Type Systems

Lecture 8: Using Monads to Control Effects

Neel Krishnaswami
University of Cambridge

Last Lecture

```
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

What is the Problem?

1. We began with the typed lambda calculus
2. We added state as a set of primitive operations
3. We lost termination
4. Problem: unforseen interaction between different parts of the language
    - Recursive definitions = state + functions
5. Question: is this a real problem?

What is the Solution?

- Restrict the use of state:
  1. Limit what references can store (eg, only to booleans and integers)
  2. Restrict how references can be referred to (eg, in core safe Rust)
  3. We don't have time to pursue these in this course
- Mark the use of state:
  - Distinguish between *pure* and *impure* code
  - Impure computations can depend on pure ones
  - Pure computations cannot depend upon impure ones
  - A form of taint tracking

Monads for State

Types	X	$::=$	$1 \mid \mathbb{N} \mid X \to Y \mid \text{ref}\, X \mid \text{T}\, X$
Pure Terms	e	$::=$	$\langle\rangle \mid n \mid \lambda x : X.\, e \mid e\, e' \mid l \mid \{t\}$
Impure Terms	t	$::=$	$\text{new}\, e \mid\ !e \mid e := e'$
		$\mid$	$\text{let}\ x = e;\ t \mid \text{return}\, e$
Values	v	$::=$	$\langle\rangle \mid n \mid \lambda x : X.\, e \mid l \mid \{t\}$
Stores	σ	$::=$	$\cdot \mid \sigma, l : v$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : X$
Store Typings	Σ	$::=$	$\cdot \mid \Sigma, l : X$

Typing for Pure Terms

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

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

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

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \mathrm{ref}\, X}\ \textsc{RefBar} \qquad \frac{\Sigma; \Gamma \vdash t \div X}{\Sigma; \Gamma \vdash \{t\} : \mathrm{T}\, X}\ \mathrm{TI}$$

- Similar to STLC rules + thread Σ through all judgements
- New judgement $\Sigma; \Gamma \vdash t \div X$ for imperative computations

Typing for Effectful Terms

$$\boxed{\Sigma; \Gamma \vdash t \div X}$$

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

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

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \mathsf{return}\, e \div X}\ \textsc{TRet} \qquad \frac{\Sigma; \Gamma \vdash e : \mathsf{T}\, X \qquad \Sigma; \Gamma, x : X \vdash t \div Z}{\Sigma; \Gamma \vdash \mathsf{let}\, x = e;\ t \div Z}\ \textsc{TLet}$$

- We now mark potentially effectful terms in the judgement
- Note that return e isn't effectful – conservative approximation!

A Two-Level Operational Semantics: Pure Part

$$\frac{e_0 \leadsto e_0'}{e_0\, e_1 \leadsto e_0'\, e_1} \qquad \frac{e_1 \leadsto e_1'}{v_0\, e_1 \leadsto v_0\, e_1'} \qquad \frac{}{(\lambda x : X.\, e)\, v \leadsto [v/x]e}$$

- Similar to the basic STLC operational rules
- We no longer thread a store σ through each transition!

A Two-Level Operational Semantics: Impure Part, 1/2

$$\frac{e \leadsto e'}{\langle \sigma; \mathsf{new}\, e\rangle \leadsto \left\langle \sigma; \mathsf{new}\, e' \right\rangle}$$

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

$$\frac{e \leadsto e'}{\langle \sigma; !e\rangle \leadsto \left\langle \sigma; !e' \right\rangle}$$

$$\frac{l : v \in \sigma}{\langle \sigma; !l\rangle \leadsto \langle \sigma; \mathsf{return}\, v\rangle}$$

$$\frac{e_0 \leadsto e_0'}{\langle \sigma; e_0 := e_1\rangle \leadsto \left\langle \sigma; e_0' := e_1 \right\rangle}$$

$$\frac{e_1 \leadsto e_1'}{\langle \sigma; v_0 := e_1\rangle \leadsto \left\langle \sigma; v_0 := e_1' \right\rangle}$$

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

A Two-Level Operational Semantics: Impure Part, 2/2

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \mathrm{return}\, e \rangle \rightsquigarrow \langle \sigma; \mathrm{return}\, e' \rangle} \qquad \frac{e \rightsquigarrow e'}{\langle \sigma; \mathrm{let}\, x = e;\ t \rangle \rightsquigarrow \langle \sigma; \mathrm{let}\, x = e';\ t \rangle}$$

$$\frac{}{\langle \sigma; \mathrm{let}\, x = \{\mathrm{return}\, v\};\ t_1 \rangle \rightsquigarrow \langle \sigma; [v/x]t_1 \rangle}$$

$$\frac{\langle \sigma; t_0 \rangle \rightsquigarrow \langle \sigma'; t_0' \rangle}{\langle \sigma; \mathrm{let}\, x = \{t_0\};\ t_1 \rangle \rightsquigarrow \langle \sigma'; \mathrm{let}\, x = \{t_0'\};\ t_1 \rangle}$$

Store and Configuration Typing

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

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

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

- 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

Substitution and Structural Properties, 1/2

- **Pure Term Weakening**:
  If $\Sigma; \Gamma, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash e : X$.
- **Pure Term Exchange**:
  If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash e : X$.
- **Pure Term Substitution**:
  If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash e' : Z$ then $\Sigma; \Gamma \vdash [e/x]e' : Z$.

Substitution and Structural Properties, 2/2

- **Effectful Term Weakening**:
  If $\Sigma; \Gamma, \Gamma' \vdash t \div X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash t \div X$.
- **Effectful Term Exchange**:
  If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash t \div X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash t \div X$.
- **Effectful Term Substitution**:
  If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash t \div Z$ then $\Sigma; \Gamma \vdash [e/x]t \div Z$.

Proof Order

1. Prove Pure Term Weakening and Impure Term Weakening mutually inductively
2. Prove Pure Term Exchange and Impure Term Exchange mutually inductively
3. Prove Pure Term Substitution and Impure Term Substitution mutually inductively

Two mutually-recursive judgements $\Longrightarrow$ Two mutually-inductive proofs

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 $\Sigma; \Gamma \vdash t \div X$ then $\Sigma'; \Gamma \vdash t \div X$.
3. If $\Sigma \vdash \sigma_0 : \Sigma_0$ then $\Sigma' \vdash \sigma_0 : \Sigma_0$.

The proof is by structural induction on the appropriate definition. (Prove 1. and 2. mutually-inductively!)

This property means allocating new references never breaks the typability of a term.

Theorem (Pure Progress):

If $\Sigma; \cdot \vdash e : X$ then $e = v$ or $e \leadsto e'$.

Theorem (Pure Preservation):

If $\Sigma; \cdot \vdash e : X$ and $e \leadsto e'$ then $\Sigma; \cdot \vdash e' : X$.

Proof:

- For progress, induction on derivation of $\Sigma; \cdot \vdash e : X$
- For preservation, induction on derivation of $e \leadsto e'$

Type Safety for the Monadic Language

Theorem (Progress):

If $\langle \sigma; t \rangle : \langle \Sigma; X \rangle$ then $t = \text{return}\, v$ or $\langle \sigma; t \rangle \rightsquigarrow \langle \sigma'; t' \rangle$.

Theorem (Preservation):

If $\langle \sigma; t \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; t \rangle \rightsquigarrow \langle \sigma'; t' \rangle$ then there exists $\Sigma' \geq \Sigma$ such that $\langle \sigma'; t' \rangle : \langle \Sigma'; X \rangle$.

Proof:

- For progress, induction on derivation of $\Sigma; \cdot \vdash t \div X$
- For preservation, induction on derivation of $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$

What Have we Accomplished?

- In the monadic language, pure and effectful code is strictly separated
- As a result, *pure programs terminate*
- However, *we can still write imperative programs*

Types	X	$::=$	$1 \mid \mathbb{N} \mid X \to Y \mid \mathsf{T}_{\mathsf{IO}}\, X$
Pure Terms	e	$::=$	$\langle\rangle \mid n \mid \lambda x : X.\, e \mid e\, e' \mid \{t\}$
Impure Terms	t	$::=$	$\mathsf{print}\, e \mid \mathsf{let}\, x = e;\ t \mid \mathsf{return}\, e$
Values	v	$::=$	$\langle\rangle \mid n \mid \lambda x : X.\, e \mid \{t\}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : X$

Monads for I/O: Typing Pure Terms

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

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

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

$$\frac{\Gamma \vdash t \div X}{\Gamma \vdash \{t\} : \mathsf{T}\, X}\ \mathsf{T}\mathrm{I}$$

- Similar to STLC rules (no store typing!)
- New judgement $\Gamma \vdash t \div X$ for imperative computations

Typing for Effectful Terms

$$\boxed{\Gamma \vdash t \div X}$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{print}\, e \div 1}\ \textsc{TPrint}$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{return}\, e \div X}\ \textsc{TRet} \qquad \frac{\Gamma \vdash e : \mathsf{T}\,X \qquad \Gamma, x : X \vdash t \div Z}{\Gamma \vdash \mathsf{let}\, x = e;\ t \div Z}\ \textsc{TLet}$$

- TRet and TLet are identical rules
- Difference is in the operations – print e vs get/set/new

Operational Semantics for I/O: Pure Part

$$\frac{e_0 \leadsto e_0'}{e_0\, e_1 \leadsto e_0'\, e_1} \qquad \frac{e_1 \leadsto e_1'}{v_0\, e_1 \leadsto v_0\, e_1'} \qquad \frac{}{(\lambda x : X.\, e)\, v \leadsto [v/x]e}$$

- *Identical* to the pure rules for state!

Operational Semantics for I/O: Impure Part

$$\frac{e \leadsto e'}{\langle \omega; \text{print}\, e \rangle \leadsto \langle \omega; \text{print}\, e' \rangle} \qquad \frac{}{\langle \omega; \text{print}\, n \rangle \leadsto \langle (n :: \omega); \text{return}\, \langle\rangle \rangle}$$

$$\frac{e \leadsto e'}{\langle \omega; \text{return}\, e \rangle \leadsto \langle \omega; \text{return}\, e' \rangle} \qquad \frac{e \leadsto e'}{\langle \omega; \text{let}\ x = e;\ t \rangle \leadsto \langle \omega; \text{let}\ x = e';\ t \rangle}$$

$$\frac{}{\langle \omega; \text{let}\ x = \{\text{return}\, v\};\ t_1 \rangle \leadsto \langle \omega; [v/x]t_1 \rangle} \qquad \frac{\langle \omega; t_0 \rangle \leadsto \langle \omega'; t_0' \rangle}{\langle \omega; \text{let}\ x = \{t_0\};\ t_1 \rangle \leadsto \langle \omega'; \text{let}\ x = \{t_0'\};\ t_1 \rangle}$$

- State is now a list of output tokens
- All rules otherwise identical except for operations

Limitations of Monadic Style: Encapsulating Effects

```
let fact : int -> int = fun n ->
  let r = ref 1 in
  let rec loop n =
    match n with
    | 0 ->  !r
    | n -> let () = r := !r * n in
           loop (n-1)
  in
  loop n
```

- This function use *local state*
- No caller can tell if it uses state or not
- Should it have a pure type, or a monadic type?

Limitations of Monadic Style: Encapsulating Effects

```
let rec find' : ('a -> bool) -> 'a list -> 'a =
  fun p ys ->
    match ys with
    | [] -> raise Not_found
    | y :: ys -> if p y then y else find' p ys

let find : ('a -> bool) -> 'a list -> 'a option =
  fun p xs ->
    try Some (find' p xs)
    with Not_found -> None
```

- `find'` has an effect – it can raise an exception
- But `find` calls `find'`, *and* catches the exception
- Should `find` have an exception monad in its type?

Limitations of Monadic Style: Combining Effects

Suppose you have two programs:

```
p1 : (int -> ans) state
p2 : int io
```

- we write `a state` for a state monad computation
- we write `b io` for a I/O monad computation
- How do we write a program that does `p2`, and passes its argument to `p1`?

Checked Exceptions in Java

- Java *checked exceptions* implement a simple form of effect typing
- Method declarations state which exceptions a method can raise
- Programmer must catch and handle any exceptions they haven't declared they can raise
- Not much used in modern code – type system too inflexible

Effects in Koka

checked

```
fun square1( x : int ) : total int   { x*x }
fun square2( x : int ) : console int { println( "a not so secret side-effect" ); x*x }
fun square3( x : int ) : div int     { x * square3( x ) }
fun square4( x : int ) : exn int     { throw( "oops" ); x*x }
```

- Koka is a new language from Microsoft Research
- Uses effect tracking to track totality, partiality, exceptions, I/O, state and even user-defined effects
- Good playground to understand how monadic effects could look like in a practical language
- See: `https://github.com/koka-lang/koka`

Questions

For the monadic I/O language:

1. State the weakening, exchange, and substitution lemmas
2. Define machine configurations and configuration typing
3. State the type safety property