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:

$$[\![stmt]\!] : Store \to Store_\perp \qquad \text{(mutable state)}$$

$$[\![stmt]\!] : Env \to Store \to Store_\perp \qquad \text{(environment + state)}$$

$$[\![stmt]\!] : Env \to Store \to (Store \to Res_\perp) \to Res_\perp$$

(environment + state + goto)

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

$$[\![s_1; s_2]\!]\,\sigma = [\![s_2]\!]\,([\![s_1]\!]\,\sigma)$$

$$[\![s_1; s_2]\!]\,\rho\,\sigma = [\![s_2]\!]\,\rho\,([\![s_1]\!]\,\rho\,\sigma)$$

$$[\![s_1; s_2]\!]\,\rho\,\sigma\,k = [\![s_1]\!]\,\rho\,\sigma\,(\lambda\sigma'.\,[\![s_2]\!]\,\rho\,\sigma'\,k)$$

A proliferation of program transformations

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 λ-abstractions:

$$
\begin{aligned}
\mathcal{C}(cst) &= \lambda k.\ k\ cst & \mathcal{C}(\lambda x.\, M) &= \lambda k.\ k\ (\lambda x.\, \mathcal{C}(M)) \\
\mathcal{C}^2(cst) &= \lambda k_1 k_2.\ k_1\ cst & \mathcal{C}^2(\lambda x.\, M) &= \lambda k_1 k_2.\ k_1\ (\lambda x.\, \mathcal{C}^2(M)) \\
\mathcal{E}(cst) &= V\ cst & \mathcal{E}(\lambda x.\, M) &= V\ (\lambda x.\, \mathcal{E}(M))
\end{aligned}
$$

In all cases, we return a value (cst or $\lambda x \ldots$)
presenting it as a trivial computation.

Commonalities between these transformations

For `let` bindings:

$$\mathcal{C}(\texttt{let}\, x = e_1 \,\texttt{in}\, e_2) = \lambda k.\, \mathcal{C}(e_1)\, (\lambda x.\, \mathcal{C}(e_2)\, k)$$
$$\mathcal{C}^2(\texttt{let}\, x = e_1 \,\texttt{in}\, e_2) = \lambda k_1 k_2.\, \mathcal{C}^2(e_1)\, (\lambda x.\, \mathcal{C}^2(e_2)\, k_1\, k_2)\, k_2$$
$$\mathcal{E}(\texttt{let}\, x = e_1 \,\texttt{in}\, e_2) = \texttt{match}\, \mathcal{E}(e_1)\, \texttt{with}\, E\, x \to E\, x \mid V\, x \to \mathcal{E}(e_2)$$

In the three transformations, we perform the computation 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:

$$\begin{aligned}
\mathcal{C}(e_1\ e_2) &= \lambda k.\, \mathcal{C}(e_1)\ (\lambda v_1.\, \mathcal{C}(e_2)\ (\lambda v_2.\, v_1\ v_2\ k)) \\
\mathcal{C}^2(e_1\ e_2) &= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ (\lambda v_1.\, \mathcal{C}^2(e_2)\ (\lambda v_2.\, v_1\ v_2\ k_1\ k_2)\ k_2)\ k_2 \\
\mathcal{E}(e_1\ e_2) &= \texttt{match}\ \ \mathcal{E}(e_1)\ \ \texttt{with}\ \ E\,x_1 \to E\,x_1 \mid V\,v_1 \to \\
&\qquad \texttt{match}\ \ \mathcal{E}(e_2)\ \ \texttt{with}\ \ E\,x_2 \to E\,x_2 \mid V\,v_2 \to v_1\ v_2
\end{aligned}$$

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

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

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)

The computational lambda-calculus

Different choices for T correspond to known denotational semantics / program transformations for different effects:

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

Base operations over computations

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

- $\texttt{ret} : A \to T\,A$ (injection)

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

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

  $\texttt{bind}\ \ a\ (\lambda x.b)$ executes the computation a, bind its result value to x, then executes the computation b, 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, η, μ) where

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

satisfying certain laws.

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

$$\texttt{ret}\ v \stackrel{def}{=} \eta(v)$$

$$\texttt{bind}\ a\,f \stackrel{def}{=} \mu(T(f)\ a)$$

(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)

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

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

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

Non-determinism as a monad

$$T\,A = \mathcal{P}(A) \qquad \text{(or } \mathit{List}(A)\text{)}$$

$$\mathtt{ret}\ \ v = \{v\}$$

$$\mathtt{bind}\ \ a\,f = \bigcup_{x \in a} f\,x$$

Operations specific to non-determinism:

$$\mathtt{fail} = \emptyset$$

$$\mathtt{choose}\ \ a\,b = a \cup b$$

Exceptions as a monad

$$T\,A = V \;\texttt{of}\; A \mid E \;\texttt{of}\; \mathit{Exn} \qquad (\approx A + \mathit{Exn})$$

$$\begin{aligned}
\texttt{ret}\;\; v &= V\,v \\
\texttt{bind}\;\; (V\,v)\,f &= f\,v \\
\texttt{bind}\;\; (E\,e)\,f &= E\,e \qquad\qquad \text{(exception propagation)}
\end{aligned}$$

Operations specific to exceptions:

$$\begin{aligned}
\texttt{raise}\,e &= E\,e \\
\texttt{try}\;\; a \;\;\texttt{with}\;\; x \to b &= \texttt{match}\;\; a \;\;\texttt{with}\;\; V\,v \to V\,v \mid E\,x \to b
\end{aligned}$$

Mutable state as a monad

$$T\,A = S \to A \times S$$ (*S* = type of states)

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

Specific operations: (ℓ = reference identifier)

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

Continuations as a monad

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

$$\begin{aligned} \texttt{ret}\ \ v &= \lambda k.\ k\ v \\ \texttt{bind}\ \ a\,f &= \lambda k.\ a\ (\lambda x.\ f\ x\ k) \end{aligned}$$

Control operator:

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

Monads that combine several effects

State + exceptions:

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

State + continuations:

$$T\,A = S \to (A \to S \to R) \to R$$

Continuations + exceptions:

$$T\,A = ((A + E) \to R) \to R$$

$$\text{or} \quad T\,A = (A \to R) \to (E \to R) \to R$$

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

A computational lambda-calculus

"Values are; computations do."

Values:

$$v \quad ::= cst \mid x \mid \lambda x.\, M$$

Computations:

$M, N ::= v_1\ v_2$	application
$\mid \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N$	conditional
$\mid \texttt{val}\ v$	trivial computation
$\mid \texttt{do}\ x \Leftarrow M\ \texttt{in}\ N$	sequencing of computations
$\mid \ldots$	monad-specific operations

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

The laws of the computational lambda-calculus

Function application:

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

The three monadic laws:

$$\begin{aligned}
\mathtt{do}\; x \Leftarrow \mathtt{val}\; v \;\mathtt{in}\; M &= M\{x \leftarrow v\} \\
\mathtt{do}\; x \Leftarrow M \;\mathtt{in}\; \mathtt{val}\; x &= M \\
\mathtt{do}\; x \Leftarrow (\mathtt{do}\; y \Leftarrow M \;\mathtt{in}\; N) \;\mathtt{in}\; P &= \mathtt{do}\; y \Leftarrow M \;\mathtt{in}\; (\mathtt{do}\; x \Leftarrow N \;\mathtt{in}\; P)
\end{aligned}$$

The monadic transformation

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

$$
\begin{aligned}
\mathcal{M}(cst) &= \texttt{val}\ cst \\
\mathcal{M}(\lambda x.\, e) &= \texttt{val}\ (\lambda x.\, \mathcal{M}(e)) \\
\mathcal{M}(x) &= \texttt{val}\ x \\
\mathcal{M}(\texttt{let}\ x = e_1\ \ \texttt{in}\ \ e_2) &= \texttt{do}\ x \Leftarrow \mathcal{M}(e_1)\ \texttt{in}\ \mathcal{M}(e_2) \\
\mathcal{M}(e_1\ e_2) &= \texttt{do}\ f \Leftarrow \mathcal{M}(e_1)\ \texttt{in}\ \texttt{do}\ v \Leftarrow \mathcal{M}(e_2)\ \texttt{in}\ f\ v \\
\mathcal{M}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \texttt{do}\ b \Leftarrow \mathcal{M}(e_1)\ \texttt{in} \\
&\qquad \texttt{if}\ b\ \texttt{then}\ \mathcal{M}(e_2)\ \texttt{else}\ \mathcal{M}(e_3)
\end{aligned}
$$

By combining this transformation with the appropriate monads, we recover the CPS / ERS / double-barreled CPS transformations.

Programming directly in monadic style

(Notations `do` in Haskell, `let*` in OCaml.)

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

```
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` $x \rightarrow b$) .)

Programming directly in monadic style

In the non-determinism monad:
all the ways to insert an element `x` in a list `l`.

```
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`.

```
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:

$$v \ ::= cst \mid x \mid \lambda x.\, M$$

Computations:

$$\begin{aligned} M ::= \;& v_1\, v_2 \mid \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N \\ & \mid \texttt{val}\ \ v \mid \texttt{do}\ x \Leftarrow M_1\ \texttt{in}\ M_2 \\ & \mid \texttt{get}\ \ell \mid \texttt{set}\ \ell\ v && \text{mutable state} \\ & \mid \texttt{choose}\ M_1\ M_2 \mid \texttt{fail} && \text{non-determinism} \end{aligned}$$

Can we evaluate the `do`, the function calls, and the conditionals while leaving the effects uninterpreted?

Executing a monadic program without performing the effects

We define a type of intermediate evaluation results, representing all possible sequences of program effects.

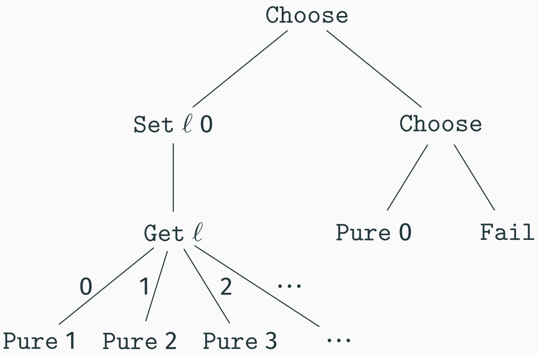
$$
\begin{array}{rcl}
R\,A & = & \texttt{Pure} : A \to R\,A \\
& \mid & \texttt{Get} : \mathit{Loc} \to (\mathit{Val} \to R\,A) \to R\,A \\
& \mid & \texttt{Set} : \mathit{Loc} \to \mathit{Val} \to R\,A \to R\,A \\
& \mid & \texttt{Choose} : R\,A \to R\,A \to R\,A \\
& \mid & \texttt{Fail} : R\,A
\end{array}
$$

A tree-shaped representation of effects

Program:

$$\texttt{choose}\ (\texttt{do}\ \_ \Leftarrow \texttt{set}\ \ell\ 0\ \texttt{in}\ \texttt{do}\ x \Leftarrow \texttt{get}\ \ell\ \texttt{in}\ \texttt{val}\ (x+1))$$
$$(\texttt{choose}\ (\texttt{val}\ 0)\ \texttt{fail})$$

Intermediate result:



The monad of intermediate results

$$\begin{array}{rcl} R\,A & = & \texttt{Pure} : A \to R\,A \\ & | & \texttt{Get} : \mathit{Loc} \to (\mathit{Val} \to R\,A) \to R\,A \\ & | & \texttt{Set} : \mathit{Loc} \to \mathit{Val} \to R\,A \to R\,A \\ & | & \texttt{Choose} : R\,A \to R\,A \to R\,A \\ & | & \texttt{Fail} : R\,A \end{array}$$

This type is a monad, wit $\texttt{ret} \stackrel{def}{=} \texttt{Pure}$ and $\texttt{bind}$ defined as:

$$\begin{aligned} \texttt{bind}\,(\texttt{Pure}\,v)\,f &= f\,v \\ \texttt{bind}\,(\texttt{Get}\,\ell\,k)\,f &= \texttt{Get}\,\ell\,(\lambda v.\,\texttt{bind}\,(k\,\ell)\,f) \\ \texttt{bind}\,(\texttt{Set}\,\ell\,v\,R)\,f &= \texttt{Set}\,\ell\,v\,(\texttt{bind}\,R\,f) \\ \texttt{bind}\,(\texttt{Choose}\,R_1\,R_2)\,f &= \texttt{Choose}\,(\texttt{bind}\,R_1\,f)\,(\texttt{bind}\,R_2\,f) \\ \texttt{bind}\,\texttt{Fail}\,f &= \texttt{Fail} \end{aligned}$$

A denotational semantics for monadic programs

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

$$\begin{aligned}
[\![v_1\ v_2]\!] &= [\![v_1]\!]_v\ [\![v_2]\!]_v \\
[\![\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_2]\!] &= \texttt{if}\ [\![v]\!]_v\ \texttt{then}\ [\![M_1]\!]\ \texttt{else}\ [\![M_2]\!] \\
[\![\texttt{val}\ v]\!] &= \texttt{Pure}\ [\![v]\!]_v \\
[\![\texttt{do}\ x \Leftarrow M_1\ \texttt{in}\ M_2]\!] &= \texttt{bind}\ [\![M_1]\!]\ (\lambda x.\ [\![M_2]\!]) \\
[\![\texttt{get}\ \ell]\!] &= \texttt{Get}\ \ell\ (\lambda v.\ \texttt{Pure}\ v) \\
[\![\texttt{set}\ \ell\ v]\!] &= \texttt{Set}\ \ell\ [\![v]\!]_v\ (\texttt{Pure}\ (\,)) \\
[\![\texttt{choose}\ M_1\ M_2]\!] &= \texttt{Choose}\ [\![M_1]\!]\ [\![M_2]\!] \\
[\![\texttt{fail}]\!] &= \texttt{Fail}
\end{aligned}$$

Where $[\![cst]\!]_v = cst$, $[\![x]\!]_v = x$, $[\![\lambda x.\, M]\!]_v = \lambda x.\ [\![M]\!]$.

Interpreting effects

Finally, we can interpret effects (function `run`) using a *fold* traversal of the result tree R.

With backtracking of the store at choice points: `run` has type $R\ A \to \mathit{Store} \to \mathit{Set}\ A$ and we take

$$\begin{aligned}
\texttt{run}\ (\texttt{Pure}\ v)\ s &= \{v\} \\
\texttt{run}\ (\texttt{Get}\ \ell\ k)\ s &= \texttt{run}\ (k\ (s\ \ell))\ s \\
\texttt{run}\ (\texttt{Set}\ \ell\ v\ R)\ s &= \texttt{run}\ R\ (s\{\ell \leftarrow v\}) \\
\texttt{run}\ \texttt{Fail}\ s &= \emptyset \\
\texttt{run}\ (\texttt{Choose}\ R_1\ R_2)\ s &= \texttt{run}\ R_1\ s \cup \texttt{run}\ R_2\ s
\end{aligned}$$

Interpreting effects

Finally, we can interpret effects (function `run`) using a *fold* traversal of the result tree R.

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

$$
\begin{aligned}
\mathtt{run}\ (\mathtt{Pure}\ v)\ s &= (\{v\}, s) \\
\mathtt{run}\ (\mathtt{Get}\ \ell\ k)\ s &= \mathtt{run}\ (k\ (s\ \ell))\ s \\
\mathtt{run}\ (\mathtt{Set}\ \ell\ v\ R)\ s &= \mathtt{run}\ R\ (s\{\ell \leftarrow v\}) \\
\mathtt{run}\ \mathtt{Fail}\ s &= (\emptyset, s) \\
\mathtt{run}\ (\mathtt{Choose}\ R_1\ R_2)\ s &= (V_1 \cup V_2, s_2) \\
&\qquad \text{with } \mathtt{run}\ R_1\ s = (V_1, {\color{red}s_1}) \text{ and } \mathtt{run}\ R_2\ {\color{red}s_1} = (V_2, s_2)
\end{aligned}
$$

The free monad

The type $R\,A$ is an instance of a more general categorical construction: the free monad.

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & \mid & \mathtt{Op} : F\,(R\,A) \to R\,A \end{array}$$

where $F : \mathit{Type} \to \mathit{Type}$ is a functor: it comes with an operation

$$\mathtt{fmap} : \forall A, B,\ (A \to B) \to (F\,A \to F\,B)$$

We recover the previous example by defining F as

$$\begin{array}{rll} F\,X = & \mathtt{Get} : \mathit{Loc} \to (\mathit{Val} \to X) \to F\,X & \mid \mathtt{Set} : \mathit{Loc} \to \mathit{Val} \to X \to F\,X \\ \mid & \mathtt{Choose} : X \to X \to F\,X & \mid \mathtt{Fail} : F\,X \end{array}$$

Exercise: define `fmap`.

The free monad

$$
\begin{array}{rcl}
R\,A & = & \mathtt{Pure} : A \to R\,A \\
& \mid & \mathtt{Op} : F\,(R\,A) \to R\,A
\end{array}
$$

This “functorial” presentation makes it possible to define `ret` and `bind` in a generic way:

$$
\begin{array}{rcl}
\mathtt{ret}\;v & = & \mathtt{Pure}\;v \\
\mathtt{bind}\;(\mathtt{Pure}\;v)\;f & = & f\;v \\
\mathtt{bind}\;(\mathtt{Op}\;\varphi)\;f & = & \mathtt{Op}\;(\mathtt{fmap}\;(\lambda x.\,\mathtt{bind}\;x\;f)\;\varphi)
\end{array}
$$

The freer monad

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

Another generic construction of the type of intermediate execution results.

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & | & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \end{array}$$

$\mathit{Eff}\ B$ is the type of effects producing a result of type B. Each specific effect is a constructor of type Eff.

If $\varphi : \mathit{Eff}\ B$, the subtrees of $\mathtt{Op}(\varphi, k)$ 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:

$$
\begin{array}{rll}
\texttt{Get} : \mathit{Loc} \to \mathit{Eff}\ \mathit{Val} & & \text{(one subtree per possible value)} \\
\texttt{Set} : \mathit{Loc} \to \mathit{Val} \to \mathit{Eff}\ \texttt{unit} & & \text{(one subtree)} \\
\texttt{Fail} : \mathit{Eff}\ \texttt{empty} & & \text{(no subtree)} \\
\texttt{Flip} : \mathit{Eff}\ \texttt{bool} & & \text{(two subtrees)}
\end{array}
$$

We encode the `choose` operation using the `Flip` effect:

$$
\texttt{choose}\ R_1\ R_2 \stackrel{\mathit{def}}{=} \texttt{Op}(\texttt{Flip}, \lambda b.\ \texttt{if}\ b\ \texttt{then}\ R_1\ \texttt{else}\ R_2)
$$

The freer monad

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & | & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \end{array}$$

This presentation “indexed by type B” also makes it possible to define `ret` and `bind` generically:

$$\begin{array}{rcl} \mathtt{ret}\ v & = & \mathtt{Pure}\ v \\ \mathtt{bind}\ (\mathtt{Pure}\ v)\ f & = & f\ v \\ \mathtt{bind}\ (\mathtt{Op}\ \varphi\ k)\ f & = & \mathtt{Op}\ (\varphi, \lambda x.\ \mathtt{bind}\ (k\ x)\ f) \end{array}$$

We no longer need a functor nor a `fmap`.

Interpreting effects in the freer monad

Using a generic *fold* over the type of results:

$$\begin{aligned}
\mathtt{run} &: (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\
\mathtt{run}\ f\ g\ (\mathtt{Pure}\ v) &= f\ v \\
\mathtt{run}\ f\ g\ (\mathtt{Op}\ \varphi\ k) &= g\ \varphi\ (\lambda x.\ \mathtt{run}\ f\ g\ (k\ x))
\end{aligned}$$

For non-determinism with backtracking of state, we take

$$\begin{aligned}
f &: A \to \mathit{Store} \to \mathit{Set}\ A \\
f\ x\ s &= \{x\} \\
g &: \mathit{Eff}\ B \to (B \to \mathit{Store} \to \mathit{Set}\ A) \to \mathit{Store} \to \mathit{Set}\ A
\end{aligned}$$

$$g\ (\mathtt{Get}\ \ell)\ k\ s = k\ (s\ \ell)\ s \qquad g\ (\mathtt{Set}\ \ell\ v)\ k\ s = k\ ()\ s\{\ell \leftarrow v\}$$

$$g\ \mathtt{Flip}\ k\ s = k\ \mathtt{false}\ s \cup k\ \mathtt{true}\ s \qquad g\ \mathtt{Fail}\ k\ s = \emptyset$$

Interpreting effects in the freer monad

Using a generic *fold* over the type of results:

$$\begin{aligned}
\mathtt{run} &: (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\
\mathtt{run}\ f\ g\ (\mathtt{Pure}\ v) &= f\ v \\
\mathtt{run}\ f\ g\ (\mathtt{Op}\ \varphi\ k) &= g\ \varphi\ (\lambda x.\ \mathtt{run}\ f\ g\ (k\ x))
\end{aligned}$$

Note the control inversion: it's no longer the program that calls the `get`, `set`, … operations of the monad;
it's the implementation of these operations (the g function) that evaluates the program "on demand" using the continuation k.

Interaction trees

(Xia, Zakowski, *et al*, *Interaction Trees*, POPL 2020).

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

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & \mid & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \\ & \mid & \mathtt{Tau} : R\,A \to R\,A \end{array}$$

`Tau` denotes one step of computation without effects.

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

The infinite tree $x \stackrel{def}{=} \mathtt{Op}(\mathtt{Flip}, \lambda b.\ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{Pure}\ \ 0\ \mathtt{else}\ x)$ represents `let rec f () = 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

$$
\begin{aligned}
\varepsilon &: T && \text{identityl element} \\
\cdot &: T \to T \to T && \text{composition} \\
\varepsilon \cdot x &= x && \text{left identity} \\
x \cdot \varepsilon &= x && \text{right identity} \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z) && \text{associativity}
\end{aligned}
$$

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 group is $(T, 0, +, -)$ where

$0 : T$	identity element
$+ : T \to T \to T$	composition
$- : T \to T$	inverse
$0 + x = x + 0 = x$	identity
$(x + y) + z = x + (y + z)$	associativity
$(-x) + x = x + (-x) = 0$	inverse

Theories and models

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”):

$$(\mathbb{N}, 0, +) \qquad (\mathbb{R}, 1, \times) \qquad (T \to T, id, \circ)$$

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

$$(\mathbb{Z}, 0, +, -) \qquad (\mathbb{R}^*, 1, \times, ^{-1})$$

Algebraic abstract types

An algebraic abstract type is the specification of a persistent data structure as a signature and equations.

(→ 2022–2023 course, lecture #1)

Example: stacks

$$\mathtt{empty} : S \qquad \mathtt{push} : E \to S \to S \qquad \mathtt{top} : S \to E \quad \mathtt{pop} : S \to S$$

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

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

$$\mathtt{add} : S \to E \to S$$

$$\mathtt{add}\ \mathtt{empty}\ v = \mathtt{push}\ v\ \mathtt{empty} \qquad \mathtt{add}\ (\mathtt{push}\ w\ s)\ v = \mathtt{push}\ w\ (\mathtt{add}\ s\ v)$$

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 $a_1 a_2 \dots a_n$;
- identity element ε: the empty list;
- composition $\cdot$: list concatenation.

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

$$1 \cdot (23 \cdot 456) = (1 \cdot 23) \cdot 456 = 123456$$

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

$$\Phi(a_1 \dots a_n) = 0 + a_1 + \cdots + a_n$$

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

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

$$\Phi(\varepsilon) = 0 \qquad \Phi(\ell_1 \cdot \ell_2) = \Phi(\ell_1) + \Phi(\ell_2)$$

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 $f : X \to supp(M)$ such that:

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

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "f'"'] & supp(M) \arrow[d, "\Phi"] \\ & supp(M') \end{tikzcd}$$

Monads viewed as algebraic structures

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

$$\begin{aligned}
\mathtt{ret} &: A \to T\ A \\
\mathtt{bind} &: T\ A \to (A \to T\ B) \to T\ B \\
\mathtt{op}(F) &: P \to (B \to T\ A) \to T\ A \quad \text{if } F : P \to \mathit{Eff}\ B
\end{aligned}$$

The equations are the three monadic laws, plus other laws for the $\mathtt{op}(F)$ operations.

Free monads are free!

The free monad and the freer monad are free monads generated by the constructors of type *Eff*.

Let's check this fact for the freer monad.

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & | & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \end{array}$$

The associated monad structure, with the expected signature:

$$\begin{aligned} \mathtt{ret}\ x &= \mathtt{Pure}\ x \\ \mathtt{bind}\ (\mathtt{Pure}\ x)\ f &= f\ x \\ \mathtt{bind}\ (\mathtt{Op}(\varphi, k))\ f &= \mathtt{Op}(\varphi, \lambda x.\ \mathtt{bind}\ (k\ x)\ f) \\ \mathtt{op}(F) &= \lambda x.\ \mathtt{Op}(F\ x, \lambda y.\ \mathtt{Pure}\ y) \end{aligned}$$

Free monads are free!

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

$$\begin{array}{rcl} \Phi & : & R\,A \to T\,A \\ \Phi(\mathtt{Pure}\ v) & = & \mathtt{ret}_M\ v \\ \Phi(\mathtt{Op}\ (F\,x, k)) & = & \mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(k\ y)) \end{array}$$

This morphism commutes with operations `ret` and `bind`.

$$\begin{array}{rcl} \Phi(\mathtt{bind}\ (\mathtt{Pure}\ v)\ f) & = & \Phi(f\ v) \ = \ \mathtt{bind}_M\ (\Phi(\mathtt{Pure}\ v))\ (\lambda y.\ \Phi(f\ y)) \quad \text{(1st law)} \\ \Phi(\mathtt{bind}\ (\mathtt{Op}(F\,x, k))\ f) & = & \Phi(\mathtt{Op}(F\,x, \lambda y.\ \mathtt{bind}\ (k\ y)\ f)) \\ & = & \mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(\mathtt{bind}\ (k\ y)\ f)) \\ \text{(3rd law)} & = & \mathtt{bind}_M\ (\mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(k\ y)))\ (\lambda z.\ \Phi(f\ z)) \\ & = & \mathtt{bind}_M\ (\Phi(\mathtt{Op}(F\,x, k)))\ (\lambda z.\ \Phi(f\ z)) \end{array}$$

Algebraic effects

An algebraic vision of 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:
$$\begin{array}{rll} M, N ::= & v\ v' & \text{application} \\ \mid & \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N & \text{conditional} \\ \mid & \texttt{val}\ v & \text{trivial computation} \\ \mid & \texttt{do}\ x \Leftarrow M\ \texttt{in}\ N & \text{sequencing} \\ \mid & F(\vec{v}; y.\, M) & \text{effectful operation} \end{array}$$

$F(v_1 \ldots v_n; y.\, M)$ 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: $F(\vec{v}\,) \stackrel{def}{=} F(\vec{v}; y.\, \texttt{val}(y))$ (trivial continuation).

The laws of the computational lambda-calculus with effects

Same laws as for the computational lambda-calculus:

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

$$\mathtt{do}\; x \Leftarrow \mathtt{val}\; v \;\mathtt{in}\; M = M\{x \leftarrow x\}$$

$$\mathtt{do}\; x \Leftarrow M \;\mathtt{in}\; \mathtt{val}\; x = M$$

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

Plus commutation between `do` and effectful operations:

$$\mathtt{do}\; x \Leftarrow F(\vec{v}; y.\, M) \;\mathtt{in}\; N = F(\vec{v}; y.\, \mathtt{do}\; x \Leftarrow M \;\mathtt{in}\; N)$$

Plus laws specific to some effects.

Laws for mutable state

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

$$\mathtt{set}(\ell, v;\ \_.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{set}(\ell, v;\ \_.\, M\{z \leftarrow v\})$$

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

Other commutations between accesses to different locations:

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{get}(\ell; y.\, M))$$

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

Other commutations between accesses to the same location:

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{get}(\ell; y.\, M\{z \leftarrow y\}) \qquad \text{(double read)}$$

$$\mathtt{get}(\ell; y.\, \mathtt{set}(\ell, y;\ \_.\, M)) = M \qquad \text{(read then rewrite)}$$

$$\mathtt{set}(\ell, v_1; \_.\, \mathtt{set}(\ell, v_2; \_.\, M)) = \mathtt{set}(\ell, v_2; \_.\, M) \qquad \text{(double write)}$$

Laws for non-determinism

For failure:

$$\mathtt{Fail}(;k) = \mathtt{Fail}(;k') = \mathtt{Fail}() \qquad \text{(propagation)}$$

For choice:

$$\mathtt{choose}\ M\ M = M \qquad \text{(idempotent)}$$

$$\mathtt{choose}\ M\ N = \mathtt{choose}\ N\ M \qquad \text{(commutative)}$$

$$\mathtt{choose}\ (\mathtt{choose}\ M\ N)\ P = \mathtt{choose}\ M\ (\mathtt{choose}\ N\ P) \qquad \text{(associative)}$$

$$\mathtt{choose}\ \mathtt{Fail}()\ M = \mathtt{choose}\ M\ \mathtt{Fail}() = M \qquad \text{(identity)}$$

Less natural to express with the encoding

$$\mathtt{choose}\ M\ N = \mathtt{Flip}(; \lambda b.\ \mathtt{if}\ b\ \mathtt{then}\ M\ \mathtt{else}\ N)$$

A semantic for computational lambda-calculus with effects

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

$$
\begin{aligned}
[\![v_1\ v_2]\!] &= [\![v_1]\!]_v\ [\![v_2]\!]_v \quad \text{or} \quad \mathtt{Tau}([\![v_1]\!]_v\ [\![v_2]\!]_v) \\
[\![\mathtt{val}\ v]\!] &= \mathtt{Pure}\ [\![v]\!]_v \\
[\![\mathtt{do}\ x \Leftarrow M_1\ \mathtt{in}\ M_2]\!] &= \mathtt{bind}\ [\![M_1]\!]\ (\lambda x.\ [\![M_2]\!]) \\
[\![F(\vec{v}; y.\ M)]\!] &= \mathtt{Op}\ (F\ \vec{v})\ (\lambda y.\ [\![M]\!])
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{fold} : (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\
\mathtt{fold}\, f\ g\ (\mathtt{Pure}\ v) = f\ v \\
\mathtt{fold}\, f\ g\ (\mathtt{Op}\ \varphi\ k) = g\ \varphi\ (\lambda x.\ \mathtt{fold}\, f\ g\ (k\ x))
\end{aligned}
$$

Interpreting effects by composing handlers

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.

$$\begin{aligned}
\texttt{state} : R\,A \to \mathit{Store} \to R\,A &= \texttt{fold}\, f_{state}\; g_{state} \\
f_{state}\; v &= \lambda s.\, \texttt{Pure}\; v \\
g_{state}\; (\texttt{Get}\;\ell)\; k &= \lambda s.\, k\,(s\,\ell)\; s \\
g_{state}\; (\texttt{Set}\;\ell\; v)\; k &= \lambda s.\, k\;()\; s\{\ell \leftarrow v\} \\
g_{state}\; \varphi\; k &= \lambda s.\, \texttt{Op}(\varphi, \lambda x.\, k\; x\; s) \quad \text{for all other } \varphi
\end{aligned}$$

Interpreting effects by composing handlers

Example: a handler for the `Flip` and `Fail` effects.

$$\begin{aligned}
\texttt{nondet} : R\,A \to R\,(Set\,A) &= \texttt{fold}\, f_{nondet}\, g_{nondet} \\
f_{nondet}\, v &= \texttt{Pure}\ \{v\} \\
g_{nondet}\, \texttt{Fail}\ k &= \texttt{Pure}\ \emptyset \\
g_{nondet}\, \texttt{Flip}\ k &= \texttt{bind}\ (k\ \texttt{true})\ (\lambda x_1. \\
&\quad\ \texttt{bind}\ (k\ \texttt{false})\ (\lambda x_2. \\
&\quad\ \texttt{Pure}\ (x_1 \cup x_2))) \\
g_{nondet}\, \varphi\ k &= \texttt{Op}(\varphi, k) \quad \text{for all other } \varphi
\end{aligned}$$

Interpreting effects by composing handlers

The composition `nondet` (`state` $t\ s_0$) implements the semantics where the store is backtracked at choice points.

The composition `state` (`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 `Pure` v where v is the final value of the program.

Implementing equations correctly

The equations related to `bind` are automatically satisfied by the semantics based on interaction trees.

The other equations must be satisfied by the handlers that interpret the effects.

Laws for mutable state

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):

$$s\{\ell \leftarrow v\}\ \ell = v$$

$$s\{\ell \leftarrow v\}\ \ell' = s\ \ell' \quad \text{if } \ell' \neq \ell$$

$$s\{\ell \leftarrow v\}\{\ell \leftarrow v'\} = s\{\ell \leftarrow v'\}$$

$$s\{\ell \leftarrow v\}\{\ell' \leftarrow v'\} = s\{\ell' \leftarrow v'\}\{\ell \leftarrow v\} \quad \text{if } \ell' \neq \ell$$

$$s\{\ell \leftarrow s\ \ell\} = s$$

Exercise: show that `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: $v ::= x \mid cst \mid \lambda x.\, M$

Computations:
$$
\begin{array}{rll}
M, N ::= & v\ v' & \\
\mid & \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N & \\
\mid & \texttt{val}\ v & \\
\mid & \texttt{do}\ x \Leftarrow M\ \texttt{in}\ N & \\
\mid & F(\vec{v}; y.\, M) & \text{effectful operation} \\
\mid & \texttt{with}\ H\ \texttt{handle}\ M & \text{effect handler}
\end{array}
$$

Handlers:
$$
\begin{array}{rl}
H ::= \{ & \texttt{val}(x) \to M_{val}; \\
& F_1(\vec{x}; k) \to M_1; \\
& \dots \\
& F_n(\vec{x}; k) \to M_n\ \}
\end{array}
$$

Intuitive semantics for effect handlers

$$\texttt{with}\ \{\texttt{val}(x) \to M_{val}\,; \ldots; F_i(\vec{x}; k) \to M_i\,; \ldots\}\ \texttt{handle}\ M$$

If M terminates with value v, the M_{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

$k = \lambda y.\,N$ or $k = \lambda y.\,\texttt{with}\ \{\ldots\}\ \texttt{handle}\ N$.

(shallow handler) *(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 $F(\vec{v}; y.\,\texttt{with}\ \{\ldots\}\ \texttt{handle}\ N)$.

(shallow handler) *(deep handler)*

Denotational semantics of effect handlers

The denotation $[\![H]\!]$ of an effect handler is an interaction tree transformer, so that

$$[\![\texttt{with}\ H\ \texttt{handle}\ M]\!] = [\![H]\!]\ [\![M]\!]$$

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

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

`fold` and `case` are defined as

$$\begin{aligned} \texttt{fold}\ f\ g\ (\texttt{Pure}\ v) &= f\ v \\ \texttt{fold}\ f\ g\ (\texttt{Op}(\varphi, k)) &= g\ \varphi\ (\lambda x.\ \texttt{fold}\ f\ g\ (k\ x)) \end{aligned} \qquad \begin{aligned} \texttt{case}\ f\ g\ (\texttt{Pure}\ v) &= f\ v \\ \texttt{case}\ f\ g\ (\texttt{Op}(\varphi, k)) &= g\ \varphi\ k \end{aligned}$$

Denotational semantics of effect handlers

$$H = \{\mathtt{val}(x) \to M_{val}\ ; F_1(\vec{x}; k) \to M_1\ ; \dots ; F_n(\vec{x}; k) \to M_n\}$$

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

$$\begin{aligned}
[\![H]\!]_{ret}\ x &= [\![M_{val}]\!] \\
[\![H]\!]_{eff}\ (F_i\ \vec{x}\,)\ k &= [\![M_i]\!] \\
[\![H]\!]_{eff}\ (F\ \vec{x}\,)\ k &= \mathtt{Op}\ (F\ \vec{x}\,)\ k \quad \text{if } F \notin \{F_1, \dots, F_n\}
\end{aligned}$$

Summary

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:

- Philip Wadler: *Monads for functional programming*. 1995. https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

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

The freer monad:

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

The algebraic viewpoint:

- Andrej Bauer: *What is algebraic about algebraic effects and handlers?* 2018. https://arxiv.org/abs/1807.05923