[CSE301 / Lecture 4]
Side-effects and monads

Noam Zeilberger

Ecole Polytechnique

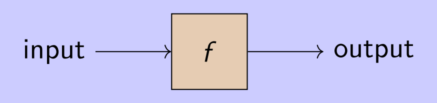
27 September 2023

What are side-effects?

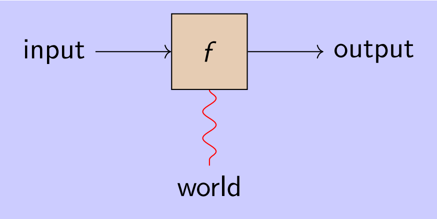
Everything that a function does besides computing a functional relation from inputs to outputs.

In other words, the difference between "functions in math" and "functions in Python".

What are side-effects?



versus



“pure” function = no side-effects

```
def sqr(x):
    y = x ** 2
    return y
```

```
>>> sqr(3)
9
>>> sqr(4)
16
```

Printing debugging information

```
def sqr_debug(x):
    y = x ** 2
    print("Squaring {} gives {}!!".format(x, y))
    return y
```

```
>>> sqr_debug(3)
Squaring 3 gives 9!!
9
>>> sqr_debug(4)
Squaring 4 gives 16!!
16
```

Getting input from the user, and raising exceptions

```
def exp_by_input(x):
    k = int(input("Enter exponent: "))
    y = x ** k
    return y
```

```
>>> exp_by_input(3)
Enter exponent: 2
9
>>> exp_by_input(3)
Enter exponent: two
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 7, in exp_by_input
ValueError: invalid literal for int() with base 10: 'two'
```

Querying a random number generator

```
def exp_by_random(x):
    k = random.randrange(0,10)
    y = x ** k
    return y
```

```
>>> exp_by_random(3)
2187
>>> exp_by_random(3)
243
```

Reading and writing global variables

```
k = 0
def exp_by_counter(x):
    global k
    y = x ** k
    k = k + 1
    return y
```

```
>>> exp_by_counter(3)
1
>>> exp_by_counter(3)
3
>>> exp_by_counter(3)
9
```

Non-standard control flow (e.g., nondeterminism via generators)

```
def exp_by_nondet(x):
    for k in range(0,10):
        y = x ** k
        yield y
```

```
>>> exp_by_nondet(3)
<generator object exp_by_nondet at 0x7ff77d38c830>
>>> list(exp_by_nondet(3))
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683]
```

Side-effects in Haskell

Despite claims, Haskell is not really a pure language...

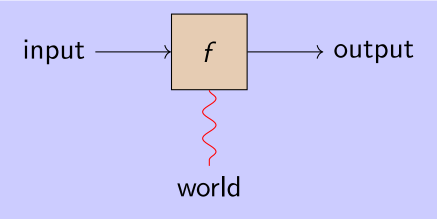
1. Functions may not terminate
2. Functions may raise exceptions
3. Run-time performance can vary wildly due to laziness

Nevertheless, these effects (at least 1 & 2) are relatively "benign".

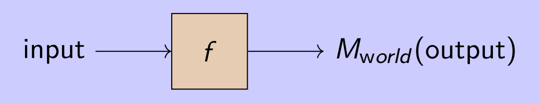
In Haskell, most "serious" effects (like getting input from the user, or reading and writing global variables) are confined to *monads.*

The idea, very roughly

Replace



by



where M_{world} captures all possible interactions with the world.

We'll make this more precise, but first let's talk a bit about the principles of *referential transparency* and *compositionality*...

The principle of referential transparency

Informal principle that we can replace an expression by the value it computes without changing the behavior of a program, e.g.:

```
>>> sqr(3)
9
>>> 9 == 9
True
>>> sqr(3) == 9
True
>>> sqr(3) == sqr(3)
True
```

The principle of referential transparency

The presence of side-effects can break referential transparency!

```
>>> exp_by_counter(3)
9
>>> exp_by_counter(3) == 9
False
>>> exp_by_counter(3) == exp_by_counter(3)
False
```

More generally, a *semantics* for a programming language is a way of assigning meanings to program expressions. A desired property of a semantics is that it is *compositional*, in the sense that the meaning of an expression is built from the meanings of its subexpressions.

The presence of side-effects presents a challenge to defining a compositional semantics![1] But we can try to surmount it...

[1] Aside: this is also true for semantics of natural languages! See Chung-chieh Shan's PhD thesis, *Lingustic side effects* (2005).

Toy example:[2] arithmetic expressions

Consider a little language of arithmetic expressions, with constants, subtraction, and division:

$$e ::= c \mid e1 - e2 \mid e1 \mathbin{/} e2$$

Each expression e denotes a number $[\![e]\!] \in \mathbb{R}$, defined inductively:

$$\begin{aligned} [\![c]\!] &= c \\ [\![e1 - e2]\!] &= [\![e1]\!] - [\![e2]\!] \\ [\![e1 \mathbin{/} e2]\!] &= [\![e1]\!] \mathbin{/} [\![e2]\!] \end{aligned}$$

Division by zero is undefined, so $[\![e]\!]$ is sometimes undefined.

[2] Inspired in part by Philip Wadler, "Monads for functional programming", Proceedings of the Båstad Spring School, May 1995.

Toy example: arithmetic expressions

Translated to Haskell:

data *Expr* = *Con Double* | *Sub Expr Expr* | *Div Expr Expr*

```
eval :: Expr → Double
eval (Con c) = c
eval (Sub e1 e2) = eval e1 − eval e2
eval (Div e1 e2) = eval e1 / eval e2
```

Toy example: arithmetic expressions

Example expressions:

$$\textit{e1} = \textit{Sub}\ (\textit{Div}\ (\textit{Con}\ 2)\ (\textit{Con}\ 4))\ (\textit{Con}\ 3)$$
$$\textit{e2} = \textit{Sub}\ (\textit{Con}\ 1)\ (\textit{Div}\ (\textit{Con}\ 2)\ (\textit{Con}\ 2))$$
$$\textit{e3} = \textit{Div}\ (\textit{Con}\ 1)\ (\textit{Sub}\ (\textit{Con}\ 2)\ (\textit{Con}\ 2))$$

And their semantics:

$\textit{eval e1} = -2.5$ $\qquad$ $\textit{eval e2} = 0$ $\qquad$ *eval e3* undefined

Variation #1: error-handling

Modify the semantics to handle division-by-zero.

In a language with exceptions, we could simply raise an exception. Haskell has them, but let's pretend it doesn't and stay "pure"...

Idea: e no longer denotes a number, but a "number or error".

That is, $[\![e]\!] \in \mathbb{R} \uplus \{error\}$

In Haskell, we can return a Maybe type...

Variation #1: error-handling

```
eval1 :: Expr → Maybe Double
eval1 (Con c)    = Just c
eval1 (Sub e1 e2) =
  case (eval1 e1, eval1 e2) of
    (Just x1, Just x2) → Just (x1 − x2)
    _ → Nothing
eval1 (Div e1 e2) =
  case (eval1 e1, eval1 e2) of
    (Just x1, Just x2)
      | x2 ≢ 0 → Just (x1 / x2)
      | otherwise → Nothing
    _              → Nothing
```

Variation #1: error-handling

The example expressions:

$$\mathit{e1} = \mathit{Sub}\ (\mathit{Div}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 4))\ (\mathit{Con}\ 3)$$
$$\mathit{e2} = \mathit{Sub}\ (\mathit{Con}\ 1)\ (\mathit{Div}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 2))$$
$$\mathit{e3} = \mathit{Div}\ (\mathit{Con}\ 1)\ (\mathit{Sub}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 2))$$

In the new semantics:

$$\mathit{eval1}\ \mathit{e1} = \mathit{Just}\ (-2.5) \qquad \mathit{eval1}\ \mathit{e2} = \mathit{Just}\ 0.0$$
$$\mathit{eval1}\ \mathit{e3} = \mathit{Nothing}$$

Variation #2: global state

Modify the semantics of expressions so that every third constant is interpreted as 0. (Yeah this is a bit weird, but so is most of life.)

The meaning of a subexpression now depends on its position. E.g., $[\![3-2]\!] = [\![1]\!]$, but $[\![6 \,/\, (3-2)]\!] = [\![6 \,/\, (3-0)]\!] \neq [\![6 \,/\, 1]\!]$.

Can we define a compositional semantics?...

Variation #2: global state

…Yes, in state-passing style!

Idea: every subexpression e denotes a function $[\![e]\!] \in \mathbb{N} \to \mathbb{R} \times \mathbb{N}$ taking a count of the previously seen constants, and returning a number together with an updated count.

For the top-level expression, initialize count to 0.

Variation #2: global state

$$\begin{array}{l}
\mathit{eval2} :: \mathit{Expr} \to \mathit{Int} \to (\mathit{Double}, \mathit{Int}) \\
\mathit{eval2}\ (\mathit{Con}\ c)\ n = (\textbf{if}\ n\ `\mathit{mod}`\ 3 \equiv 2\ \textbf{then}\ 0\ \textbf{else}\ c, n + 1) \\
\mathit{eval2}\ (\mathit{Sub}\ \mathit{e1}\ \mathit{e2})\ n = \\
\quad \textbf{let}\ (\mathit{x1}, o) = \mathit{eval2}\ \mathit{e1}\ n\ \textbf{in} \\
\quad \textbf{let}\ (\mathit{x2}, p) = \mathit{eval2}\ \mathit{e2}\ o\ \textbf{in} \\
\quad (\mathit{x1} - \mathit{x2}, p) \\
\mathit{eval2}\ (\mathit{Div}\ \mathit{e1}\ \mathit{e2})\ n = \\
\quad \textbf{let}\ (\mathit{x1}, o) = \mathit{eval2}\ \mathit{e1}\ n\ \textbf{in} \\
\quad \textbf{let}\ (\mathit{x2}, p) = \mathit{eval2}\ \mathit{e2}\ o\ \textbf{in} \\
\quad (\mathit{x1}\ /\ \mathit{x2}, p)
\end{array}$$

$$\begin{array}{l}
\mathit{eval2Top} :: \mathit{Expr} \to \mathit{Double} \\
\mathit{eval2Top}\ e = \mathit{fst}\ (\mathit{eval2}\ e\ 0)
\end{array}$$

Variation #2: global state

The example expressions:

$$\mathit{e1} = \mathit{Sub}\ (\mathit{Div}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 4))\ (\mathit{Con}\ 3)$$
$$\mathit{e2} = \mathit{Sub}\ (\mathit{Con}\ 1)\ (\mathit{Div}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 2))$$
$$\mathit{e3} = \mathit{Div}\ (\mathit{Con}\ 1)\ (\mathit{Sub}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 2))$$

In the new semantics:

$\mathit{eval2Top}\ \mathit{e1} = \mathit{eval2Top}\ \mathit{e3} = 0.5$ $\qquad$ *eval2Top e2* undefined

Variation #3: combining error-handling and state

$$
\begin{array}{l}
\mathit{eval3} :: \mathit{Expr} \to \mathit{Int} \to \mathit{Maybe}\ (\mathit{Double}, \mathit{Int}) \\
\mathit{eval3}\ (\mathit{Con}\ c)\ n = \mathit{Just}\ (\textbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 3 \equiv 2\ \textbf{then}\ 0\ \textbf{else}\ c, n + 1) \\
\mathit{eval3}\ (\mathit{Sub}\ \mathit{e1}\ \mathit{e2})\ n = \\
\quad \textbf{case}\ \mathit{eval3}\ \mathit{e1}\ n\ \textbf{of} \\
\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad \mathit{Just}\ (\mathit{x1}, o) \to \textbf{case}\ \mathit{eval3}\ \mathit{e2}\ o\ \textbf{of} \\
\qquad\quad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad\quad \mathit{Just}\ (\mathit{x2}, p) \to \mathit{Just}\ (\mathit{x1} - \mathit{x2}, p) \\
\mathit{eval3}\ (\mathit{Div}\ \mathit{e1}\ \mathit{e2})\ n = \\
\quad \textbf{case}\ \mathit{eval3}\ \mathit{e1}\ n\ \textbf{of} \\
\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad \mathit{Just}\ (\mathit{x1}, o) \to \textbf{case}\ \mathit{eval3}\ \mathit{e2}\ o\ \textbf{of} \\
\qquad\quad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad\quad \mathit{Just}\ (\mathit{x2}, p) \\
\qquad\qquad \mid \mathit{x2} \not\equiv 0 \to \mathit{Just}\ (\mathit{x1}\ /\ \mathit{x2}, p) \\
\qquad\qquad \mid \mathit{otherwise} \to \mathit{Nothing}
\end{array}
$$

Variation #3: combining error-handling and state

$$\begin{array}{l} \textit{eval3Top} :: \textit{Expr} \rightarrow \textit{Maybe Double} \\ \textit{eval3Top}\ e = \textbf{case}\ \textit{eval3}\ e\ 0\ \textbf{of} \\ \quad \textit{Nothing} \rightarrow \textit{Nothing} \\ \quad \textit{Just}\ (x, \_) \rightarrow \textit{Just}\ x \end{array}$$

In this last semantics:

$$\textit{eval3Top e1} = \textit{eval3Top e3} = \textit{Just}\ 0.5$$
$$\textit{eval3Top e2} = \textit{Nothing}$$

Compare with the OCaml version...

```
type expr = Con of float
          | Sub of expr * expr | Div of expr * expr
let cnt = ref 0
let rec eval3 (e : expr) : float =
  match e with
  | Con c -> let n = !cnt in
             (cnt := n+1; if n mod 3 == 2 then 0.0 else c)
  | Sub (e1,e2) -> let x1 = eval3 e1 in
                   let x2 = eval3 e2 in
                   x1 -. x2
  | Div (e1,e2) -> let x1 = eval3 e1 in
                   let x2 = eval3 e2 in
                   if x2 <> 0.0 then x1 /. x2
                   else raise Division_by_zero
let rec eval3Top e = (cnt := 0; eval3 e)
```

Haskell version #3, rewritten using a monad and do notation

$$
\begin{aligned}
&\mathit{eval3}' :: \mathit{Expr} \rightarrow \mathit{StateT}\ \mathit{Int}\ \mathit{Maybe}\ \mathit{Double}\\
&\mathit{eval3}'\ (\mathit{Con}\ c) = \mathbf{do}\\
&\quad n \leftarrow \mathit{get}\\
&\quad \mathit{put}\ (n+1)\\
&\quad \mathit{return}\ (\mathbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 3 \equiv 2\ \mathbf{then}\ 0\ \mathbf{else}\ c)\\
&\mathit{eval3}'\ (\mathit{Sub}\ \mathit{e1}\ \mathit{e2}) = \mathbf{do}\\
&\quad \mathit{x1} \leftarrow \mathit{eval3}'\ \mathit{e1}\\
&\quad \mathit{x2} \leftarrow \mathit{eval3}'\ \mathit{e2}\\
&\quad \mathit{return}\ (\mathit{x1} - \mathit{x2})\\
&\mathit{eval3}'\ (\mathit{Div}\ \mathit{e1}\ \mathit{e2}) = \mathbf{do}\\
&\quad \mathit{x1} \leftarrow \mathit{eval3}'\ \mathit{e1}\\
&\quad \mathit{x2} \leftarrow \mathit{eval3}'\ \mathit{e2}\\
&\quad \mathbf{if}\ \mathit{x2} \not\equiv 0\ \mathbf{then}\ \mathit{return}\ (\mathit{x1}\ /\ \mathit{x2})\ \mathbf{else}\ \mathit{lift}\ \mathit{Nothing}\\
&\mathit{eval3}'\mathit{Top}\ e = \mathit{runStateT}\ (\mathit{eval3}'\ e)\ 0 \ggg\!= \mathit{return} \circ \mathit{fst}
\end{aligned}
$$

What is a monad?

A mathematical concept originating in category theory.

Proposed by Eugenio Moggi as a unifying categorical model for different notions of computation.

Adapted by Phil Wadler as a way of integrating side-effects with pure functional programming, in particular in Haskell.

What is a category?

A **category** consists of the following:

- A set of objects, and a set of arrows between objects.
  (Just like a directed graph.)
- For each object a, an identity arrow $a \to a$.
- For each $a \to b$ and $b \to c$, a composite arrow $a \to c$.
- Such that composition and identity are associative and unital.

Examples:

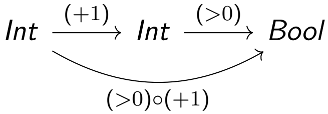
- FG = category freely generated from a graph G, with nodes as objects and arrows $a \to b$ given by *paths* from a to b
- Set = category whose objects are sets and whose arrows $a \to b$ are *functions* from a to b
- Rel = category whose objects are sets and whose arrows $a \to b$ are *relations* from a to b

A category of pure Haskell functions

Informally, we can think of pure functions (of one argument) as forming the arrows of a category, whose objects are types.

$$\mathit{Int} \xrightarrow{(+1)} \mathit{Int} \qquad \mathit{Int} \xrightarrow{(>0)} \mathit{Bool} \qquad \mathit{etc}.$$

Composition of arrows is defined by function composition.



The identity function $\backslash x \rightarrow x$ serves as the identity arrow $a \rightarrow a$.

Beyond the category of pure functions

But... we don't always want to program inside this category!

Monads give us a way of building new categories to program in.

A monad is a special kind of *functor.*

What is a functor?

A **functor** is a way of mapping one category into another (possibly the same) category:

- It should map both objects and arrows.
  (Just like a graph homomorphism.)
- It should preserve identity and composition.
  (Just like a monoid/group homomorphism.)

Examples:

- $F\phi : FG \to FH$ where $\phi : G \to H$ is a graph homomorphism
- the powerset functor $P : \mathrm{Set} \to \mathrm{Set}$

Functors in Haskell

The Functor type class:

> **class** *Functor f* **where**
> $\quad fmap :: (a \to b) \to f\ a \to f\ b$

Note here f is a type constructor $f :: * \to *$.

Any instance should satisfy the *functor laws*:

$$fmap\ id = id \qquad fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

Example #1: the List functor

The list type constructor is a functor:

> **instance** *Functor* [ ] **where**
> -- fmap :: (a -> b) -> [a] -> [b]
> $fmap = map$

(Exercise: prove the functor laws $map\ id\ xs = xs$ and $map\ (f \circ g)\ xs = map\ f\ (map\ g\ xs)$ by structural induction!)

Example #2: the Maybe functor

The *Maybe* type constructor is a functor:

```
instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Rough definition of a monad, in category theory

A **monad** is a functor m from a category to itself, equipped with an arrow $a \to m\,a$ for every object a, together with a way of transforming arrows $a \to m\,b$ into arrows $m\,a \to m\,b$, subject to certain equations.

For example, the powerset functor is a monad: the functions $a \to m\,a$ are defined by taking singletons, and any function $a \to m\,b$ extends to a function $m\,a \to m\,b$ by taking unions.

Rough definition of a monad, in category theory

What's remarkable about the definition is that it allows to build a new category with the same objects, but where an arrow $a \to b$ in the new category corresponds to an arrow $a \to m\, b$ in the old category. This is called the "Kleisli category" construction.

For example, taking the Kleisli category of the powerset monad gives a category with sets as objects but whose arrows are *relations*.

Monads in Haskell (before 2014)

The Monad type class:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
&\quad \mathit{return} :: a \to m\ a \\
&\quad (\ggg\!=) :: m\ a \to (a \to m\ b) \to m\ b \quad \text{-- pronounced "bind"}
\end{aligned}
$$

Subject to the *monad laws*:

$$
\begin{aligned}
&\mathit{return}\ x \ggg\!= f = f\ x \\
&mx \ggg\!= \mathit{return} = mx \\
&(mx \ggg\!= f) \ggg\!= g = mx \ggg\!= (\backslash x \to (f\ x \ggg\!= g))
\end{aligned}
$$

(Note that *flip* $(\ggg\!=) :: (a \to m\ b) \to (m\ a \to m\ b)$.)

The List and Maybe monads

```
instance Monad [] where
  -- return :: a -> [a]
  return x = [x]
  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concatMap f xs

instance Monad Maybe where
  -- return :: a -> Maybe a
  return x = Just x
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

Monads as notions of computation

Via the Kleisli category constructions...

- List monad: category of nondeterministic functions $a \to [b]$.
- Maybe monad: category of partial functions $a \to Maybe\ b$.

Evaluator #1, re-expressed using the Maybe monad

$$
\begin{array}{l}
\mathit{eval1}' :: \mathit{Expr} \rightarrow \mathit{Maybe\ Double} \\
\mathit{eval1}'\ (\mathit{Con\ c}) = \mathit{return\ c} \\
\mathit{eval1}'\ (\mathit{Sub\ e1\ e2}) = \\
\quad \mathit{eval1}'\ \mathit{e1} \gg\!= \backslash \mathit{x1} \rightarrow \\
\quad \mathit{eval1}'\ \mathit{e2} \gg\!= \backslash \mathit{x2} \rightarrow \\
\quad \mathit{return}\ (\mathit{x1} - \mathit{x2}) \\
\mathit{eval1}'\ (\mathit{Div\ e1\ e2}) = \\
\quad \mathit{eval1}'\ \mathit{e1} \gg\!= \backslash \mathit{x1} \rightarrow \\
\quad \mathit{eval1}'\ \mathit{e2} \gg\!= \backslash \mathit{x2} \rightarrow \\
\quad \mathbf{if}\ \mathit{x2} \not\equiv 0\ \mathbf{then}\ \mathit{return}\ (\mathit{x1}\ /\ \mathit{x2})\ \mathbf{else}\ \mathit{Nothing}
\end{array}
$$

The State monad

The type constructor *State s* is defined essentially as follows:

$$\textbf{newtype}\ State\ s\ a = State\ \{\ runState :: s \rightarrow (a, s)\ \}$$

It is a monad:

```
instance Monad State where
  return x = State (\s → (x, s))
  xt >>= f = State (\s0 →
    let (x, s1) = runState xt s0 in
    runState (f x) s1)
```

The State monad

Also, it supports “get” and “set” operations:

$$
\begin{aligned}
&get :: State\ s\ s \\
&get = State\ (\backslash s \rightarrow (s, s)) \\
&put :: s \rightarrow State\ s\ () \\
&put\ s' = State\ (\backslash s \rightarrow ((), s'))
\end{aligned}
$$

Evaluator #2, redefined using the State monad

$$
\begin{array}{l}
\mathit{eval2}' :: \mathit{Expr} \rightarrow \mathit{State}\ \mathit{Int}\ \mathit{Double} \\
\mathit{eval2}'\ (\mathit{Con}\ c) = \\
\quad \mathit{get} \ggg\!\!= \backslash n \rightarrow \\
\quad \mathit{put}\ (n + 1) \ggg\!\!= \backslash\_ \rightarrow \\
\quad \mathit{return}\ (\textbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 3 \equiv 2\ \textbf{then}\ 0\ \textbf{else}\ c) \\
\mathit{eval2}'\ (\mathit{Sub}\ e1\ e2) = \\
\quad \mathit{eval2}'\ e1 \ggg\!\!= \backslash x1 \rightarrow \\
\quad \mathit{eval2}'\ e2 \ggg\!\!= \backslash x2 \rightarrow \\
\quad \mathit{return}\ (x1 - x2) \\
\mathit{eval2}'\ (\mathit{Div}\ e1\ e2) = \\
\quad \mathit{eval2}'\ e1 \ggg\!\!= \backslash x1 \rightarrow \\
\quad \mathit{eval2}'\ e2 \ggg\!\!= \backslash x2 \rightarrow \\
\quad \mathit{return}\ (x1\ /\ x2) \\
\mathit{eval2Top}'\ e = \mathit{fst}\ (\mathit{runState}\ (\mathit{eval2}'\ e)\ 0)
\end{array}
$$

Do notation

$$
\begin{aligned}
&\textbf{do}\ x1 \leftarrow e1 \\
&\quad x2 \leftarrow e2 \\
&\quad \ldots \\
&\quad xn \leftarrow en \\
&\quad f\ x1\ x2\ \ldots\ xn
\end{aligned}
$$

is syntactic sugar for

$$
\begin{aligned}
&e1 \ggg= \backslash x1 \rightarrow \\
&e2 \ggg= \backslash x2 \rightarrow \\
&\ldots \\
&en \ggg= \backslash xn \rightarrow \\
&f\ x1\ x2\ \ldots\ xn
\end{aligned}
$$

Evaluator #2, equivalently expressed with do notation

$$
\begin{aligned}
&\mathit{eval2}' :: \mathit{Expr} \to \mathit{State\ Int\ Double}\\
&\mathit{eval2}'\ (\mathit{Con}\ c) = \mathbf{do}\\
&\quad n \leftarrow \mathit{get}\\
&\quad \mathit{put}\ (n + 1)\\
&\quad \mathit{return}\ (\mathbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 3 \equiv 2\ \mathbf{then}\ 0\ \mathbf{else}\ c)\\
&\mathit{eval2}'\ (\mathit{Sub}\ e1\ e2) = \mathbf{do}\\
&\quad x1 \leftarrow \mathit{eval2}'\ e1\\
&\quad x2 \leftarrow \mathit{eval2}'\ e2\\
&\quad \mathit{return}\ (x1 - x2)\\
&\mathit{eval2}'\ (\mathit{Div}\ e1\ e2) =\\
&\quad x1 \leftarrow \mathit{eval2}'\ e1\\
&\quad x2 \leftarrow \mathit{eval2}'\ e2\\
&\quad \mathit{return}\ (x1\ /\ x2)
\end{aligned}
$$

The IO monad

A built-in monad used to perform real system I/O.

Supports operations like

$$getLine :: IO\ String$$
$$putStrLn :: String \rightarrow IO\ ()$$

etc.

The use of a monad ensures proper sequentialization, as we can never "escape" the IO monad![3]

[3] Technically, this is not true. There is a back door in the form of a function $unsafePerformIO :: IO\ a \rightarrow a$, contained in the module *System.IO.Unsafe*. But as the name suggests, this function should be used with care...

Monads in Haskell (post 2014)

A bit more heavy since the "Functor-Applicative-Monad" hierarchy:

class *Functor f* **where**
  $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
class *Functor f* $\Rightarrow$ *Applicative f* **where**
  $pure :: a \rightarrow f\ a$
  $(\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
class *Applicative m* $\Rightarrow$ *Monad m* **where**
  $return :: a \rightarrow m\ a$
  $(\ggg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
  $return = pure$

So to define an instance of Monad, you first need instances of Functor and Applicative.

Monads in Haskell (post 2014)

But instances of Functor and Applicative can always be retrofitted from a Monad instance:

instance *Functor M* **where**
  $fmap\ f\ xm = xm \ggg\!= return \circ f$
instance *Applicative M* **where**
  $pure = return$
  $fm\ \langle * \rangle\ xm = fm \ggg\!= \backslash f \to xm \ggg\!= return \circ f$

Combining monads

To define Evaluator #3, we implicitly used a *monad transformer*:

$$\textbf{newtype}\ \mathit{StateT}\ s\ m\ a = \mathit{StateT}\ \{\, \mathit{runStateT} :: s \rightarrow m\ (a, s)\}$$

Given a monad m representing some notion of computation (e.g., partiality or nondeterminism), $\mathit{StateT}\ s\ m$ defines a new monad with s state wrapped around an m-computation.

But it is not always clear how to combine monads.

More generally, the question of how to organize and reason about programs with side-effects remains an important open problem!