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:

$$\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, B} \qquad \frac{\Theta; \Gamma \vdash e : \forall\alpha.\, B \qquad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash e\, A : [A/\alpha]B}$$

This is fine in theory, but what do programs look like in practice?

System F is Very, Very Explicit

Suppose we have a map functional and an isEven function:

$$\begin{aligned} \mathit{map} &: \forall \alpha. \forall \beta. (\alpha \to \beta) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\beta \\ \mathit{isEven} &: \mathbb{N} \to \mathsf{bool} \end{aligned}$$

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

$$\mathit{map}\,\mathbb{N}\,\mathsf{bool}\,\mathit{isEven} \quad : \quad \mathsf{list}\,\mathbb{N} \to \mathsf{list}\,\mathsf{bool}$$

If you have a list of lists of natural numbers:

$$\begin{gathered} \mathit{map}\,(\mathsf{list}\,\mathbb{N})\,(\mathsf{list}\,\mathsf{bool})\,(\mathit{map}\,\mathbb{N}\,\mathsf{bool}\,\mathit{isEven}) \\ : \mathsf{list}\,(\mathsf{list}\,\mathbb{N}) \to \mathsf{list}\,(\mathsf{list}\,\mathsf{bool}) \end{gathered}$$

The type arguments overwhelm everything else!

Type Inference

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

Example:

map *isEven*	Term that needs type inference
map ?*a* ?*b* *isEven*	Introduce placeholders ?*a* and ?*b*
map ?*a* ?*b*	$: (?a \to ?b) \to \text{list}\ ?a \to \text{list}\ ?b$
isEven $: \mathbb{N} \to$ bool	So $?a \to ?b$ must equal $\mathbb{N} \to$ bool
$?a = \mathbb{N}, ?b =$ bool	Only choice that makes $?a \to ?b = \mathbb{N} \to$ bool

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

```
# let r = ref 5;;
val r : int ref = {contents = 5}

# !r;;
- : int = 5

# 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

$$
\begin{array}{llcl}
\text{Types} & X & ::= & 1 \mid \mathbb{N} \mid X \to Y \mid \mathsf{ref}\, X \\
\text{Terms} & e & ::= & \langle\rangle \mid n \mid \lambda x : X.\, e \mid e\, e' \\
 & & \mid & \mathsf{new}\, e \mid \,!e \mid e := e' \mid l \\
\\
\text{Values} & v & ::= & \langle\rangle \mid n \mid \lambda x : X.\, e \mid l \\
\text{Stores} & \sigma & ::= & \cdot \mid \sigma, l : v \\
\\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X \\
\text{Store Typings} & \Sigma & ::= & \cdot \mid \Sigma, l : X
\end{array}
$$

Operational Semantics

$$\frac{\langle \sigma; e_0 \rangle \rightsquigarrow \langle \sigma'; e_0' \rangle}{\langle \sigma; e_0\, e_1 \rangle \rightsquigarrow \langle \sigma'; e_0'\, e_1 \rangle} \qquad \frac{\langle \sigma; e_1 \rangle \rightsquigarrow \langle \sigma'; e_1' \rangle}{\langle \sigma; v_0\, e_1 \rangle \rightsquigarrow \langle \sigma'; v_0\, e_1' \rangle}$$

$$\frac{}{\langle \sigma; (\lambda x : X.\, e)\, v \rangle \rightsquigarrow \langle \sigma; [v/x]e \rangle}$$

- Similar to the basic STLC operational rules
- Threads a store σ through each transition

Operational Semantics

$$\frac{\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle}{\langle \sigma; \mathsf{new}\, e \rangle \leadsto \langle \sigma'; \mathsf{new}\, e' \rangle} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{\langle \sigma; \mathsf{new}\, v \rangle \leadsto \langle (\sigma, l : v); l \rangle}$$

$$\frac{\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle}{\langle \sigma; !e \rangle \leadsto \langle \sigma'; !e' \rangle} \qquad \frac{l : v \in \sigma}{\langle \sigma; !l \rangle \leadsto \langle \sigma; v \rangle}$$

$$\frac{\langle \sigma; e_0 \rangle \leadsto \langle \sigma'; e_0' \rangle}{\langle \sigma; e_0 := e_1 \rangle \leadsto \langle \sigma'; e_0' := e_1 \rangle} \qquad \frac{\langle \sigma; e_1 \rangle \leadsto \langle \sigma'; e_1' \rangle}{\langle \sigma; v_0 := e_1 \rangle \leadsto \langle \sigma'; v_0 := e_1' \rangle}$$

$$\frac{}{\langle (\sigma, l : v, \sigma'); l := v' \rangle \leadsto \langle (\sigma, l : v', \sigma'); \langle\rangle \rangle}$$

Typing for Terms

$$\boxed{\Sigma; \Gamma \vdash e : X}$$

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X}\ \textsc{Hyp} \qquad \frac{}{\Sigma; \Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I} \qquad \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\mathrm{I}$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I} \qquad \frac{\Sigma; \Gamma \vdash e : X \to Y \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

- Similar to STLC rules + thread Σ through all judgements

Typing for Imperative Terms

$$\boxed{\Sigma; \Gamma \vdash e : X}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \mathsf{new}\, e : \mathsf{ref}\, X}\ \textsc{RefI} \qquad \frac{\Sigma; \Gamma \vdash e : \mathsf{ref}\, X}{\Sigma; \Gamma \vdash\ !e : X}\ \textsc{RefGet}$$

$$\frac{\Sigma; \Gamma \vdash e : \mathsf{ref}\, X \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' : 1}\ \textsc{RefSet} \qquad \frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \mathsf{ref}\, X}\ \textsc{RefBar}$$

- 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

$$\boxed{\Sigma \vdash \sigma' : \Sigma'} \qquad \boxed{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}$$

$$\frac{}{\Sigma \vdash \cdot : \cdot}\ \textsc{StoreNil} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \qquad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)}\ \textsc{StoreCons}$$

$$\frac{\Sigma \vdash \sigma : \Sigma \qquad \Sigma; \cdot \vdash e : X}{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}\ \textsc{ConfigOK}$$

- Check that all the closed values in the store σ' are well-typed
- Types come from Σ', checked in store Σ
- 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. $\langle \cdot; \text{new}\,\langle\rangle\rangle \leadsto \langle (l : \langle\rangle); l\rangle$ 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!

Store Monotonicity

Definition (Store extension):

Define $\Sigma \leq \Sigma'$ to mean there is a Σ'' 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 $\Sigma; \Gamma \vdash [e/x]e' : Z$.

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

A Curious Higher-order Function

- 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 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

$$
\begin{aligned}
\mathit{count} &: ((1 \to 1) \to 1) \to \mathbb{N} \\
\mathit{count}\ f &= \text{let } r : \text{ref}\,\mathbb{N} = \text{new}\,0 \text{ in} \\
&\quad\ \text{let } \mathit{inc} : \mathit{1} \to \mathit{1} = \lambda z : 1.\, r := !r + 1 \text{ in} \\
&\quad\ f(\mathit{inc});\, !r
\end{aligned}
$$

- 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!

Another False Theorem

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

- Landin's knot lets us *define recursive functions* by backpatching
- As a result, we can write nonterminating programs!

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.