Programming = proving?
The Curry-Howard correspondence today

Fifth lecture

Can we change the world? Imperative programming, monadic effects, algebraic effects

Xavier Leroy

Collège de France

2018-12-12

COLLÈGE
DE FRANCE
1530

I

Effects in programming and in semantics

Pure functional programming

Executing a program is computing its final result, also called normal form or value.

We can also observe that the program does not terminate (divergence). (Except if the type system guarantees termination.)

Programming “in the real world”

Executing a program has an effect on the outside world:

- displaying things on the screen, writing files, …
- communicating over the network
- reading sensors, controlling actuators.

An imperative, “cooking recipe” view of programming:

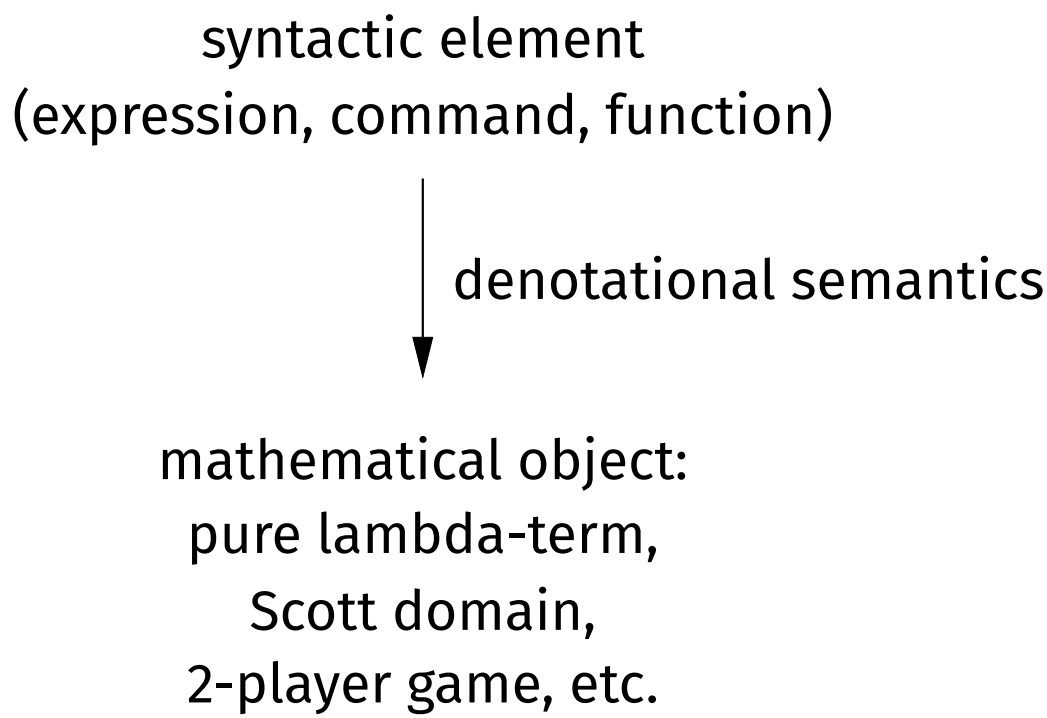
Executing a program has an effect on the computer:

- assigning variables or array elements;
- allocating, modifying, freeing data structures;
- jumping to another control point (exceptions, continuations, backtracking).

Semantics for effects

What formal semantics can we give to languages with effects?

In particular, which denotational semantics?



Semantics for mutable state

A command `x = x+1` is viewed as a state transformer:

$$\text{state where } \mathtt{x} \text{ is } n \xrightarrow{\mathtt{x} = \mathtt{x+1}} \text{state where } \mathtt{x} \text{ is } n+1$$

The denotation of a command c is therefore a function $S \to S$
from the state $s_1 : S$ at the beginning of c's execution
to the state $s_2 : S$ at the end of c's execution.

Ex: a sequence $c_1; c_2$ is the composition of denotations $[\![c_2]\!] \circ [\![c_1]\!]$.

Likewise, the denotation of an expression with effects $e : T$ is a function
$S \to T \times S$, state "before" $\mapsto$ (value, state "after).

Note: this technique of passing the current state as extra argument and extra result allows us to program imperative algorithms in pure functional languages (Haskell, Agda, Coq).

Semantics for other effects

We can change the shape of results: for an expression $e : T$,

- $[\![e]\!]$ is a set of $T \Longrightarrow$ non-determinism
- $[\![e]\!]$ is a value of type T or an exception $\Longrightarrow$ exceptions.

We can add one or several continuations:

- $[\![e]\!] = \lambda k \ldots$: control operators, non-local `goto`;
- $[\![e]\!] = \lambda k_{\text{success}}.\lambda k_{\text{failure}} \ldots$: exceptions, backtracking.

This is all *ad hoc* and not modular: adding one effect changes the whole semantics. Can we be more abstract and more modular?

II

Monads

Monads

A metaphysical concept
(Plato, Leibniz, …)

A structure in category theory
(Godement's "standard construction"; Mac Lane)

A semantic tool to describe languages with effects
(Moggi, 1989)

A technique to program with effects in a pure language
(Wadler, 1991; the Haskell community)

A tool to write programs with effects and reason over them.

The computational lambda-calculus

(Eugenio Moggi, *Computational lambda-calculus and monads*, LICS 1989; *Notions of computations and monads*, Inf. Comput. 93(1), 1991.)

To model effectful programming, Moggi was looking for a "computational" lambda-calculus and its program equivalence principles.

He chose to separate clearly

- values (results of computations) from
- computations (which eventually produce values).

A computation that produces a value of type A has type $T\,A$.

The computational lambda-calculus

Various choices for T correspond to known denotational semantics for various effects:

Non-determinism:	$T\,A = \mathcal{P}(A)$	
Exceptions:	$T\,A = A + E$	(E type of exceptions)
Mutable state:	$T\,A = S \to A \times S$	(S type of states)
Continuations	$T\,A = (A \to R) \to R$	(R type of results)

The monad structure

To give semantics to effectful languages, we need two base operations on computations:

- $\texttt{ret} : A \to T\,A$ (injection)
  $\texttt{ret}\ v$ is the trivial computation that produces value v, without effects.
- $\texttt{bind} : T\,A \to (A \to T\,B) \to T\,B$ (sequential composition)
  $\texttt{bind}\ a\ (\lambda x.b)$ performs computation a, binds its result value to x, then performs computation b, and returns its result.

(The name "monad" is a bit of a misnomer: modulo notations, $(T, \texttt{ret}, \texttt{bind})$ is a Kleisli triple, equivalent to a monad in category theory.)

Monad laws

$$\texttt{bind}\ (\texttt{ret}\ v)\ f = f\ v \qquad \text{(left neutral)}$$

$$\texttt{bind}\ a\ \texttt{ret} = a \qquad \text{(right neutral)}$$

$$\texttt{bind}\ (\texttt{bind}\ a\ f)\ g = \texttt{bind}\ a\ (\lambda x.\ \texttt{bind}\ (f\ x)\ g) \qquad \text{(associative)}$$

Alternate presentation of monads

In category theory, a monad is a triple (T, η, μ) where

$$\eta : A \to T\,A \qquad \mu : T\,(T\,A) \to T\,A \qquad T(f) : T\,A \to T\,B \text{ if } f : A \to B$$

Both presentations are related by taking $\texttt{ret} = \eta$ and

$$\begin{aligned} \texttt{bind}\; a\; f &= \mu(T(f)\; a) \\ \mu\; a &= \texttt{bind}\; a\; (\lambda y.\, y) \\ T(f) &= \lambda a.\, \texttt{bind}\; a\; (\lambda x.\, \texttt{ret}(f\; x)) \end{aligned}$$

An example of monad: non-determinism

$$T\ A = \mathcal{P}(A)$$

$$\texttt{ret}\ v = \{v\}$$

$$\texttt{bind}\ a\ f = \bigcup_{x \in a} f\ x$$

Specific operations for non-determinism:

$$\texttt{fail} = \emptyset$$

$$\texttt{choose}\ a\ b = a \cup b$$

An example of monad: exceptions

$$T\,A = A + E \qquad (E \text{ = type of exception values})$$

$$\begin{aligned}
\texttt{ret}\ v &= \texttt{inj}_1(v) \\
\texttt{bind}\ (\texttt{inj}_1(v))\ f &= f\ v \\
\texttt{bind}\ (\texttt{inj}_2(e))\ f &= \texttt{inj}_2(e) \qquad \text{(exception propagation)}
\end{aligned}$$

Specific operations for exceptions:

$$\begin{aligned}
\texttt{raise}\ e &= \texttt{inj}_2(e) \\
\texttt{try}\ a\ \texttt{with}\ x \to b &= \texttt{match}\ a\ \texttt{with}\ \texttt{inj}_1(x) \to \texttt{inj}_1(x) \mid \texttt{inj}_2(x) \to b
\end{aligned}$$

An example of monad: mutable state

$$T\,A = S \to A \times S \qquad (S \text{ = type of states})$$

$$\begin{aligned} \texttt{ret}\ v &= \lambda s.\,(v, s) \\ \texttt{bind}\ a\ f &= \lambda s_1.\,\texttt{let}\ (x, s_2) = a\ s_1\ \texttt{in}\ f\ x\ s_2 \end{aligned}$$

Specific operations: (ℓ = memory locations)

$$\begin{aligned} \texttt{get}\ \ell &= \lambda s.\,(s(\ell), s) \\ \texttt{set}\ \ell\ v &= \lambda s.\,((), s\{\ell \leftarrow v\}) \end{aligned}$$

An example of monad: continuations

$$T\,A = (A \to R) \to R \qquad (R \text{ = type of the final result})$$

$$\texttt{ret}\; v = \lambda k.\; k\; v$$

$$\texttt{bind}\; a\; f = \lambda k.\; a\; (\lambda x.\; f\; x\; k)$$

Control operators:

$$\texttt{callcc}\; f = \lambda k.\; f\; (\lambda v.\lambda k'.\; k\; v)\; k$$

$$\mathcal{C}\; f = \lambda k.\; f\; (\lambda v.\lambda k'.\; k\; v)\; (\lambda x.x)$$

Monads that combine several effects

State + exceptions: $T\,A = S \to (A + E) \times S$

Stat + continuations: $T\,A = S \to (A \to S \to R) \to R$

Continuations + exceptions: $T\,A = ((A + E) \to R) \to R$
or $T\,A = (A \to R) \to (E \to R) \to R$

Exercise: write `ret` and `bind` for these 4 monads.

See also: the monad transformers, a more systematic approach to combining effects.

Even more monads

Environment (*reader monad*): $T\,A = Env \to A$

$\texttt{ret}\ v = \lambda e.\ v$

$\texttt{bind}\ a\ f = \lambda e.\ f\ (a\ e)\ e$

Logging (*writer monad*): $T\,A = A \times \texttt{string}$

$\texttt{ret}\ v = (v, \texttt{""})$

$\texttt{bind}\ a\ f = \texttt{let}\ (x, s_1) = a\ \texttt{in}\ \texttt{let}\ (y, s_2) = f\ x\ \texttt{in}\ (y, s_1.s_2)$

Distributions: $T\,A = \mathcal{P}(A \times \mathbb{I})$ (= non-determinism + probabilities)

$\texttt{ret}\ v = \{(v, 1)\}$

$\texttt{bind}\ a\ f = \{(y, p_1 \times p_2) \mid (x, p_1) \in a,\ (y, p_2) \in f\ x\}$

$\texttt{choose}\ p\ a\ b = \{(a, p); (b, 1-p)\}$

Expectations: $T\,A = (A \to \mathbb{I}) \to \mathbb{I}$ (= continuations + probabilities)

$\texttt{ret}\ v = \lambda\mu.\ \mu\ v$

$\texttt{bind}\ a\ f = \lambda\mu.\ a\ (\lambda x.\ f\ x\ \mu)$

$\texttt{choose}\ p\ a\ b = \lambda\mu.\ p \times (a\ \mu) + (1-p) \times (b\ \mu)$

The computational lambda-calculus

$$
\begin{array}{rll}
M, N ::= & x \mid \lambda x.\, M \mid M\ N & \text{lambda-calculus} \\
\mid & \ldots & \text{products, sums, inductive types} \\
\mid & \mathtt{val}\ M & \text{trivial computation} \\
\mid & \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N & \text{sequence of 2 computations} \\
\mid & \ldots & \text{specific operations of the monad}
\end{array}
$$

For a given monad $(T, \mathtt{ret}, \mathtt{bind})$, the semantics is obtained by interpreting $\mathtt{val}\ M$ by $\mathtt{ret}\ M$ and $\mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N$ by $\mathtt{bind}\ M\ (\lambda x.\, N)$.
Equivalences:

$$(\lambda x.\, M)\ N = M\{x \leftarrow N\} \qquad (\beta)$$

$$\lambda x.\, M\ x = M \qquad (\eta)$$

$$\mathtt{let}\ x \Leftarrow \mathtt{val}\ M\ \mathtt{in}\ N = N\{x \leftarrow M\}$$

$$\mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ \mathtt{val}\ x = M$$

$$\mathtt{let}\ x \Leftarrow (\mathtt{let}\ y \Leftarrow M\ \mathtt{in}\ N)\ \mathtt{in}\ P = \mathtt{let}\ y \Leftarrow M\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow N\ \mathtt{in}\ P$$

Example program

In the non-determinism monad.

All the ways to insert an element `x` in a list `l`:

```
let rec insert x l =
  choose (val (x :: l))
         (match l with
            | [] -> fail
            | h :: t -> let t' ⇐ insert x t in val (h :: t'))
```

All the permutations of a list `l`:

```
let rec permut l =
  match l with
    | [] -> val []
    | h :: t -> let t' ⇐ permut t in insert h t'
```

The monadic transformation

Transforms an impure functional language with implicit effects (Caml, Scheme, etc) to computational lambda-calculus with monadic effects.

Makes explicit monadic effects and evaluation strategy.

Call by value

$$
\begin{aligned}
[\![cst]\!]_v &= \mathtt{val}\ cst \\
[\![\lambda x.\, M]\!]_v &= \mathtt{val}(\lambda x.\, [\![M]\!]_v) \\
[\![x]\!]_v &= \mathtt{val}\ x \\
[\![M\ N]\!]_v &= \mathtt{let}\ f \Leftarrow [\![M]\!]_v\ \mathtt{in} \\
&\qquad \mathtt{let}\ a \Leftarrow [\![N]\!]_v\ \mathtt{in}\ f\ a
\end{aligned}
$$

Note: CPS transformation = monadic transformation + continuation monad.

The monadic transformation

Transforms an impure functional language with implicit effects (Caml, Scheme, etc) to computational lambda-calculus with monadic effects.

Makes explicit monadic effects and evaluation strategy.

Call by value

$$\begin{aligned}
[\![cst]\!]_v &= \texttt{val}\ cst \\
[\![\lambda x.\, M]\!]_v &= \texttt{val}(\lambda x.\, [\![M]\!]_v) \\
[\![x]\!]_v &= \texttt{val}\ x \\
[\![M\ N]\!]_v &= \texttt{let}\ f \Leftarrow [\![M]\!]_v\ \texttt{in} \\
&\qquad \texttt{let}\ a \Leftarrow [\![N]\!]_v\ \texttt{in}\ f\ a
\end{aligned}$$

Call by name

$$\begin{aligned}
[\![cst]\!]_n &= \texttt{val}\ cst \\
[\![\lambda x.\, M]\!]_n &= \texttt{val}(\lambda x.\, [\![M]\!]_n) \\
[\![x]\!]_n &= x \\
[\![M\ N]\!]_n &= \texttt{let}\ f \Leftarrow [\![M]\!]_n\ \texttt{in} \\
&\qquad f\ [\![N]\!]_n
\end{aligned}$$

Note: CPS transformation = monadic transformation + continuation monad.

The monadic transformation

Effect on types:

$$[\![A]\!] = T\, A^*$$

$$\iota^* = \iota$$

$$(A \to B)^* = \begin{cases} A^* \to [\![B]\!] & \text{(call by value)} \\ [\![A]\!] \to [\![B]\!] & \text{(call by name)} \end{cases}$$

III

The logic behind monads

Curry-Howard for monads

In the spirit of Curry-Howard: what do monads and monadic transformations mean when viewed as propositions and transformations of propositions and proofs?

- For specific monads (continuations, exceptions): interesting “logical” interpretations.
- In general: a connection with modal logics.

Continuation monad and classical logic

As seen in the previous lecture:

Call-by-name monadic translation
for the continuation monad $T\,A = (A \to R) \to R = \neg_R \neg_R A$
$\Rightarrow$ relative negative translation
from classical logic to minimal logic.

$$\begin{aligned}
[\![A]\!]_R &= \neg_R \neg_R A & [\![P \Rightarrow Q]\!]_R &= \neg_R \neg_R ([\![P]\!]_R \Rightarrow [\![Q]\!]_R) \\
[\![P \wedge Q]\!]_R &= \neg_R \neg_R ([\![P]\!]_R \wedge [\![Q]\!]_R) & [\![P \vee Q]\!]_R &= \neg_R \neg_R ([\![P]\!]_R \vee [\![Q]\!]_R) \\
[\![\forall x.\, P]\!]_R &= \neg_R \neg_R \forall x.\, [\![P]\!]_R & [\![\exists x.\, P]\!]_R &= \neg_R \neg_R \exists x.\, [\![P]\!]_R
\end{aligned}$$

The `callcc` operation of the monad corresponds to Clavius's law, and the $\mathcal{C}$ operation to double negation elimination.

Exception monad and ex falso quodlibet

Call-by-name monadic translation
for the exception monad $T\,A = A + E$
$\Rightarrow$ a translation from intuitionistic logic to minimal logic.

$$\llbracket \bot \rrbracket = E \qquad \llbracket A \rrbracket = A \vee E \ \text{ if } A \text{ atomic}$$
$$\llbracket P \Rightarrow Q \rrbracket = (\llbracket P \rrbracket \Rightarrow \llbracket Q \rrbracket) \vee E$$
$$\llbracket P \wedge Q \rrbracket = (\llbracket P \rrbracket \wedge \llbracket Q \rrbracket) \vee E \qquad \llbracket P \vee Q \rrbracket = (\llbracket P \rrbracket \vee \llbracket Q \rrbracket) \vee E$$
$$\llbracket \forall x.\, P \rrbracket = (\forall x.\, \llbracket P \rrbracket) \vee E \qquad \llbracket \exists x.\, P \rrbracket = (\exists x.\, \llbracket P \rrbracket) \vee E$$

The rule $\bot \Rightarrow P$, *ex falso quod libet*, becomes derivable after translation:
$E \Rightarrow \cdots \vee E$.
It corresponds to the `raise` operation of the monad.

Monad = modality?

$$\texttt{ret} : A \to T\,A$$

$$\texttt{bind} : T\,A \to (A \to T\,B) \to T\,B$$

The types for the `ret` and `bind` monad operations are reminiscent of rules of modal logic, viewing the type constructor T as a modality.

Modal logics

Qualify logical propositions by modalities that describe aspects of truth.

For example, following Aristotle, we can distinguish necessary truths $\Box P$, contingent truths A, and possible truths $\Diamond P$.

The $\Box$ and $\Diamond$ modalities are connected in classical logic:

$$\Box\neg P \iff \neg\Diamond P \qquad \Diamond\neg P \iff \neg\Box P$$

They can be interpreted in various ways:

- Alethic: $\Box$ = necessarily, $\Diamond$ = possibly.
- Temporal: $\Box$ = forever, $\Diamond$ = eventually.
- Geographic: $\Box$ = everywhere, $\Diamond$ = somewhere.

Other modalities can be considered, for instance “known by agent i” in epistemic logics.

Modal logics

Many different axiomatizations, depending on the intended meaning of modalities.

Example: in modal logic S4, the rules for $\Box$ are:

$$\Box P \text{ if } P \text{ is a classical tautology} \tag{N}$$
$$\Box(P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q) \tag{K}$$
$$\Box P \Rightarrow P \tag{T}$$
$$\Box P \Rightarrow \Box\Box P \tag{4}$$

The rules for $\Diamond$ follow from the definition $\Diamond P \stackrel{def}{=} \neg\Box\neg P$.

Monad = modality?

$$\texttt{ret} : A \to T\,A$$

$$\texttt{bind} : T\,A \to (A \to T\,B) \to T\,B$$

The type of `ret` can be read as $A \Rightarrow \Diamond A$, suggesting that T is the $\Diamond$ modality, “possibly”.

However, the type of `bind` is logically false: ✘ $\Diamond A \Rightarrow (A \Rightarrow \Diamond B) \Rightarrow \Diamond B$.

Symmetrically, if T is read as the $\Box$ modality, “necessarily”, the type of `bind` is valid, but not that of `ret`: ✘ $A \Rightarrow \Box A$.

The lax modality $\bigcirc$

(Mendler, 1991; Fairtlough and Mendler, 1997, 2003)

Introduced by Mendler in the context of formal verification of hardware circuits, the $\bigcirc P$ modality can be read as "P is true under some conditions", or as $C \Rightarrow P$ for an implicit condition C.

It is characterized by the axioms

$$P \Rightarrow \bigcirc P \tag{I}$$

$$\bigcirc \bigcirc P \Rightarrow \bigcirc P \tag{M}$$

$$(P \Rightarrow Q) \Rightarrow (\bigcirc P \Rightarrow \bigcirc Q) \tag{Ext}$$

$$\bigcirc P \wedge \bigcirc Q \Rightarrow \bigcirc (P \wedge Q) \tag{S}$$

Monad = lax modality

(Benton, Bierman, de Paiva, JFP(8), 1998)

The type constructor T of a monad corresponds to the lax modality $\bigcirc$. The axioms of the modality are realized by terms of the computational lambda-calculus.

$$\begin{aligned}
\texttt{val} &: P \Rightarrow \bigcirc P \\
\lambda x.\, \texttt{let}\ y \Leftarrow x\ \texttt{in}\ y &: \bigcirc \bigcirc P \Rightarrow \bigcirc P \\
\lambda f.\lambda x.\, \texttt{let}\ v \Leftarrow x\ \texttt{in}\ \texttt{val}(f\ v) &: (P \Rightarrow Q) \Rightarrow (\bigcirc P \Rightarrow \bigcirc Q) \\
\lambda x.\, \texttt{let}\ v_1 \Leftarrow \pi_1(x)\ \texttt{in} & \\
\quad \texttt{let}\ v_2 \Leftarrow \pi_2(x)\ \texttt{in}\ &: \bigcirc P \wedge \bigcirc Q \Rightarrow \bigcirc(P \wedge Q) \\
\quad \texttt{val}(v_1, v_2) &
\end{aligned}$$

Another modal encoding

(Pfenning and Davies, MSCS(11), 2001)

We can also encode the types of a monadic language using the standard $\Box$ and $\Diamond$ modalities:

$$\begin{aligned} [\![\iota]\!] &= \iota \quad \text{for base types } \iota \\ [\![A \to B]\!] &= \Box[\![A]\!] \Rightarrow [\![B]\!] \\ [\![T\ A]\!] &= \Diamond\Box[\![A]\!] \end{aligned}$$

Temporal logic intuitions:

- a value of type A is stable against future effects
  $\Longrightarrow \Box[\![A]\!]$, "forever A";
- a computation of type A, after performing effects, will eventually produce a value of type A
  $\Longrightarrow \Diamond\Box[\![A]\!]$, "eventually, forever A".

IV

Monads that support logic

Dependent types, preconditions, postconditions

In a dependently-typed language (like Agda, Coq, or F*), we can write very precise types, such as

$\forall x : A.\ P(x) \to B$	function taking an $x : A$ and a proof of $P(x)$
$\{y : B \mid Q(y)\}$	pair of a $y : B$ and a proof of $Q(y)$
$\forall x : A.\ P(x) \to \{y : B \mid Q(x, y)\}$	function $A \to B$ respecting the precondition P and the postcondition Q

Example: Euclidean division.

```
div: ∀(a b: nat), b > 0 -> { q | ∃r, a = b * q + r ∧ 0 <= r < b }
```

State monad: invariants, monotonic evolution

$$T\ A = S \to A \times S$$

We can enforce an invariant $Inv : S \to \texttt{Prop}$ over states by replacing S by a subset type S_{Inv}:

$$T\ A = S_{Inv} \to A \times S_{Inv} \quad \text{with} \quad S_{Inv} = \{s : S \mid Inv\ s\}$$

We can also enforce monotonic evolution of states w.r.t. an order $Ord : S \to S \to \texttt{Prop}$:

$$T\ A = \forall (s : S),\ A \times \{s' : S \mid Ord\ s\ s'\}$$

A monotonic state: time

Assume the state is just a timestamp. We can guarantee that computations do not “go back in time” using the monad

$$T\,A = \forall (t : Z),\ A \times \{t' : Z \mid t \leq t'\}$$

A computation $c : T\,A$ in this monad automatically guarantees that $c\,t_1 = (v, t_2) \Rightarrow t_2 \geq t_1$.

This greatly helps establishing uniqueness properties of timestamps:

```
let t1 ⇐ timestamp in
let x ⇐ f ... in
let t2 ⇐ timestamp in
(t1, x, t2)
```

Regardless of `f`’s effects, we know that `t1` $<$ `t2` and therefore `t1` $\neq$ `t2`.

A monotonic state: time

Monad operations are more complex and contain proof terms:

```
Definition T(A: Type) := forall (t: Z), A *  t' : Z | t <= t' .

Definition ret (A: Type) (a: A) : T A :=
  fun (t: Z) => (a, exist _ t (Z.le_refl t)).

Definition bind (A B: Type) (a: T A) (f: A -> T B) : T B :=
  fun (t1: Z) =>
    let '(x, exist _ t2 p12) := a t1 in
    let '(y, exist _ t3 p23) := f x t2 in
    (y, exist _ t3 (Z.le_trans t1 t2 t3 p12 p23)).

Definition timestamp : T Z :=
  fun (t: Z) => (t, exist _ (Z.succ t) (Z.le_succ_diag_r t)).
```

Hoare Type Theory (HTT)

(Nanevski et al, ICFP 2008, POPL 2010.)

Instead of fixing in advance expected properties of one state (*Inv*) or two states (*Ord*), we can also parameterize the state monad by any precondition P and any postcondition Q.

$$pre \stackrel{def}{=} S \to \texttt{Prop}$$

$$post\ A \stackrel{def}{=} A \to S \to S \to \texttt{Prop}$$

$$ST\ :\ pre \to \forall(A : \texttt{Type}), post\ A \to \texttt{Type}$$

$$ST\ P\ A\ Q \stackrel{def}{=} \forall(s_1 : S),\ P\ s_1 \to \{(a, s_2) : A \times S \mid Q\ a\ s_1\ s_2\}$$

A computation $c : ST\ P\ A\ Q$ is the functional, monadic equivalent of a command c satisfying the Hoare triple $\{P\}\ c\ \{Q\}$:
evaluated in an initial state s_1 satisfying P, the computation c produces a value a and a final state s_2 satisfying Q.

Typing the operations

We can give frighteningly precise types to the operations of the state monad:

$$\texttt{ret} : \forall (A : \texttt{Type})(v : A), ST\ (\lambda s_1.\top)\ A\ (\lambda x, s_1, s_2.\ s_2 = s_1 \wedge x = v)$$

$$\begin{aligned}\texttt{get} : \ & \forall (A : \texttt{Type})(l : loc\ A), \\ & ST\ (\lambda s_1.\ \texttt{valid}\ l\ s_1)\ A\ (\lambda x, s_1, s_2.\ s_2 = s_1 \wedge x = get\ l\ s_1)\end{aligned}$$

$$\begin{aligned}\texttt{set} : \ & \forall (A : \texttt{Type})(l : loc\ A)(v : A), \\ & ST\ (\lambda s_1.\ \texttt{valid}\ l\ s_1)\ \texttt{unit}\ (\lambda x, s_1, s_2.\ s_2 = set\ l\ v\ s_1 \wedge x = \texttt{tt})\end{aligned}$$

$$\begin{aligned}\texttt{bind} : \ & \forall (A\ B : \texttt{Type})(P_1 : pre)(Q_1 : post\ A)(P_2 : A \to pre)(Q_2 : A \to post\ B), \\ & ST\ P_1\ A\ Q_1 \to (\forall (a : A), ST\ (P_2\ a)\ B\ (Q_2\ a)) \to ST\ P\ B\ Q\end{aligned}$$

where $P = \lambda s_1.\ P_1\ s_1 \wedge \forall a, s_2.\ Q_1\ a\ s_1\ s_2 \Rightarrow P_2\ s_2$
and $Q = \lambda b, s_1, s_3.\ \exists a, s_2.\ Q_1\ a\ s_1\ s_2 \wedge Q_2\ a\ b\ s_2\ s_3$.

Weakest preconditions and predicate transformers

Since Dijkstra (1975), we know that for any command c and postcondition Q, there exists a weakest precondition P such that $\{P\}\ c\ \{Q\}$.

It can be defined as a function of c and Q: $\quad P = wp(c, Q)$.

In other words: the behavior of command c is entirely characterized by the predicate transformer $Q \mapsto wp(c, Q)$, that is, a function
W : *postcondition* $\mapsto$ *weakest precondition*.

The Dijkstra monad

(Swamy et al, PLDI 2013, POPL 2016)

A state monad *ST A W* that describes computations producing values of type *A* and satisfying the predicate transformer *W*.

$$pre \stackrel{def}{=} S \to \texttt{Prop}$$

$$post\ A \stackrel{def}{=} A \to S \to \texttt{Prop}$$

$$wptransf\ A \stackrel{def}{=} post\ A \to pre$$

$$ST\ :\ \forall (A : \texttt{Type}), wptransf\ A \to \texttt{Type}$$

$$ST\ A\ W \stackrel{def}{=} \forall (Q : post\ A)(s_1 : S),\ W\ Q\ s_1 \to \{(a, s_2) : A \times S \mid Q\ a\ s_1\ s_2\}$$

Typing operations of the Dijkstra monad

The types for the operations of the Dijkstra monad are slightly simpler than those of the HTT monad, and better support inference by unification.

$$\texttt{ret} : \forall (A : \texttt{Type})(x : A), ST\ A\ (\lambda Q.\ Q\ x)$$

$$\begin{aligned}\texttt{get} : \ & \forall (A : \texttt{Type})(l : loc\ A), \\ & ST\ A\ (\lambda Q.\lambda s.\ \texttt{valid}\ l\ s \wedge Q\ (\texttt{get}\ l\ s)\ s)\end{aligned}$$

$$\begin{aligned}\texttt{set} : \ & \forall (A : \texttt{Type})(l : loc\ A)(v : A), \\ & ST\ \texttt{unit}\ (\lambda Q.\lambda s.\ \texttt{valid}\ l\ s \wedge Q\ \texttt{tt}\ (\texttt{set}\ l\ v\ s))\end{aligned}$$

$$\begin{aligned}\texttt{bind} : \ & \forall (A\ B : \texttt{Type})(W_1 : wptransf\ A)(W_2 : A \to wptransf\ B), \\ & ST\ A\ W_1 \to (\forall (a : A), ST\ B\ (W_2\ a)) \to ST\ B\ (\lambda Q.\ W_1\ (\lambda a.\ W_2\ a\ Q))\end{aligned}$$

Moreover, the “Dijkstra monad” approach extends to other effects (partiality, exceptions) and to their combinations $\Longrightarrow$ the F* language.

V

Algebraic effects and effect handlers

Where do effects come from?

Moggi's computational lambda-calculus, and more generally the monadic approach, accounts for propagation and sequencing of effects in a generic manner (independently of the kind of effects considered).

Can we account (in a generic manner too) for the base operations that create effects? For example,

- Input/output: `print`, `read`
- Exceptions: `raise`
- Mutable state: `set`, `get`
- Non-determinism: `choose`, `fail`.

Plotkin and Power (2003) introduce an algebraic presentation of these operations that create effects.

Algebraic structures

In mathematics, an algebraic structure is a set equipped with operations that satisfy identities (equations).

Example: a group is a set G with three operations:
a constant 1, a binary operation $\cdot$, a unary operation $^{-1}$,
satisfying the identities

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$1 \cdot x = x = x \cdot 1$$
$$x \cdot x^{-1} = 1 = x^{-1} \cdot x$$

Algebraic abstract types

In computing, an algebraic abstract type is an abstract type
(= a type name + operations) specified by equations over the operations.

Example: functional arrays (operations `get`, `set`)

$$\texttt{get}\ i\ (\texttt{set}\ i\ v\ t) = v$$
$$\texttt{get}\ i\ (\texttt{set}\ j\ v\ t) = \texttt{get}\ i\ t \quad \text{if } i \neq j$$

Example: stacks (operations `empty`, `push`, `pop`, `top`)

$$\texttt{top}\ (\texttt{push}\ v\ s) = v$$
$$\texttt{pop}\ (\texttt{push}\ v\ s) = s$$

Algebraic effects

(Plotkin, Power, Pretnar, et al; 2003–)

Values:	v	$::= x \mid cst \mid \lambda x.\, M$	
Computations:	M, N	$::= \texttt{val}\ v$	trivial computation
		$\mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$	sequence of 2 computations
		$\mid v\ v'$	application
		$\mid op(\vec{v}; y.\, M)$	effectful operation

The term $op(v_1 \ldots v_n; y.\, M)$ stands for an operation that produces an effect. The values v_i are the parameters. The operation produces a result value that is bound to y in continuation M.

Notation: $op(\vec{v}) \stackrel{def}{=} op(\vec{v}; y.\, \texttt{val}(y))$ (trivial continuation).

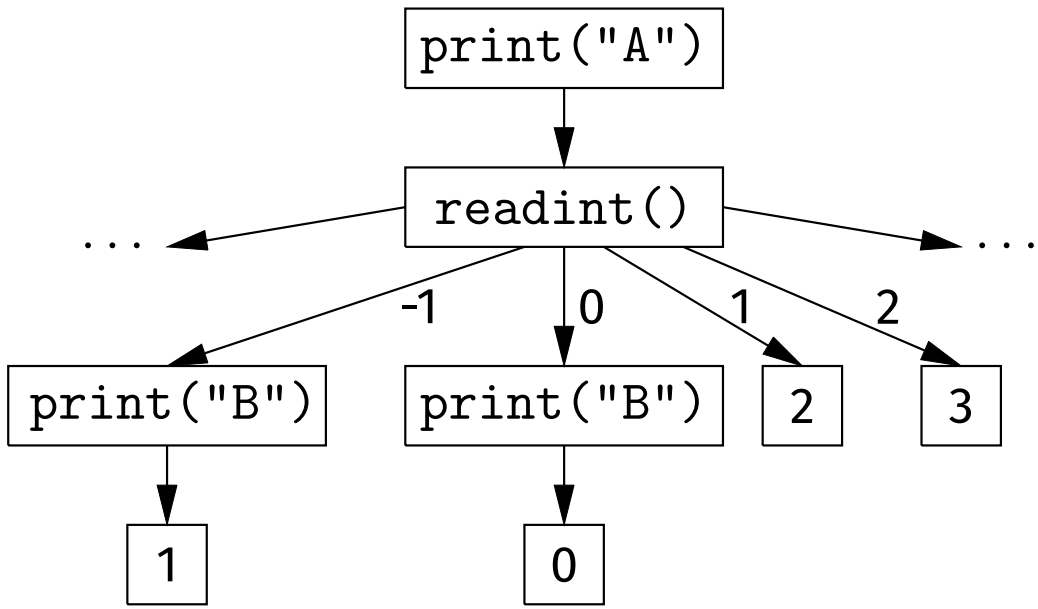
Semantically, we have the equivalence $op(\vec{v}, y.\, M) = \texttt{let}\ y \Leftarrow op(\vec{v})\ \texttt{in}\ M$.

Example: input/output

(Pretnar, *An introduction to algebraic effects and handlers*, MFPS 2015)

Operations: `print` that takes a string and `readint` that returns an integer.

```
let _ ⇐ print("A") in
let n ⇐ readint() in
if n <= 0 then
  (let _ ⇐ print("B")
   in val (-n))
else
  val (n+1)
```



Intuitive semantics: a tree of actions, with operations at the nodes and values (or $\bot$) at the leaves.

Equations over effects

The I/O effects are "free": after an output, all inputs remain possible. This is not the case for other effects. For mutable state (operations $\mathtt{get}$ and $\mathtt{set}$ over locations ℓ), we have at least the following equations:

$$\mathtt{set}(\ell, v;\ \_.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{set}(\ell, v;\ \_.\, M\{z \leftarrow v\})$$
$$\mathtt{set}(\ell, v;\ \_.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{set}(\ell, v; \_.\, M)) \quad \text{if } \ell' \neq \ell$$

For completeness we can add

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{get}(\ell; y.\, M\{z \leftarrow y\}) \qquad \text{(double read)}$$
$$\mathtt{get}(\ell; y.\, \mathtt{set}(\ell, y;\ \_.\, M)) = M \qquad \text{(read then rewrite)}$$
$$\mathtt{set}(\ell, v_1; \_.\, \mathtt{set}(\ell, v_2; \_.\, M)) = \mathtt{set}(\ell, v_2; \_.\, M) \qquad \text{(double write)}$$
$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{get}(\ell; y.\, M)) \quad \text{if } \ell' \neq \ell$$
$$\mathtt{set}(\ell, v; y.\, \mathtt{set}(\ell', v'; z.\, M)) = \mathtt{set}(\ell', v'; z.\, \mathtt{set}(\ell, v; y.\, M)) \quad \text{if } \ell' \neq \ell$$

Handling effects

For I/O or mutable state, we can imagine that effects are executed by the operating system or the runtime system of the language.

Can we enable the program to handle ("execute") itself some of the effects it produces?

Exception handling

`raise`(e) can be viewed as an operator producing the "exception e" effect. It can be handled by the construct

$$\texttt{try}\ a\ \texttt{with}\ x \to b$$

that catches exceptions raised by a and then evaluates b (the exception handler).

Some languages (Common Lisp, Dylan) allow the handler to restart the computation at the point where the exception was raised. We can model this by a parameter k to the handler, bound to the continuation of the `raise`(e) exception:

$$\texttt{try}\ a\ \texttt{with}\ (x,k) \to \texttt{if}\ \ldots\ \texttt{then}\ k\ 0\ \texttt{else}\ b$$

(resume with value 0 for the raise) (abort with value b)

Effect handlers

Values:	v	$::= x \mid cst \mid \lambda x.\, M$	
Computations:	M, N	$::= \texttt{val}\ v$	trivial computation
		$\mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$	sequencing of 2 computations
		$\mid v\ v'$	application
		$\mid op(\vec{v}; y.\, M)$	effectful operation
		$\mid \texttt{with}\ H\ \texttt{handle}\ M$	effect handler
Handlers:	H	$::= \{\ \texttt{val}(x) \to M_{val};$	
		$\quad op_1(\vec{x}; k) \to M_1;$	
		$\quad \ldots$	
		$\quad op_n(\vec{x}; k) \to M_n\ \}$	

In `with` H `handle` M,

- if M performs $op_i(\vec{v}; y.\, N)$, the M_i case is evaluated with $\vec{x} = \vec{v}$ and $k = \lambda y.\, N$;
- if M evaluates `val` v, the M_{val} case is evaluated with $x = v$.

Examples of effect handlers

Exception handling:

```
with { val(x) → val(x);
       raise(e; k) → if ... then k 0 else b }
handle a
```

Invert the order of `print` operations performed:

```
with { val(x) → val(x);
       print(s; k) → let _ ⇐ k() in print(s) }
handle a
```

Collect `print` operations in a character string:

```
with { val(x) → val(x, "");
       print(s; k) → let (x, acc) ⇐ k()
                     in val (x, concat s acc) }
```

(This changes the type of the computation: from A to $A \times \texttt{string}$.)

Examples of effect handlers

Non-determinism by backtracking:
(`choose()` is an effect that returns `true` or `false` non-deterministically)

```
with { val(x) → val(x);
       choose(_; k) → with { fail(_; k') → k false }
                      handle k true }
```

Mutable state:

```
with { val(x) → λs. (x, s);
       get(l; k) → λs. (k (lookup l s)) s;
       set(l, v; k) → λs. (k ()) (update l v s) }
```

(This changes the type of the computation: from A to $S \to A \times S$.)

Ongoing work on algebraic effects

Static typing that keeps tracks of effects, for example

Value types: $A ::= \iota \mid A_1 \times A_2 \mid A \to C \mid C_1 \Rightarrow C_2$ (handler type)

Computation types: $C ::= A!\{op_1, \ldots, op_n\}$

Language designs and implementations:

- Eff `https://www.eff-lang.org`
- Frank `https://github.com/frank-lang/frank`
- Multicore OCaml `https://github.com/ocamllabs/ocaml-multicore/wiki`

VI

Concluding remarks

Monadic effects, algebraic effects

A success for the “categorical” approach to programming languages.

> *The view that “category theory comes, logically, before the λ-calculus” led us to consider a categorical semantics of computations first, rather than to modify directly the rules of $\beta\eta$-conversion to get a correct calculus.*
>
> *(E. Moggi, Notions of Computations and Monads, 1991)*

Not a success for the “Curry-Howard” approach:
the connections with mathematical logic are weak.

VII

Further reading

Further reading

Programming with monads:

- *All About Monads*, `https://wiki.haskell.org/All_About_Monads`

Programming and proving with Dijkstra monads:

- *Verified programming in F**, `https://www.fstar-lang.org/tutorial/`

Algebraic effects and effect handlers:

- Matija Pretnar, *An Introduction to Algebraic Effects and Handlers*, tutorial, MFPS 2015, `https://www.eff-lang.org/handlers-tutorial.pdf`
- Andrej Bauer, *Algebraic effects and handlers*, OPLSS 2018 summer school, `https://www.cs.uoregon.edu/research/summerschool/summer18/lectures/bauer_notes.pdf`