash e' : X \quad \text{REFSET} \\
\Sigma; \Gamma \vdash e := e' : 1
\end{align*}
\]

\[
\begin{align*}
l : X \in \Sigma \quad \text{REFBAR} \\
\Sigma; \Gamma \vdash l : \text{ref } X
\end{align*}
\]

- Usual rules for references
- But why do we have the bare reference rule?
• Original progress and preservations talked about well-typed terms $e$ and evaluation steps $e \leadsto e'$

• New operational semantics $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$ mentions stores, too.

• To prove type safety, we will need a notion of store typing
Store and Configuration Typing

\[
\Sigma \vdash \sigma' : \Sigma' \quad \langle \sigma; e \rangle : \langle \Sigma; X \rangle
\]

\[
\frac{\Sigma \vdash \sigma' : \Sigma' \quad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)} \quad \text{STORECONS}
\]

\[
\frac{\Sigma \vdash \sigma : \Sigma \quad \Sigma; \cdot \vdash e : X}{\langle \sigma; e \rangle : \langle \Sigma; X \rangle} \quad \text{CONFIGOK}
\]

- Check that all the closed values in the store \( \sigma' \) are well-typed
- Types come from \( \Sigma' \), checked in store \( \Sigma \)
- Configurations are well-typed if the store and term are well-typed
A Broken Theorem

Progress:
If \(\langle \sigma; e \rangle: \langle \Sigma; X \rangle\) then \(e\) is a value or \(\langle \sigma; e \rangle \sim \langle \sigma'; e' \rangle\).

Preservation:
If \(\langle \sigma; e \rangle: \langle \Sigma; X \rangle\) and \(\langle \sigma; e \rangle \sim \langle \sigma'; e' \rangle\) then \(\langle \sigma'; e' \rangle: \langle \Sigma; X \rangle\).

• One of these theorems is false!
Note that

1. $\langle \cdot; \text{new} \langle \rangle \rangle : \langle \cdot; \text{ref} \ 1 \rangle$
2. $\langle \cdot; \text{new} \langle \rangle \rangle \sim \langle (l : \langle \rangle); l \rangle$ for some $l$

However, it is not the case that

$\langle l : \langle \rangle; l \rangle : \langle \cdot; \text{ref} \ 1 \rangle$

The heap has grown!
Definition (Store extension):

Define $\Sigma \leq \Sigma'$ to mean there is a $\Sigma''$ such that $\Sigma' = \Sigma, \Sigma''$.

Lemma (Store Monotonicity):

If $\Sigma \leq \Sigma'$ then:

1. If $\Sigma; \Gamma \vdash e : X$ then $\Sigma'; \Gamma \vdash e : X$.
2. If $\Sigma \vdash \sigma_0 : \Sigma_0$ then $\Sigma' \vdash \sigma_0 : \Sigma_0$.

The proof is by structural induction on the appropriate definition.

This property means allocating new references never breaks the typability of a term.
• (Weakening)
  If $\Sigma; \Gamma, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash e : X$.

• (Exchange)
  If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash e : X$.

• (Substitution)
  If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash e' : Z$ then $\Sigma; \Gamma \vdash [e/x]e' : Z$. 
Type Safety, Repaired

Theorem (Progress):
If \( \langle \sigma; e \rangle : \langle \Sigma; X \rangle \) then \( e \) is a value or \( \langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle \).

Theorem (Preservation):
If \( \langle \sigma; e \rangle : \langle \Sigma; X \rangle \) and \( \langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle \) then there exists \( \Sigma' \geq \Sigma \) such that \( \langle \sigma'; e' \rangle : \langle \Sigma'; X \rangle \).

Proof:

- For progress, induction on derivation of \( \Sigma; \cdot \vdash e : X \)
- For preservation, induction on derivation of \( \langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle \)
• Suppose we have an unknown function in the STLC:

\[ f : ((1 \to 1) \to 1) \to \mathbb{N} \]

• Q: What can this function do?
• A: It is a constant function, returning some \( n \)

• Q: Why?
• A: No matter what \( f(g) \) does with its argument \( g \), it can only get \( \langle \rangle \) out of it. So the argument can never influence the value of type \( \mathbb{N} \) that \( f \) produces.
The Power of the State

\[
\text{count} : ((1 \to 1) \to 1) \to \mathbb{N}
\]

\[
\text{count } f = \text{let } r : \text{ref} \mathbb{N} = \text{new} 0 \text{ in}
\]

\[
\text{let } inc : 1 \to 1 = \lambda z : 1. r := !r + 1 \text{ in}
\]

\[
f(inc)
\]

- This function initializes a counter \( r \)
- It creates a function \( inc \) which silently increments \( r \)
- It passes \( inc \) to its argument \( f \)
- Then it returns the value of the counter \( r \)
- That is, it returns the number of times \( inc \) was called!
Backpatching with Landin’s Knot

let knot : ((int -> int) -> int -> int) -> int -> int = 
  fun f ->
    let r = ref (fun n -> 0) in
    let recur = fun n -> !r n in
    let () = r := fun n -> f recur n in
    recur

1. Create a reference holding a function
2. Define a function that forwards its argument to the ref
3. Set the reference to a function that calls $f$ on the forwarder and the argument $n$
4. Now $f$ will call itself recursively!
Not a Theorem: (Termination) Every well-typed program 
\( \vdash e : X \) terminates.

- Landin’s knot lets us define recursive functions by backpatching
- As a result, we can write nonterminating programs
- So every type is inhabited, and consistency fails
Consistency vs Computation

- Do we have to choose between state/effects and logical consistency?
- Is there a way to get the best of both?
- Alternately, is there a Curry-Howard interpretation for effects?
- Next lecture:
  - A modal logic suggested by Curry in 1952
  - Now known to functional programmers as *monads*
  - Also known as *effect systems*
Questions

1. Using Landin’s knot, implement the fibonacci function.
2. The type safety proof for state would fail if we added a C-like `free()` operation to the reference API.
   2.1 Give a plausible-looking typing rule and operational semantics for `free`.
   2.2 Find an example of a program that would break.