Control structures, sixth lecture

The theory of effects: from monads to algebraic effects

Xavier Leroy
2024-02-29

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
The effects of a program

Whatever goes beyond computing the final value of the program.

Effects on the outside world:

• display things on the screen, write to files, …
• communicate over the network;
• read sensors, send commands to actuators;
• terminate or diverge (for some authors).

Effects on the state of the computer:

• assignments to variables, to array elements;
• allocation, modification, deallocation of data structures;
• jumps to alternate program points (exceptions, continuations, backtracking).

Which theories can account for all these kinds of effects?
Monads
Monads

A philosophical concept (metaphysics)  
(Platon, Leibniz, …)

A structure in category theory  
(Godement’s “standard construction”; Mac Lane)

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

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

A tool to formalize effectful programs and reason about them.
A proliferation of denotational semantics

In lecture #4, we saw several forms of denotational semantics:

\[
\begin{align*}
\llbracket \text{stmt} \rrbracket & : \text{Store} \to \text{Store}_\bot & \text{ (mutable state)} \\
\llbracket \text{stmt} \rrbracket & : \text{Env} \to \text{Store} \to \text{Store}_\bot & \text{ (environment + state)} \\
\llbracket \text{stmt} \rrbracket & : \text{Env} \to \text{Store} \to (\text{Store} \to \text{Res}_\bot) \to \text{Res}_\bot & \text{ (environment + state + goto)}
\end{align*}
\]

The semantics of base constructs such as sequencing changes every time we add a feature to the language:

\[
\begin{align*}
\llbracket s_1; s_2 \rrbracket \sigma & = \llbracket s_2 \rrbracket (\llbracket s_1 \rrbracket \sigma) \\
\llbracket s_1; s_2 \rrbracket \rho \sigma & = \llbracket s_2 \rrbracket \rho (\llbracket s_1 \rrbracket \rho \sigma) \\
\llbracket s_1; s_2 \rrbracket \rho \sigma \kappa & = \llbracket s_1 \rrbracket \rho \sigma (\lambda \sigma'. \llbracket s_2 \rrbracket \rho \sigma' \kappa)
\end{align*}
\]
In lectures #4 and #5, we saw several transformations over functional programs:

- $\mathcal{C}$, the CPS (continuation-passing style) transformation, to make evaluation strategy explicit and to account for callcc.
- $\mathcal{C}^2$, the “double-barreled” CPS transformation, to account for structured exceptions and exception handling;
- $\mathcal{E}$, the ERS (Exception-Returning Style) transformation, another way to account for exceptions.
Commonalities between these transformations

For constants and $\lambda$-abstractions:

\[
\begin{align*}
C(\text{cst}) &= \lambda k. \, k \, \text{cst} \\
C^2(\text{cst}) &= \lambda k_1 k_2. \, k_1 \, \text{cst} \\
E(\text{cst}) &= V \, \text{cst}
\end{align*}
\]

\[
\begin{align*}
C(\lambda x. \, M) &= \lambda k. \, k \, (\lambda x. \, C(M)) \\
C^2(\lambda x. \, M) &= \lambda k_1 k_2. \, k_1 \, (\lambda x. \, C^2(M)) \\
E(\lambda x. \, M) &= V \, (\lambda x. \, E(M))
\end{align*}
\]

In all cases, we return a value ($\text{cst}$ or $\lambda x \ldots$) presenting it as a trivial computation.
Commonalities between these transformations

For let bindings:

\[
C(\text{let } x = e_1 \text{ in } e_2) = \lambda k. C(e_1) (\lambda x. C(e_2) k)
\]

\[
C^2(\text{let } x = e_1 \text{ in } e_2) = \lambda k_1 k_2. C^2(e_1) (\lambda x. C^2(e_2) k_1 k_2) k_2
\]

\[
E(\text{let } x = e_1 \text{ in } e_2) = \text{match } E(e_1) \text{ with } E x \rightarrow E x \mid V x \rightarrow E(e_2)
\]

In the three transformations, we perform the computation of \(e_1\), extract the resulting value, **bind** it to \(x\), and continue with the computation of \(e_2\).
Commonalities between these transformations

For function applications:

\[ C(e_1 e_2) = \lambda k. C(e_1)(\lambda v_1. C(e_2)(\lambda v_2. v_1 v_2 k)) \]

\[ C^2(e_1 e_2) = \lambda k_1. \lambda k_2. C^2(e_1)(\lambda v_1. C^2(e_2)(\lambda v_2. v_1 v_2 k_1 k_2) k_2) k_2 \]

\[ \mathcal{E}(e_1 e_2) = \text{match } \mathcal{E}(e_1) \text{ with } E \ x_1 \rightarrow E \ x_1 \mid V \ v_1 \rightarrow \]

\[ \quad \text{match } \mathcal{E}(e_2) \text{ with } E \ x_2 \rightarrow E \ x_2 \mid V \ v_2 \rightarrow v_1 \ v_2 \]

In the three transformations, we **bind** the value of \( e_1 \) to \( v_1 \), then **bind** the value of \( e_2 \) to \( v_2 \), then apply \( v_1 \) to \( v_2 \).
The computational lambda-calculus


To facilitate the writing and evolution of denotational semantics and program transformations, Moggi designed a “computational lambda-calculus” and its equivalence principles.

He chose to distinguish clearly between

• values (the final results of computations), and
• computations (producing values).

“Values are; computations do.” (P. B. Levy)

A computation producing a value of type $A$ has type $T A$
(where $T$ is a type constructor that depends on the effects considered)
Different choices for $T$ correspond to known denotational semantics / program transformations for different effects:

- **Environments:** $T A = Env \rightarrow A$
- **Mutable state:** $T A = S \rightarrow A \times S$ \hspace{1cm} ($S$ = type of states)
- **Exceptions:** $T A = A + Exn$
- **Non-determinism:** $T A = \mathcal{P}(A)$
- **Continuations:** $T A = (A \rightarrow R) \rightarrow R$ \hspace{1cm} ($R$ = type of results)
To give semantics to effectful languages, we need two base operations over computations:

1. \texttt{ret} : \texttt{A} \rightarrow \texttt{T A} \quad \text{(injection)}

\texttt{ret} \; v \text{ is the trivial computation that produces value } v \text{ and has no effects.}

2. \texttt{bind} : \texttt{T A} \rightarrow (\texttt{A} \rightarrow \texttt{T B}) \rightarrow \texttt{T B} \quad \text{(sequential composition)}

\texttt{bind} \; a \; (\lambda x. b) \text{ executes the computation } a, \text{ bind its result value to } x, \text{ then executes the computation } b, \text{ and returns the result value of } b.
The monad structure

To define `ret` and `bind`, Moggi uses a monad from category theory, that is, a triple \((T, \eta, \mu)\) where

\[\eta : A \to T A \quad \mu : T (T A) \to T A \quad T(f) : T A \to T B \text{ if } f : A \to B\]

satisfying certain laws.

We can then define the Kleisli triple \((T, \text{ret}, \text{bind})\) as:

\[
\begin{align*}
\text{ret } v & \overset{\text{def}}{=} \eta(v) \\
\text{bind } a f & \overset{\text{def}}{=} \mu(T(f) a)
\end{align*}
\]

(Nowadays, computer scientists prefer to define the Kleisli triple directly, and call it “a monad” by abuse of terminology.)
The laws of monads (Kleisli triples)

\[
\begin{align*}
\text{bind (ret } v) f &= f v & \text{(left neutral)} \\
\text{bind } a \text{ ret} &= a & \text{(right neutral)} \\
\text{bind (bind } a f) g &= \text{bind } a (\lambda x. \text{bind } (f x) g) & \text{(associativity)}
\end{align*}
\]
Non-determinism as a monad

\[ T A = \mathcal{P}(A) \quad \text{(or List}(A)\text{)} \]

\[ \text{ret } v = \{ v \} \]

\[ \text{bind } a f = \bigcup_{x \in a} f x \]

Operations specific to non-determinism:

\[ \text{fail} = \emptyset \]

\[ \text{choose } a b = a \cup b \]
Exceptions as a monad

\[ T A = V \text{ of } A \mid E \text{ of } Exn \quad (\approx A + Exn) \]

\[
\begin{align*}
\text{ret } v &= V v \\
\text{bind } (V v)f &= f v \\
\text{bind } (E e)f &= E e \quad \text{(exception propagation)}
\end{align*}
\]

Operations specific to exceptions:

\[
\begin{align*}
\text{raise } e &= E e \\
\text{try } a \text{ with } x \to b &= \text{match } a \text{ with } V v \to V v \mid E x \to b
\end{align*}
\]
Mutable state as a monad

\[ T A = S \rightarrow A \times S \quad (S = \text{type of states}) \]

\[
\text{ret } v = \lambda s. (v, s)
\]

\[
\text{bind } a f = \lambda s_1. \text{let } (x, s_2) = a s_1 \text{ in } f x s_2 \quad \text{(threading the state)}
\]

Specific operations: \((\ell = \text{reference identifier})\)

\[
\text{get } \ell = \lambda s. (s(\ell), s)
\]

\[
\text{set } \ell v = \lambda s. (\ell, s[\ell \leftarrow v])
\]
Continuations as a monad

\[ T \mathcal{A} = (A \to R) \to R \quad (R = \text{type of the final result}) \]

\[
\begin{align*}
\text{ret } v &= \lambda k. \ k \ v \\
\text{bind } a \ f &= \lambda k. \ a \ (\lambda x. \ f \ x \ k)
\end{align*}
\]

Control operator:

\[
\begin{align*}
\text{callcc } f &= \lambda k. \ f \ (\lambda v. \lambda k'. \ k \ v) \ k
\end{align*}
\]
Monads that combine several effects

State + exceptions:

\[ T A = S \rightarrow (A + E) \times S \]

State + continuations:

\[ T A = S \rightarrow (A \rightarrow S \rightarrow R) \rightarrow R \]

Continuations + exceptions:

\[ T A = ((A + E) \rightarrow R) \rightarrow R \]

or \[ T A = (A \rightarrow R) \rightarrow (E \rightarrow R) \rightarrow R \]

*Exercise:* define `ret` and `bind` for these 4 monads.
A computational lambda-calculus

“Values are; computations do.”

Values:
\[ v ::= \text{cst} \mid x \mid \lambda x. M \]

Computations:
\[ M, N ::= v_1 v_2 \quad \text{application} \]
\[ \mid \text{if } v \text{ then } M \text{ else } N \quad \text{conditional} \]
\[ \mid \text{val } v \quad \text{trivial computation} \]
\[ \mid \text{do } x \leftarrow M \text{ in } N \quad \text{sequencing of computations} \]
\[ \mid \ldots \quad \text{monad-specific operations} \]

For a given monad \((T, \text{ret}, \text{bind})\), the semantics is obtained by interpreting \text{val } M \text{ as } \text{ret } M \text{ and } \text{do } x \leftarrow M \text{ in } N \text{ as } \text{bind } M (\lambda x. N).
The laws of the computational lambda-calculus

Function application:

$$(\lambda x. M) v = M\{x \leftarrow v\}$$

The three monadic laws:

$$\text{do}\ x \leftarrow \text{val}\ v\ \text{in}\ M = M\{x \leftarrow v\}$$

$$\text{do}\ x \leftarrow M\ \text{in}\ \text{val}\ x = M$$

$$\text{do}\ x \leftarrow (\text{do}\ y \leftarrow M\ \text{in}\ N)\ \text{in}\ P = \text{do}\ y \leftarrow M\ \text{in}\ (\text{do}\ x \leftarrow N\ \text{in}\ P)$$
The monadic transformation

Transforms an impure functional language with implicit effects into the computational lambda-calculus with explicit monadic effects.

\[ \mathcal{M}(\text{cst}) = \text{val cst} \]

\[ \mathcal{M}(\lambda x. e) = \text{val} (\lambda x. \mathcal{M}(e)) \]

\[ \mathcal{M}(x) = \text{val} x \]

\[ \mathcal{M}(\text{let } x = e_1 \ \text{in } e_2) = \text{do } x \leftarrow \mathcal{M}(e_1) \ \text{in } \mathcal{M}(e_2) \]

\[ \mathcal{M}(e_1 \ e_2) = \text{do } f \leftarrow \mathcal{M}(e_1) \ \text{in } \text{do } v \leftarrow \mathcal{M}(e_2) \ \text{in } f \ v \]

\[ \mathcal{M}(\text{if } e_1 \ \text{then } e_2 \ \text{else } e_3) = \text{do } b \leftarrow \mathcal{M}(e_1) \ \text{in } \]

\[ \quad \text{if } b \ \text{then } \mathcal{M}(e_2) \ \text{else } \mathcal{M}(e_3) \]

By combining this transformation with the appropriate monads, we recover the CPS / ERS / double-barreled CPS transformations.
(Notations do in Haskell, let* in OCaml.)

We can write code that can be used in any monad, e.g. a monadic map iterator:

```ocaml
let (let*) = bind
let rec mmap (f: 'a -> 'b t) (l: 'a list) : 'b list t =
  match l with
  | [] -> ret []
  | h :: t ->
    let* h' = f h in let* l' = mmap f l in ret (h' :: l')
```

(let* x = a in b expands to bind a (fun x -> b).)
In the non-determinism monad: all the ways to insert an element $x$ in a list $l$.

```ocaml
let rec insert (x: 'a) (l: 'a list) : 'a list t =
  choose (ret (x :: l))
  (match l with
   | [] -> fail
   | h :: t -> let* t' = insert x t in ret (h :: t'))
```

All the permutations of a list $l$.

```ocaml
let rec permut (l: 'a list) : 'a list t =
  match l with
  | [] -> ret []
  | h :: t -> let* t' = permut t in insert h t'
```
Free monads
and interaction trees
Executing a monadic program without performing the effects

Consider mutable state and non-determinism.

Values:

\[ v ::= \text{cst} \mid x \mid \lambda x. \ M \]

Computations:

\[ M ::= v_1 \ v_2 \mid \text{if } v \ \text{then } M \ \text{else } N \]
\[ \mid \text{val } v \mid \text{do } x \leftarrow M_1 \ \text{in } M_2 \]
\[ \mid \text{get } \ell \mid \text{set } \ell \ v \]
\[ \mid \text{choose } M_1 \ M_2 \mid \text{fail} \]

Can we evaluate the \texttt{do}, the function calls, and the conditionals while leaving the effects uninterpreted?
We define a type of intermediate evaluation results, representing all possible sequences of program effects.

\[
R A = \text{Pure : } A \to R A \\
\mid \text{Get : } \text{Loc} \to (\text{Val} \to R A) \to R A \\
\mid \text{Set : } \text{Loc} \to \text{Val} \to R A \to R A \\
\mid \text{Choose : } R A \to R A \to R A \\
\mid \text{Fail : } R A
\]
A tree-shaped representation of effects

Program:

```
choose (do _ ⇐ set ℓ 0 in do x ⇐ get ℓ in val (x + 1))
(choose (val 0) fail)
```

Intermediate result:
The monad of intermediate results

\[ R A \quad = \quad \text{Pure} : A \rightarrow R A \]
\[ \quad | \quad \text{Get} : \text{Loc} \rightarrow (\text{Val} \rightarrow R A) \rightarrow R A \]
\[ \quad | \quad \text{Set} : \text{Loc} \rightarrow \text{Val} \rightarrow R A \rightarrow R A \]
\[ \quad | \quad \text{Choose} : R A \rightarrow R A \rightarrow R A \]
\[ \quad | \quad \text{Fail} : R A \]

This type is a monad, with \( \text{ret} \overset{\text{def}}{=} \text{Pure} \) and \( \text{bind} \) defined as:

\[
\text{bind} (\text{Pure} \ v) \ f = f \ v \\
\text{bind} (\text{Get} \ \ell \ k) \ f = \text{Get} \ \ell (\lambda v. \text{bind} (k \ \ell) \ f) \\
\text{bind} (\text{Set} \ \ell \ v \ R) \ f = \text{Set} \ \ell \ v (\text{bind} R \ f) \\
\text{bind} (\text{Choose} \ R_1 \ R_2) \ f = \text{Choose} (\text{bind} R_1 \ f) (\text{bind} R_2 \ f) \\
\text{bind} \text{Fail} \ f = \text{Fail}
\]
A denotational semantics for monadic programs

Using this monad of results, we can compute the intermediate result $[M]$ of a monadic computation $M$.

$$[v_1 \ v_2] = [v_1]_v \ [v_2]_v$$

$$[\text{if } v \text{ then } M_1 \text{ else } M_2] = \text{if } [v]_v \text{ then } [M_1] \text{ else } [M_2]$$

$$[\text{val } v] = \text{Pure } [v]_v$$

$$[\text{do } x \leftarrow M_1 \text{ in } M_2] = \text{bind } [M_1] \ (\lambda x. \ [M_2])$$

$$[\text{get } \ell] = \text{Get } \ell \ (\lambda v. \text{Pure } v)$$

$$[\text{set } \ell \ v] = \text{Set } \ell \ [v]_v \ (\text{Pure } ())$$

$$[\text{choose } M_1 \ M_2] = \text{Choose } [M_1] \ [M_2]$$

$$[\text{fail}] = \text{Fail}$$

Where $[\text{cst}]_v = \text{cst}$, $[x]_v = x$, $[\lambda x. M]_v = \lambda x. \ [M]$. 
Finally, we can interpret effects (function \texttt{run}) using a \textit{fold} traversal of the result tree \( R \).

With backtracking of the store at choice points: \texttt{run} has type \( R \ A \to Store \to Set \ A \) and we take

\[
\begin{align*}
\text{run (Pure } v\text{) } s &= \{ v \} \\
\text{run (Get } \ell \text{ } k\text{) } s &= \text{run } (k (s \ell)) \ s \\
\text{run (Set } \ell \text{ } v \text{ } R\text{) } s &= \text{run } R \ (s\{\ell \leftarrow v\}) \\
\text{run Fail } s &= \emptyset \\
\text{run (Choose } R_1 \text{ } R_2\text{) } s &= \text{run } R_1 \ s \cup \text{run } R_2 \ s
\end{align*}
\]
Interpreting effects

Finally, we can interpret effects (function run) using a \textit{fold} traversal of the result tree $R$.

With a store that persists across choice points:
run has type $R A \rightarrow \text{Store} \rightarrow \text{Set} A \times \text{Store}$ and we take

\[
\begin{align*}
\text{run} \ (\text{Pure} \ v) \ s & = (\{v\}, s) \\
\text{run} \ (\text{Get} \ \ell \ k) \ s & = \text{run} \ (k \ (s \ \ell)) \ s \\
\text{run} \ (\text{Set} \ \ell \ v \ R) \ s & = \text{run} \ R \ (s\{\ell \leftarrow v\}) \\
\text{run} \ \text{Fail} \ s & = (\emptyset, s) \\
\text{run} \ (\text{Choose} \ R_1 \ R_2) \ s & = (V_1 \cup V_2, s_2)
\end{align*}
\]

with \text{run} \ R_1 \ s = (V_1, s_1) \text{ and } \text{run} \ R_2 \ s_1 = (V_2, s_2)
The type $R\ A$ is an instance of a more general categorical construction: the **free monad**.

$$R\ A = \text{Pure} : A \rightarrow R\ A$$

$$| \quad \text{Op} : F (R\ A) \rightarrow R\ A$$

where $F : Type \rightarrow Type$ is a **functor**: it comes with an operation

$$\text{fmap} : \forall A, B, (A \rightarrow B) \rightarrow (F\ A \rightarrow F\ B)$$

We recover the previous example by defining $F$ as

$$F\ X = \text{Get} : Loc \rightarrow (Val \rightarrow X) \rightarrow F\ X$$

$$| \quad \text{Set} : Loc \rightarrow Val \rightarrow X \rightarrow F\ X$$

$$| \quad \text{Choose} : X \rightarrow X \rightarrow F\ X$$

$$| \quad \text{Fail} : F\ X$$

**Exercise**: define $\text{fmap}$. 
The free monad

\[ R A = \text{Pure} : A \to R A \]
\[ | \quad \text{Op} : F (R A) \to R A \]

This “functorial” presentation makes it possible to define \( \text{ret} \) and \( \text{bind} \) in a generic way:

\[
\begin{align*}
\text{ret } \nu & = \text{Pure } \nu \\
\text{bind } (\text{Pure } \nu ) f & = f \nu \\
\text{bind } (\text{Op } \varphi ) f & = \text{Op } (\text{fmap } (\lambda x. \text{bind } x f) \varphi)
\end{align*}
\]
The freer monad

(O. Kiselyov, H. Ishii, Freer Monads, More Extensible Effects, 2015.)

Another generic construction of the type of intermediate execution results.

\[ RA = \text{Pure} : A \rightarrow RA \]

\[ | \text{Op} : \forall B, \text{Eff} B \rightarrow (B \rightarrow RA) \rightarrow RA \]

\( \text{Eff} B \) is the type of effects producing a result of type \( B \). Each specific effect is a constructor of type \( \text{Eff} \).

If \( \varphi : \text{Eff} B \), the subtrees of \( \text{Op}(\varphi, k) \) are \( k b \) for \( b : B \). There are as many subtrees as there are elements in \( B \).
Declaring and typing effects

For mutable state and non-determinism:

\begin{align*}
\text{Get} & : \text{Loc} \rightarrow \text{Eff Val} & \text{(one subtree per possible value)} \\
\text{Set} & : \text{Loc} \rightarrow \text{Val} \rightarrow \text{Eff unit} & \text{(one subtree)} \\
\text{Fail} & : \text{Eff empty} & \text{(no subtree)} \\
\text{Flip} & : \text{Eff bool} & \text{(two subtrees)}
\end{align*}

We encode the \texttt{choose} operation using the \texttt{Flip} effect:

\[
\text{choose } R_1 \ R_2 \overset{\text{def}}{=} \text{Op}(\text{Flip}, \lambda b. \text{if } b \text{ then } R_1 \text{ else } R_2)
\]
The freer monad

\[ R A = \text{Pure} : A \rightarrow R A \]
\[ \mid \quad \text{Op} : \forall B, \text{Eff} B \rightarrow (B \rightarrow R A) \rightarrow R A \]

This presentation “indexed by type B” also makes it possible to define \textit{ret} and \textit{bind} generically:

\[
\text{ret } v = \text{Pure } v \\
\text{bind } (\text{Pure } v) f = f v \\
\text{bind } (\text{Op } \varphi k) f = \text{Op } (\varphi, \lambda x. \text{bind } (k x) f)
\]

We no longer need a functor nor a \textit{fmap}.
Interpreting effects in the freer monad

Using a generic \textit{fold} over the type of results:

\[
\text{run} : (A \to B) \to (\forall C, \text{Eff } C \to (C \to B) \to B) \to R A \to B
\]

\[
\text{run } f \ g \ (\text{Pure } v) = f \ v
\]

\[
\text{run } f \ g \ (\text{Op } \varphi \ k) = g \varphi \ (\lambda x. \text{run } f \ g \ (k \ x))
\]

For non-determinism with backtracking of state, we take

\[
f : A \to \text{Store} \to \text{Set } A
\]

\[
f \ x \ s = \{x\}
\]

\[
g : \text{Eff } B \to (B \to \text{Store} \to \text{Set } A) \to \text{Store} \to \text{Set } A
\]

\[
g \ (\text{Get } \ell) \ k \ s = k \ (s \ \ell) \ s
\]

\[
g \ (\text{Set } \ell \ v) \ k \ s = k \ () \ s\{\ell \leftarrow v\}
\]

\[
g \ \text{Flip} \ k \ s = k \ false \ s \cup k \ true \ s
\]

\[
g \ \text{Fail} \ k \ s = \emptyset
\]
Interpreting effects in the freer monad

Using a generic \textit{fold} over the type of results:

\[
\text{run} : (A \to B) \to (\forall C, \text{Eff } C \to (C \to B) \to B) \to R A \to B
\]

\[
\text{run } f \ g \ (\text{Pure } v) = f \ v
\]

\[
\text{run } f \ g \ (\text{Op } \varphi \ k) = g \ \varphi \ (\lambda x. \ \text{run } f \ g \ (k \ x))
\]

Note the \textit{control inversion}: it’s no longer the program that calls the \texttt{get}, \texttt{set}, \ldots operations of the monad; it’s the implementation of these operations (the \textit{g} function) that evaluates the program “on demand” using the continuation \textit{k}.
Interaction trees


A coinductive version of the type of intermediate results, able to account for diverging computations:

\[ R\ A \ = \ \text{Pure} : A \rightarrow R\ A \]
\[ \quad | \quad \text{Op} : \forall B, \text{Eff } B \rightarrow (B \rightarrow R\ A) \rightarrow R\ A \]
\[ \quad | \quad \text{Tau} : R\ A \rightarrow R\ A \]

\text{Tau} denotes one step of computation without effects.

The infinite tree \( \bot \) def = \text{Tau} \( \bot \) = \text{Tau}(\text{Tau}(\text{Tau}(\ldots))) \) represents a computation that diverges without observable effects.

The infinite tree \( x \) def = \text{Op}(\text{Flip}, \lambda b. \text{if } b \text{ then Pure 0 else } x) \) represents let rec \( f () = \text{choose 0 (f ())} \).
Reminders on algebraic structures
Algebraic structures

An algebraic structure =

- a set (or a type), called the **carrier** of the structure;
- **operations** over this set;
- **equations** (laws) that these operations satisfy.

**Example:** a monoid is \((T, \varepsilon, \cdot)\) where

\[
\begin{align*}
\varepsilon : T & \quad \text{idemtityl element} \\
\cdot : T \to T \to T & \quad \text{composition} \\
\varepsilon \cdot x &= x & \quad \text{left identity} \\
x \cdot \varepsilon &= x & \quad \text{right identity} \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z) & \quad \text{associativity}
\end{align*}
\]
An algebraic structure =

- a set (or a type), called the *carrier* of the structure;
- *operations* over this set;
- *equations* (laws) that these operations satisfy.

**Example:** a group is \((T, 0, +, −)\) where

\[
0 : T \quad \text{identity element}
\]
\[
+ : T \to T \to T \quad \text{composition}
\]
\[
− : T \to T \quad \text{inverse}
\]
\[
0 + x = x + 0 = x \quad \text{identity}
\]
\[
(x + y) + z = x + (y + z) \quad \text{associativity}
\]
\[
(−x) + x = x + (−x) = 0 \quad \text{inverse}
\]
A **theory**:
the signature of operators (names and types)
+ the equations.

A **model of the theory**: a definition of the support and of the operations that satisfies the equations.

Examples of models for the theory of monoids
(or just: “examples of monoids”):

\[(\mathbb{N}, 0, +) \quad (\mathbb{R}, 1, \times) \quad (T \rightarrow T, id, \circ)\]

Examples of models for the theory of groups
(or just: “examples of groups”):

\[(\mathbb{Z}, 0, +, \text{–}) \quad (\mathbb{R}^*, 1, \times, ^{-1})\]
An algebraic abstract type is the specification of a persistent data structure as a signature and equations.

(→ 2022–2023 course, lecture #1)

Example: stacks

empty : S  
push : E → S → S  
top : S → E  
pop : S → S

top(push v s) = v  
pop(push v s) = s

It becomes a queue (FIFO) if we add one operation:

add : S → E → S

add empty v = push v empty  
add (push w s) v = push w (add s v)
The free monoid

Given a set (an “alphabet”) $A$, the **free monoid over $A$** is $(A^*, \varepsilon, \cdot)$, where

- **support**: $A^*$ the set of finite lists of $A$ (“words over $A$”) such as $a_1a_2\ldots a_n$;
- **identity element** $\varepsilon$: the empty list;
- **composition** $\cdot$: list concatenation.

**Example**: taking $A = \{1, \ldots, 9\}$,

$$1 \cdot (23 \cdot 456) = (1 \cdot 23) \cdot 456 = 123456$$
The free monoid

The free monoid over \( A \) is “the simplest” or “the least constrained” among all monoids whose support contains \( A \).

Indeed, if \((B, 0, +)\) is a monoid, with \( A \subseteq B \), we can define a function \( \Phi : A^* \to B \) as

\[
\Phi(a_1 \ldots a_n) = 0 + a_1 + \cdots + a_n
\]

(It’s the “fold” of “+” over the list \( a_1 \ldots a_n \).)

This function is a morphism from \((A^*, \varepsilon, \cdot)\) to \((B, 0, +)\), since it commutes with monoid operations:

\[
\Phi(\varepsilon) = 0 \quad \Phi(\ell_1 \cdot \ell_2) = \Phi(\ell_1) + \Phi(\ell_2)
\]
Free models

Let $T$ be an algebraic theory and $X$ a set.

A free $T$-model generated by $X$ is a $T$-model $M$ and a function $f : X \to \text{supp}(M)$ such that:

For every other $T$-model $M'$ and function $f' : X \to \text{supp}(M')$, there exists a unique morphism $\Phi : M \to M'$ such that the following diagram commutes:

$$
\begin{align*}
X & \xrightarrow{f} \text{supp}(M) \\
& \downarrow f' \quad \downarrow \Phi \\
& \text{supp}(M')
\end{align*}
$$
Monads viewed as algebraic structures

A monad can be presented as an algebraic structure whose operations are \( \text{ret} \), \( \text{bind} \), and \( \text{op}(F) \) for each constructor \( F \) of type \( \text{Eff} \), with the following signatures:

\[
\begin{align*}
\text{ret} : & \ A \to T A \\
\text{bind} : & \ T A \to (A \to T B) \to T B \\
\text{op}(F) : & \ P \to (B \to T A) \to T A \quad \text{if } F : P \to \text{Eff} B
\end{align*}
\]

The equations are the three monadic laws, plus other laws for the \( \text{op}(F) \) operations.
Free monads are free!

The free monad and the freer monad are free monads generated by the constructors of type \( \text{Eff} \).

Let's check this fact for the freer monad.

\[
R A = \text{Pure} : A \rightarrow R A
\]

\[
| \text{Op} : \forall B, \text{Eff } B \rightarrow (B \rightarrow R A) \rightarrow R A
\]

The associated monad structure, with the expected signature:

\[
\text{ret } x = \text{Pure } x
\]

\[
\text{bind } (\text{Pure } x) f = f x
\]

\[
\text{bind } (\text{Op}(\varphi, k)) f = \text{Op}(\varphi, \lambda x. \text{bind } (k x) f)
\]

\[
\text{op}(F) = \lambda x. \text{Op}(F x, \lambda y. \text{Pure } y)
\]
Free monads are free!

Let \( M = (T, \text{ret}_M, \text{bind}_M, \text{op}_M(F)) \) another monad with the expected signature. We define a morphism \( \Phi \) from the freer monad to \( M \) by

\[
\Phi : R A \rightarrow T A
\]

\[
\Phi(\text{Pure } v) = \text{ret}_M v
\]

\[
\Phi(\text{op} (F x, k)) = \text{bind}_M (\text{op}_M(F) x) (\lambda y. \Phi(k y))
\]

This morphism commutes with operations \( \text{ret} \) and \( \text{bind} \).

\[
\Phi(\text{bind} (\text{Pure } v) f) = \Phi(f v) = \text{bind}_M (\Phi(\text{Pure } v)) (\lambda y. \Phi(f y)) \quad (1\text{st law})
\]

\[
\Phi(\text{bind} (\text{op}(F x, k)) f) = \Phi(\text{op}(F x, \lambda y. \text{bind}(k y) f))
\]

\[
= \text{bind}_M (\text{op}_M(F) x) (\lambda y. \Phi(\text{bind}(k y) f))
\]

\[
= \text{bind}_M (\text{bind}_M (\text{op}_M(F) x) (\lambda y. \Phi(k y))) (\lambda z. \Phi(f z)) \quad (3\text{rd law})
\]

\[
= \text{bind}_M (\Phi(\text{op}(F x, k))) (\lambda z. \Phi(f z))
\]
Algebraic effects
Moggi’s computational lambda-calculus, and more generally the monadic approach, specify the propagation and sequencing of effects in a generic manner.

How to specify the generation of effects by the specific operations of the monad? (set, get, choose, fail, ...)

Plotkin and Power (2003) propose to specify these operations by equations, therefore obtaining an algebraic structure for these effects.
A computational lambda-calculus with effects

Values: \( v ::= x \mid cst \mid \lambda x. M \)

Computations: \( M, N ::= v v' \) application
\( \mid \text{if } v \text{ then } M \text{ else } N \) conditional
\( \mid \text{val } v \) trivial computation
\( \mid \text{do } x \leftarrow M \text{ in } N \) sequencing
\( \mid F(\vec{v}; y. M) \) effectful operation

\( F(v_1 \ldots v_n; y. M) \) represents an operation that produces an effect. The values \( v_i \) are the arguments of this operation. The operation produces a result value that is bound to \( y \) in continuation \( M \).

Notation: \( F(\vec{v}) \overset{\text{def}}{=} F(\vec{v}; y. \text{val}(y)) \) (trivial continuation).
The laws of the computational lambda-calculus with effects

Same laws as for the computational lambda-calculus:

\[(\lambda x. M) v = M\{x \leftarrow v}\]
\[
\text{do } x \leftarrow \text{val } v \text{ in } M = M\{x \leftarrow x\}
\]
\[
\text{do } x \leftarrow M \text{ in } \text{val } x = M
\]
\[
\text{do } x \leftarrow (\text{do } y \leftarrow M \text{ in } N) \text{ in } P = \text{do } y \leftarrow M \text{ in } \text{do } x \leftarrow N \text{ in } P
\]

Plus commutation between do and effectful operations:

\[
\text{do } x \leftarrow F(\vec{v}; y. M) \text{ in } N = F(\vec{v}; y. \text{do } x \leftarrow M \text{ in } N)
\]

Plus laws specific to some effects.
Laws for mutable state

The “good variable” properties (read after write):

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

Other commutations between accesses to different locations:

\[
\text{get}(\ell; y. \text{get}(\ell'; z. M)) = \text{get}(\ell'; z. \text{get}(\ell; y. M)) \\
\text{set}(\ell, v; y. \text{set}(\ell', v'; z. M)) = \text{set}(\ell', v'; z. \text{set}(\ell, v; y. M)) \quad \text{if } \ell' \neq \ell
\]

Other commutations between accesses to the same location:

\[
\text{get}(\ell; y. \text{get}(\ell; z. M)) = \text{get}(\ell; y. M\{z \leftarrow y\}) \quad \text{(double read)} \\
\text{get}(\ell; y. \text{set}(\ell, y; \_ \cdot M)) = M \quad \text{(read then rewrite)} \\
\text{set}(\ell, v_1; \_ \cdot \text{set}(\ell, v_2; \_ \cdot M)) = \text{set}(\ell, v_2; \_ \cdot M) \quad \text{(double write)}
\]
Laws for non-determinism

For failure:

\[ \text{Fail}(; k) = \text{Fail}(; k') = \text{Fail()} \]  
(propagation)

For choice:

\[ \text{choose } M M = M \]  
(idempotent)

\[ \text{choose } M N = \text{choose } N M \]  
(commutative)

\[ \text{choose } (\text{choose } M N) P = \text{choose } M (\text{choose } N P) \]  
(associative)

\[ \text{choose } \text{Fail}() M = \text{choose } M \text{Fail}() = M \]  
(identity)

Less natural to express with the encoding

\[ \text{choose } M N = \text{Flip}(; \lambda b. \text{if } b \text{ then } M \text{ else } N) \]
A semantic for computational lambda-calculus with effects

To every computation we associate an interaction tree / a term of the freer monad.

\[
[v_1 \ v_2] = [v_1]_v \ [v_2]_v \quad \text{or} \quad \text{Tau}([v_1]_v \ [v_2]_v)
\]

\[
[val \ v] = \text{Pure} [v]_v
\]

\[
[\text{do } x \leftarrow M_1 \ \text{in} \ M_2] = \text{bind} [M_1] (\lambda x. [M_2])
\]

\[
[F(\vec{v} ; y. \ M)] = \text{Op} (F \ \vec{v}) (\lambda y. [M])
\]

We can then interpret effects by the appropriate “fold”:

\[
\text{fold} : (A \to B) \to (\forall C, \text{Eff} C \to (C \to B) \to B) \to R A \to B
\]

\[
\text{fold} f g (\text{Pure} \ v) = f \ v
\]

\[
\text{fold} f g (\text{Op} \ \varphi \ k) = g \ \varphi (\lambda x. \text{fold} f g (k \ x))
\]
A “fold” can rebuild an interaction tree instead of producing the final result of the execution. This enables the fold to handle a subset of the effects and to re-emit the other effects.

Example: a handler for the Get and Set effects.

\[
\text{state}: RA \rightarrow Store \rightarrow RA = \text{fold } f_{\text{state}} g_{\text{state}}
\]

\[
f_{\text{state}} v = \lambda s. \text{Pure } v
\]

\[
g_{\text{state}} (\text{Get } \ell) k = \lambda s. k (s \ell) s
\]

\[
g_{\text{state}} (\text{Set } \ell v) k = \lambda s. k () s\{\ell \leftarrow v\}
\]

\[
g_{\text{state}} \varphi k = \lambda s. \text{Op}(\varphi, \lambda x. k x s) \text{ for all other } \varphi
\]
Example: a handler for the Flip and Fail effects.

\[
\text{nondet : } R A \rightarrow R (\text{Set } A) = \text{fold } f_{\text{nondet}} g_{\text{nondet}} \\
f_{\text{nondet}} v = \text{Pure } \{v\} \\
g_{\text{nondet}} \text{Fail } k = \text{Pure } \emptyset \\
g_{\text{nondet}} \text{Flip } k = \text{bind } (k \text{ true}) (\lambda x_1. \text{bind } (k \text{ false}) (\lambda x_2. \text{Pure } (x_1 \cup x_2))) \\
g_{\text{nondet}} \varphi k = \text{Op}(\varphi, k) \quad \text{for all other } \varphi
\]
The composition \( \text{nondet} \ (\text{state} \ t \ s_0) \) implements the semantics where the store is backtracked at choice points.

The composition \( \text{state} \ (\text{nondet} \ t) \ s_0 \) implements the semantics where the store persists across choice points.

If the tree \( t \) contains no other effect besides Get, Set, Fail and Flip, the two compositions produce a trivial tree \( \text{Pure} \ v \) where \( v \) is the final value of the program.
Implementing equations correctly

The equations related to \texttt{bind} are automatically satisfied by the semantics based on interaction trees.

The other equations must be satisfied by the handlers that interpret the effects.
After conversion to interaction trees and simplification by the state handler, the 7 laws for mutable state follow from the following 5 equalities (the two get-get laws are trivial):

\[
\begin{align*}
s\{\ell \leftarrow v\} \ell &= v \\
s\{\ell \leftarrow v\} \ell' &= s \ell' \quad \text{if } \ell' \neq \ell \\
s\{\ell \leftarrow v\}\{\ell \leftarrow v'\} &= s\{\ell \leftarrow v'\} \\
s\{\ell \leftarrow v\}\{\ell' \leftarrow v'\} &= s\{\ell' \leftarrow v'\}\{\ell \leftarrow v\} \quad \text{if } \ell' \neq \ell \\
s\{\ell \leftarrow s \ell\} &= s
\end{align*}
\]

*Exercise:* show that \texttt{nondet} satisfies the laws for non-determinism.
Programming one’s effect handlers

We extend the computational lambda-calculus with a construct to define effect handlers within the language.

Values: \( v ::= x \mid \text{cst} \mid \lambda x. M \)

Computations: \( M, N ::= v \; v' \)

| if \( v \) then \( M \) else \( N \)  
| \( \text{val} \; v \)  
| \( \text{do} \; x \leftarrow M \; \text{in} \; N \)  
| \( F(\vec{v}; y. M) \) [effectful operation]  
| \( \text{with} \; H \; \text{handle} \; M \) [effect handler]  

Handlers: \( H ::= \{ \text{val}(x) \rightarrow M_{val}; \quad F_1(\vec{x}; k) \rightarrow M_1; \quad \ldots \quad F_n(\vec{x}; k) \rightarrow M_n \} \)
Intuitive semantics for effect handlers

with \{ \text{val}(x) \rightarrow M_{\text{val}} ; \ldots ; F_i(\vec{x}; k) \rightarrow M_i ; \ldots \} \text{ handle } M \]

If \( M \) terminates with value \( v \), the \( M_{\text{val}} \) case is evaluated with \( x = v \).

If \( M \) performs the effect \( F_i(\vec{v}; y. N) \), the \( M_i \) case is evaluated with \( \vec{x} = \vec{v} \) and

\[
k = \lambda y. N \quad \text{or} \quad k = \lambda y. \text{with } \{ \ldots \} \text{ handle } N.
\]

(\textit{shallow handler}) \quad (\textit{deep handler})

If \( M \) performs another effect \( F(\vec{v}; y. N) \), with \( F \notin \{ F_1, \ldots, F_n \} \), we perform the effect \( F(\vec{v}; y. N) \) or \( F(\vec{v}; y. \text{with } \{ \ldots \} \text{ handle } N) \).

(\textit{shallow handler}) \quad (\textit{deep handler})
The denotation $[H]$ of an effect handler is an interaction tree transformer, so that

$$[\text{with } H \text{ handle } M] = [H] [M]$$

This transformer is a “fold” for a deep handler and a case analysis for a shallow handler:

$$[H] = \begin{cases} 
\text{fold } [H]_{\text{ret}} [H]_{\text{eff}} & \text{(deep handler)} \\
\text{case } [H]_{\text{ret}} [H]_{\text{eff}} & \text{(shallow handler)} 
\end{cases}$$

fold and case are defined as

$$\text{fold } f g (\text{Pure } v) = f v \quad \text{case } f g (\text{Pure } v) = f v$$

$$\text{fold } f g (\text{Op}(\varphi, k)) = g \varphi (\lambda x. \text{fold } f g (k x)) \quad \text{case } f g (\text{Op}(\varphi, k)) = g \varphi k$$
Denotational semantics of effect handlers

\[ H = \{ \text{val}(x) \rightarrow M_{\text{val}} ; F_1(\vec{x} ; k) \rightarrow M_1 ; \ldots ; F_n(\vec{x} ; k) \rightarrow M_n \} \]

We define the semantics for normal return and for return on an effect:

\[
\llbracket H \rrbracket_{\text{ret}} x = \llbracket M_{\text{val}} \rrbracket
\]

\[
\llbracket H \rrbracket_{\text{eff}} (F_i \vec{x}) k = \llbracket M_i \rrbracket
\]

\[
\llbracket H \rrbracket_{\text{eff}} (F \vec{x}) k = \text{Op} (F \vec{x}) k \quad \text{if } F \notin \{F_1, \ldots, F_n\}
\]
Summary
Two steps towards a general theory of effects in programming languages.

- **Monads:**
  - “Clean up” denotational semantics and transformation of functional programs.
  - Programming in monadic style generalizes programming in CPS, and makes it possible to use effects that are not supported natively by the programming language.

- **Algebraic effects:**
  - Specifying effects by equations.
  - Implementing effects by effect handlers, which are “folds” or “cases” on interaction trees.
  - Effect handlers can be defined within the programming language.
References
References

An introduction to monadic programming:


An introduction to effects and effect handlers:

• Matija Pretnar: *An Introduction to Algebraic Effects and Handlers*, ENTCS 319, 2015. [https://doi.org/10.1016/j.entcs.2015.12.003](https://doi.org/10.1016/j.entcs.2015.12.003)

The freer monad:

• Oleg Kiselyov, Hiromi Ishii: *Freer monads, more extensible effects*. Haskell 2015: 94-105. [https://doi.org/10.1145/2804302.2804319](https://doi.org/10.1145/2804302.2804319)

The algebraic viewpoint: