Type Systems
Lecture 7: Programming with Effects

Neel Krishnaswami
University of Cambridge
Wrapping up Polymorphism
System F is Explicit

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

\[
\begin{align*}
\Theta, \alpha; \Gamma \vdash e : B & \quad \Theta; \Gamma \vdash \forall \alpha. e : \forall \alpha. B \\
\Theta; \Gamma \vdash \land \alpha. e : \forall \alpha. B & \quad \Theta; \Gamma \vdash e A : [A/\alpha]B
\end{align*}
\]

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

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

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

\[
\text{map \mathbb{N} bool isEven} : \text{list} \mathbb{N} \to \text{list bool}
\]

If you have a list of lists of natural numbers:

\[
\text{map (list \mathbb{N}) (list bool) (map \mathbb{N} bool isEven)} : \text{list (list \mathbb{N}) \to list (list bool)}
\]

The type arguments overwhelm everything else!
Type Inference

• Luckily, ML and Haskell have type inference
• Explicit type applications are omitted – we write \( \text{map isEven} \) instead of \( \text{map } \mathbb{N} \text{ bool isEven} \)
• Constraint propagation via the unification algorithm figures out what the applications should have been

Example:

\[
\begin{align*}
\text{map } ?a & \ ?b \text{ isEven} & \text{Introduce placeholders } ?a \text{ and } ?b \\
\text{map } ?a & \ ?b & : (?a \rightarrow ?b) \rightarrow \text{list } ?a \rightarrow \text{list } ?b \\
\text{isEven} & : \mathbb{N} \rightarrow \text{bool} & \text{So } ?a \rightarrow ?b \text{ must equal } \mathbb{N} \rightarrow \text{bool} \\
?a = \mathbb{N}, \ ?b = \text{bool} & \text{Only choice that makes } ?a \rightarrow ?b = \mathbb{N} \rightarrow \text{bool}
\end{align*}
\]
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
• 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 (e.g., file systems)
  • build interactive systems like GUIs
  • control physical systems (e.g., 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 (e.g., 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.
# 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

Types
\[ X ::= 1 \mid \mathbb{N} \mid X \rightarrow Y \mid \text{ref}X \]

Terms
\[ e ::= \langle \rangle \mid n \mid \lambda x : X. e \mid ee' \]
\[ \mid \text{new} e \mid !e \mid e := e' \mid l \]

Values
\[ v ::= \langle \rangle \mid n \mid \lambda x : X. e \mid l \]

Stores
\[ \sigma ::= \cdot \mid \sigma, l : v \]

Contexts
\[ \Gamma ::= \cdot \mid \Gamma, x : X \]

Store Typings
\[ \Sigma ::= \cdot \mid \Sigma, l : X \]
Operational Semantics

\[
\begin{align*}
\langle \sigma; e_0 \rangle & \rightsquigarrow \langle \sigma'; e'_0 \rangle \\
\langle \sigma; e_0 e_1 \rangle & \rightsquigarrow \langle \sigma'; e'_0 e_1 \rangle \\
\langle \sigma; e_1 \rangle & \rightsquigarrow \langle \sigma'; e'_1 \rangle \\
\langle \sigma; v e_1 \rangle & \rightsquigarrow \langle \sigma'; v e'_1 \rangle \\
\langle \sigma; (\lambda x : X. e) v \rangle & \rightsquigarrow \langle \sigma; [v/x] e \rangle
\end{align*}
\]

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

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

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

\[
\begin{align*}
    x : X & \in \Gamma \\
    & \frac{}{\Sigma; \Gamma \vdash x : X} \text{ HYP} \\
    \langle \rangle & \vdash 1 \\
    & \frac{}{\Sigma; \Gamma \vdash \langle \rangle : 1} \text{ 1I} \\
    n & \vdash n : \mathbb{N} \\
    & \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}} \text{ \mathbb{N}I} \\
    \Sigma; \Gamma, x : X & \vdash e : Y \\
    & \frac{}{\Sigma; \Gamma \vdash \lambda x : X. e : X \rightarrow Y} \text{ \rightarrow I} \\
    \Sigma; \Gamma & \vdash e : X \rightarrow Y \\
    \Sigma; \Gamma & \vdash e' : X \\
    & \frac{}{\Sigma; \Gamma \vdash e \, e' : Y} \text{ \rightarrow E} \\
\end{align*}
\]

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

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

\[ \Sigma ; \Gamma \vdash e : X \quad \text{REFL} \]
\[ \Sigma ; \Gamma \vdash \text{new} \; e : \text{ref} \; X \]

\[ \Sigma ; \Gamma \vdash e : \text{ref} \; X \]
\[ \Sigma ; \Gamma \vdash !e : X \quad \text{REFGET} \]

\[ \Sigma ; \Gamma \vdash e : \text{ref} \; X \quad \Sigma ; \Gamma \vdash e' : X \]
\[ \Sigma ; \Gamma \vdash e := e' : 1 \quad \text{REFSET} \]

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

- Usual rules for references
- But why do we have the bare reference rule?
Proving Type Safety

- 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

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

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

- 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 $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$.

Preservation:
If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$ then $\langle \sigma'; e' \rangle : \langle \Sigma; X \rangle$.

• One of these theorems is false!
The Counterexample to Preservation

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.
Substitution and Structural Properties

- (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 $\Sigma; \Gamma \vdash [e/x]e' : Z$. 
Theorem (Progress):
If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ then $e$ is a value or $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$.

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

Proof:

- For progress, induction on derivation of $\Sigma; \vdash e : X$
- For preservation, induction on derivation of $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$
A Curious Higher-order Function

• Suppose we have an unknown function in the STLC:

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

• 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 gets \( \langle \rangle \) out of it. So the argument can never influence the value of type \( \mathbb{N} \) that \( f \) produces.
The Power of the State

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

\[
\text{count } f \quad = \quad \text{let } r : \text{ref } \mathbb{N} = \text{new 0 in}
\]
\[
\text{let } \text{inc} : 1 \to 1 = \lambda z : 1. r := !r + 1 \text{ in}
\]
\[
f(\text{inc})
\]

- This function initializes a counter \( r \)
- It creates a function \( \text{inc} \) which silently increments \( r \)
- It passes \( \text{inc} \) to its argument \( f \)
- Then it returns the value of the counter \( r \)
- That is, it returns the number of times \( \text{inc} \) was called!
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!
Another False Theorem

Not a Theorem: (Termination) Every well-typed program
\( \cdot \vdash e : X \) 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
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.