# MPRI 2.4

# “Side effects” in Programming (and the rest)

Gabriel Scherer

2021

## Introduction

## What this course is about

“Side effects”. “pure”, “impure”, “effectful”.

What do those terms mean? It depends!

Effects are subjective notions used to structure systems.

Consequences in

- programming language theory
- actual programming practice
- logic
- ...

## Circuit example

TODO

## Typed effects

```
t : (bool * int) list
```

We can view `t` as describing

- a value of type `(bool * int) list`,
- or a computation of type `bool * int` with some typed effect `_ list`
- or a computation of type `bool` in some typed effect `(_ * int) list`
- ...

The style of the author favors one interpretation.

## Example

(Inspired by Philip Wadler's Bastaad lecture notes, 1995)

Variation 0: a simple calculator.

```
type expr =
  | Int of int
  | Add of expr * expr

val eval : expr -> int
```

## Errors

Variation 1: `_ option` for division error

```
type expr = ... | Div of exp * exp

val eval : expr -> int option
```

## Errors

Variation 1: `_ option` for division error

```
type expr = ... | Div of exp * exp

val eval : expr -> int option
```

Then refactor the code with

```
val return : 'a -> 'a option
val bind : 'a option -> ('a -> 'b option) -> 'b option
```

## Ad break: binding operators

```
let<op> p = <def> in <body>
(* desugars into *)
( let<op> ) <def> (fun p -> <body>)
```

Go refactor the `option` evaluator with `( let* )` for `bind`.

Example:

```
val ( let* ) : 'a list -> ('a -> 'b list) -> 'b list
let ( let* ) li f = List.concat_map f li
let* x = [1; 10] in [x; x+1]
(* [1; 2; 10; 11] *)
```

Note: there is also a desugaring for simultaneous bindings:

```
let<op> p1 = <def1> and<op'> p2 = <def2> in <body>
(* desugars into *)
( let<op> ) (and<op'> <def1> <def2>) (fun (p1, p2) -> <body>)
```

Example:

```
val ( let+ ) : 'a list -> ('a -> 'b) -> 'b list
let ( let+ ) li f = List.map f li
let ( and+ ) li1 li2 = List.combine li1 li2
let+ x = [1; 3; 5] and+ y = [10; 20; 30] in x + y
(* [11; 23; 35] *)
```

## Logging work

Variation 2: `_ * count` for counting work

```
type count = Count of int
val eval : expr -> int * count
```

## Logging work

Variation 2: `_ * count` for counting work

```
type count = Count of int
val eval : expr -> int * count
```

Then refactor the code with

```
val return : 'a -> 'a * count
val ( let* ) : 'a * count -> ('a -> 'b * count) -> 'b * count

val tick : unit * count
```

## Reading from an environment

Variation 3: `Cfg.t -> _` to access a configuation

```
type expr = ... | Current_time

module Cfg : sig
  type t = {
    current_time : int;
    ...
  }
end

val current_time : Cfg.t -> int

val eval : expr -> Cfg.t -> int
```

## Reading from an environment

Variation 3: `Cfg.t -> _` to access a configuation

```
type expr = ... | Current_time

module Cfg : sig
  type t = {
    current_time : int;
    ...
  }
end

val current_time : Cfg.t -> int

val eval : expr -> Cfg.t -> int
```

Then refactor the code with

```
val return : 'a -> (Cfg.t -> 'a)
val ( let* ) : (Cfg.t -> 'a) -> ('a -> Cfg.t -> 'b) -> Cfg.t -> '
```

## State

Variation 4: `Rng.t -> _ * Rng.t` for random number generation.

```
type expr = .. | Random of int  (* Random n in [0; n] *)

module Rng : sig
  type t
  val init : int array -> t
  val next : ~max:int -> t -> int * t
end

val eval : expr -> Rng.t -> int * Rng.t
```

## State

Variation 4: `Rng.t -> _ * Rng.t` for random number generation.

```
type expr = .. | Random of int  (* Random n in [0; n] *)

module Rng : sig
  type t
  val init : int array -> t
  val next : ~max:int -> t -> int * t
end

val eval : expr -> Rng.t -> int * Rng.t
```

Then refactor the code with

```
type 'a with_rng = Rng.t -> 'a * Rng.t
val return : 'a -> 'a with_rng
val ( let* ) : 'a with_rng -> ('a -> 'b with_rng) -> 'b with_rng
```

Effects break some equational reasoning.

## Reordering

$$\begin{array}{l}\text{let}^\star\ x = d_1\ \text{in} \\ \text{let}^\star\ y = d_2\ \text{in} \\ e\{x, y\}\end{array} \quad \overset{?}{\simeq} \quad \begin{array}{l}\text{let}^\star\ y = d_2\ \text{in} \\ \text{let}^\star\ x = d_1\ \text{in} \\ e\{x, y\}\end{array}$$

Valid for Option, Count and Cfg, but not Rng

(Note: Count works because `( + )` is commutative.)

## (De)duplicating

$$\begin{array}{l} \text{let}^\star\ x = d\ \text{in} \\ \text{let}^\star\ y = d\ \text{in} \\ e\{x, y\} \end{array} \quad \overset{?}{\simeq} \quad \begin{array}{l} \text{let}^\star\ x = d\ \text{in} \\ e[y := x]\{x\} \end{array}$$

Valid for Option and Cfg, but not Count or Rng.

## Erasing

$$\begin{array}{l}\text{let}^\star\ x = d\ \text{in} \\ e\{\}\end{array} \quad \overset{?}{\simeq} \quad e\{\}$$

Valid for Cfg, but not Option or Count or Rng.

## Typed vs. untyped effects

An effect can be

- “typed”: tracked by the type system
- “untyped”: its usage in terms is not seen in the types

Untyped languages only have untyped effects.

Typed languages can have both.

In the previous examples, all effects were typed.

Example of untyped effects:

- Non-termination
  (in most programming languages; otherwise `nat -> _ option`.)
- OCaml and Haskell both offer untyped exceptions, for convenience – with regrets.

## Primitive vs. user-defined effects

The effects in the calculators were "user-defined",

we (the users) implemented them ourselves

A language and its standard library / built-in primitives

may also provide "primitive effects", available from scratch.

Primitive effects are often untyped (most programming languages),

but may also be typed (in Haskell: IO, etc., but not looping or exceptions).

Typed effects are generally better: easier reasoning.

But: effect typing at scale brings many usability issues.

(Current state-of-the-art: Koka, Frank)

## Direct vs. indirect style

"direct style" (for an effect): using the effect without ceremony

```
(* non-standard OCaml with a built-in non-determinism effect *)
let pythagorean_triples n =
  let a = in_interval 1 n in
  let b = in_interval a n in
  let c = in_interval a n in
  if not (a * a + b * b = c * c) then fail
  else (a, b, c)
```

"indirect style": using an effect with visible plumbing/encoding

```
let pythagorean_triples n =
  in_interval 1 n |> List.concat_map @@ fun a ->
  in_interval a n |> List.concat_map @@ fun b ->
  in_interval a n |> List.concat_map @@ fun c ->
  if not (a * a + b * b = c * c) then []
  else [ (a, b, c) ]
```

Direct-style can be just syntactic sugar:

```
let pythagorean_triples n =
  let* a = in_interval 1 n in
  let* b = in_interval a n in
  let* c = in_interval a n in
  if not (a * a + b * b = c * c) then fail
  else return (a, b, c)
```

Compare:

```
(* non-standard OCaml with a built-in non-determinism effect *)
let pythagorean_triples n =
  let a = in_interval 1 n in
  let b = in_interval a n in
  let c = in_interval a n in
  if not (a * a + b * b = c * c) then fail
  else (a, b, c)
```

## No Free Lunch

It is possible to mechanically translate a direct-style program into an indirect-style program.

This makes it "pure" (for this effect), therefore better?

```
let pythagorean_triples n =
  let a = in_interval 1 n in
  let b = in_interval a n in
  let c = in_interval a n in
  if not (a * a + b * b = c * c) then fail
  else (a, b, c)
```

```
let pythagorean_triples n =
  in_interval 1 n |> List.concat_map @@ fun a ->
  in_interval a n |> List.concat_map @@ fun b ->
  in_interval a n |> List.concat_map @@ fun c ->
  if not (a * a + b * b = c * c) then []
  else [ (a, b, c) ]
```

Stronger equational reasoning... on more complex code.

## Writing better code

"Unseeing" effects does not make them go away.

Recognizing effects will clarify its program structure,

help you find the right reasoning abstractions.

To get better code, write simpler code with less powerful effects.

Example: mutable state $\Rightarrow$ commutative, write-only state

Slogan: avoid accidental effects.

```
let map f li =
  let acc = ref [] in
  List.iter (fun x -> acc := f x :: !acc) li;
  List.rev !acc
```

## Effects in logic

Logic has many effects, for example:

- Axiom of choice.
- Excluded middle: $A \vee \neg A$.
- Duplication: $A \multimap A \otimes A$.

Step indexing: $[\![P]\!] := \mathbb{N} \to P$.

## Monads: programming

## Functor

```
val map : ('a -> 'b) -> 'a t -> 'b t

val ( let+ ) : 'a t -> ('a -> 'b) -> 'b t
```

```
map (fun x -> x) d = d
map f (map g d) = map (fun x -> f (g x)) d
```

$$\text{let}^{+}\ x = d \text{ in } x \quad \simeq \quad d$$

$$\text{let}^{+}\ y = (\text{let}^{+}\ x = d \text{ in } e_1) \text{ in } e_2\{y\} \quad \simeq \quad \text{let}^{+}\ x = d \text{ in let } y = e_1 \text{ in } e_2\{y\}$$

## Monad

```
val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t

val ( let* ) : 'a t -> ('a -> 'b t) -> 'b t
```

## Monad laws (1)

```
let* x = return v in m
=
let x = v in m
```

Example:

```
match Some v with
| None -> None
| Some x -> m
=
let x = v in m
```

## Monad laws (2)

```
let* x = m in return v
=
let+ x = m in v
```

Example:

```
 match m with
 | None -> None
 | Some x -> Some v
 =
 Option.map (fun x -> v) m
```

## Monad laws (3)

```
let* y = (let* x = mx in my) in m
=
let* x = mx in (let* y = my in m)
```

Example:

```
match
  (match mx with
   | None -> None
   | Some x -> my)
 with
   | None -> None
   | Some y -> m
 =
 match mx with
 | None -> None
 | Some x ->
   match my with
   | None -> None
   | Some y -> m
```

## Common monads

- partiality: Option $A := 1 + A$
- non-determinism: NonDet $A :=$ List $A$
- reading from $I$: Reader$_I$ $A := I \to A$
- writing to a monoid $O$: Writer$_O$ $A := A \times O$
- global state $S$: State$_S$ $A := S \to (A \times S)$
- input/output: IO $A :=$ State$_{\text{World}}$ $A$

## Monads represent effects

“Notions of computations and monads”, Eugenio Moggi, 1991.

We have seen how monads can be used to refactor effectful code.

Can we make the connection more precise?

## Monadic translation

Suppose we have a programming language $\lambda^{\text{eff}}$ with some untyped effect, represented by a typing judgment $\Gamma \vdash^{\text{eff}} e : A$.

If the built-in effects of $\lambda^{\text{eff}}$ can be represented by a monad $M$,
we can transform effectful $\lambda^{\text{eff}}$ terms $e : A$
into terms $\lfloor e \rfloor_M : M\ A$ in a pure calculus.

Monadic translation: compiling direct style into indirect style.

Theorem:

$$\Gamma \vdash^{\text{eff}} e : A \qquad\Longrightarrow\qquad \lfloor \Gamma \rfloor_M \vdash \lfloor e \rfloor_M : M\,\lfloor A \rfloor_M$$

## Effectful source language

$$\frac{}{\Gamma, x : A \vdash^{\text{eff}} x : A} \qquad \frac{\Gamma \vdash^{\text{eff}} e_1 : A \qquad \Gamma, x : A \vdash^{\text{eff}} e_2 : B}{\Gamma \vdash^{\text{eff}} \text{let } x = e_1 \text{ in } e_2 : B}$$

$$\frac{\Gamma, x : A \vdash^{\text{eff}} e : B}{\Gamma \vdash^{\text{eff}} \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash^{\text{eff}} e_1 : A \to B \qquad \Gamma \vdash^{\text{eff}} e_2 : A}{\Gamma \vdash^{\text{eff}} e_1\ e_2 : B}$$

$$\frac{\text{op}(f) : A_1 \to \cdots \to A_n \to B}{\Gamma \vdash^{\text{eff}} \text{op}(f) : A_1 \to \cdots \to A_n \to B}$$

Examples:

- op(fail) : $A$
- op(choose) : List $A \to A \to A$
- op(time) : $\mathbb{N}$
- op($mathsfrand$) : $\mathbb{N}$

## Translation: inputs

We assume that $(M, \text{return}, \text{let}^\star)$ forms a monad:

$$\text{return} \; : A \to M\,A \qquad\qquad (\text{let}^\star) : M\,A \to (A \to M\,B) \to M\,B$$

and we assume a translation of each operation supported by $M$:

$$\forall \text{op}(f) : A_1 \to \cdots \to A_n \to B, \qquad\qquad \lfloor \text{op}(f) \rfloor_M : A_1 \to \cdots \to A_n \to M\,B$$

Examples:

- $\lfloor \text{op}(\text{fail}) \rfloor_{\text{Option}} \; : \text{Option}\;A = \text{None}$
- $\lfloor \text{op}(\text{fail}) \rfloor_{\text{List}} \; : \text{List}\;A = []$
- $\lfloor \text{op}(\text{choose}) \rfloor_{\text{List}} \; : \text{List}\;A \to \text{List}\;A$
- $\lfloor \text{op}(\text{time}) \rfloor_{\text{Cfg}} \; : \text{Cfg.t} \to \mathbb{N}$
- $\lfloor \text{op}(\text{rand}) \rfloor_{\text{Rng}} \; : \text{Rng.t} \to \mathbb{N} \times \text{Rng.t}$

## Translation: types

Translation of types and contexts:

$$
\begin{aligned}
\lfloor \mathbb{N} \rfloor &:= \mathbb{N} \\
\lfloor A \to B \rfloor_M &:= \lfloor A \rfloor_M \to M \lfloor B \rfloor_M \\
\lfloor \Gamma, x : A \rfloor_M &:= \lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M
\end{aligned}
$$

Translation of judgments:

$$
\left\lfloor \Gamma \vdash^{\text{eff}} e : A \right\rfloor_M := \lfloor \Gamma \rfloor_M \vdash \lfloor e \rfloor_M : M \lfloor A \rfloor_M
$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\text{eff}} x : A \right\rfloor_M \; :=$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\text{eff}} x : A \right\rfloor_M \;:=\; \frac{}{\lfloor \Gamma \rfloor_M \vdash \text{return } x : M \lfloor A \rfloor_M}$$

$$\left\lfloor \Gamma \vdash^{\text{eff}} \text{op}(f) : A_1, \ldots, A_n \to B \right\rfloor_M :=$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\text{eff}} x : A \right\rfloor_M \quad := \quad \frac{}{\lfloor \Gamma \rfloor_M \vdash \text{return } x : M \lfloor A \rfloor_M}$$

$$\left\lfloor \Gamma \vdash^{\text{eff}} \text{op}(f) : A_1, \dots, A_n \to B \right\rfloor_M := \frac{}{\lfloor \Gamma \rfloor_M \vdash \lfloor \text{op}(f) \rfloor_M : \lfloor A_1 \rfloor_M, \dots, \lfloor A_n \rfloor_M \to M \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma, x : A \vdash^{\text{eff}} e : B}{\Gamma \vdash^{\text{eff}} \lambda x.e : A \to B} \right\rfloor_M \qquad :=$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\text{eff}} x : A \right\rfloor_M \quad := \quad \frac{}{\lfloor \Gamma \rfloor_M \vdash \text{return } x : M \lfloor A \rfloor_M}$$

$$\left\lfloor \Gamma \vdash^{\text{eff}} \text{op}(f) : A_1, \dots, A_n \to B \right\rfloor_M := \frac{}{\lfloor \Gamma \rfloor_M \vdash \lfloor \text{op}(f) \rfloor_M : \lfloor A_1 \rfloor_M, \dots, \lfloor A_n \rfloor_M \to M \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma, x : A \vdash^{\text{eff}} e : B}{\Gamma \vdash^{\text{eff}} \lambda x.e : A \to B} \right\rfloor_M \quad := \quad \frac{\lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M \vdash \lfloor e \rfloor_M : M \lfloor A \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \text{return } \lambda x.\lfloor e \rfloor_M : \lfloor A \rfloor_M M \lfloor A \rfloor_M}$$

$$\left\lfloor \frac{\Gamma \vdash^{\text{eff}} e_1 : A \qquad \Gamma, x : A \vdash^{\text{eff}} e_2 : B}{\Gamma \vdash^{\text{eff}} \text{let } x = e_1 \text{ in } e_2 : B} \right\rfloor_M \quad :=$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\mathrm{eff}} x : A \right\rfloor_M \;:=\; \frac{}{\lfloor \Gamma \rfloor_M \vdash \mathrm{return}\ x : M\, \lfloor A \rfloor_M}$$

$$\left\lfloor \Gamma \vdash^{\mathrm{eff}} \mathrm{op}(f) : A_1, \dots, A_n \to B \right\rfloor_M := \frac{}{\lfloor \Gamma \rfloor_M \vdash \lfloor \mathrm{op}(f) \rfloor_M : \lfloor A_1 \rfloor_M, \dots, \lfloor A_n \rfloor_M \to M\, \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma, x : A \vdash^{\mathrm{eff}} e : B}{\Gamma \vdash^{\mathrm{eff}} \lambda x.e : A \to B} \right\rfloor_M \;:=\; \frac{\lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M \vdash \lfloor e \rfloor_M : M\, \lfloor A \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \mathrm{return}\ \lambda x. \lfloor e \rfloor_M : \lfloor A \rfloor_M M\, \lfloor A \rfloor_M}$$

$$\left\lfloor \frac{\Gamma \vdash^{\mathrm{eff}} e_1 : A \qquad \Gamma, x : A \vdash^{\mathrm{eff}} e_2 : B}{\Gamma \vdash^{\mathrm{eff}} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : B} \right\rfloor_M \;:=$$

$$\frac{\lfloor \Gamma \rfloor_M \vdash \lfloor e_1 \rfloor_M : M\, \lfloor A \rfloor_M \qquad \lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M \vdash \lfloor e_2 \rfloor_M : M\, \lfloor B \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \mathrm{let}^{\star}\ x = \lfloor e_1 \rfloor_M\ \mathrm{in}\ \lfloor e_2 \rfloor_M : M\, \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma \vdash^{\mathrm{eff}} e_1 : A \to B \qquad \Gamma \vdash^{\mathrm{eff}} e_2 : A}{\Gamma \vdash^{\mathrm{eff}} e_1\ e_2 : B} \right\rfloor_M \;:=$$

## Translation: terms

$$\left\lfloor \Gamma, x : A \vdash^{\text{eff}} x : A \right\rfloor_M \quad := \quad \frac{}{\lfloor \Gamma \rfloor_M \vdash \text{return } x : M\, \lfloor A \rfloor_M}$$

$$\left\lfloor \Gamma \vdash^{\text{eff}} \text{op}(f) : A_1, \dots, A_n \to B \right\rfloor_M := \frac{}{\lfloor \Gamma \rfloor_M \vdash \lfloor \text{op}(f) \rfloor_M : \lfloor A_1 \rfloor_M, \dots, \lfloor A_n \rfloor_M \to M\, \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma, x : A \vdash^{\text{eff}} e : B}{\Gamma \vdash^{\text{eff}} \lambda x.e : A \to B} \right\rfloor_M \quad := \quad \frac{\lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M \vdash \lfloor e \rfloor_M : M\, \lfloor A \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \text{return } \lambda x. \lfloor e \rfloor_M : \lfloor A \rfloor_M M\, \lfloor A \rfloor_M}$$

$$\left\lfloor \frac{\Gamma \vdash^{\text{eff}} e_1 : A \qquad \Gamma, x : A \vdash^{\text{eff}} e_2 : B}{\Gamma \vdash^{\text{eff}} \text{let } x = e_1 \text{ in } e_2 : B} \right\rfloor_M \quad :=$$

$$\frac{\lfloor \Gamma \rfloor_M \vdash \lfloor e_1 \rfloor_M : M\, \lfloor A \rfloor_M \qquad \lfloor \Gamma \rfloor_M, x : \lfloor A \rfloor_M \vdash \lfloor e_2 \rfloor_M : M\, \lfloor B \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \text{let}^\star\ x = \lfloor e_1 \rfloor_M \text{ in } \lfloor e_2 \rfloor_M : M\, \lfloor B \rfloor_M}$$

$$\left\lfloor \frac{\Gamma \vdash^{\text{eff}} e_1 : A \to B \qquad \Gamma \vdash^{\text{eff}} e_2 : A}{\Gamma \vdash^{\text{eff}} e_1\ e_2 : B} \right\rfloor_M \quad :=$$

$$\frac{\lfloor \Gamma \rfloor_M \vdash \lfloor e_1 \rfloor_M : M\, \lfloor A \rfloor_M \to M\, \lfloor B \rfloor_M \qquad \lfloor \Gamma \rfloor_M \vdash \lfloor e_2 \rfloor_M : M\, \lfloor A \rfloor_M}{\lfloor \Gamma \rfloor_M \vdash \begin{array}{l} \text{let}^\star\ f = \lfloor e_1 \rfloor_M \text{ in} \\ \text{let}^\star\ x = \lfloor e_2 \rfloor_M \text{ in } f\ x \end{array} : M\, \lfloor B \rfloor_M}$$

## Monadic translation: conclusion

A compilation from direct style to indirect style.

A recipe to “see” effectful programs as pure programs.

An instance: the continuation-passing-style translation seen earlier.

$$\begin{aligned}
\mathsf{Cont}_A\ B &= ((B \to A) \to A) \\
\lfloor x \rfloor_{\mathsf{Cont}_A} &= \lambda k.\ k\ x \\
\lfloor \lambda x.t \rfloor_{\mathsf{Cont}_A} &= \lambda k.\ k\ (\lambda x. \lfloor t \rfloor_{\mathsf{Cont}_A}) \\
\lfloor t_1\ t_2 \rfloor_{\mathsf{Cont}_A} &= \lambda k.\ \lfloor t_1 \rfloor_{\mathsf{Cont}_A}\ (\lambda x_1.\ \lfloor t_2 \rfloor_{\mathsf{Cont}_A}\ (\lambda x_2.\ x_1\ x_2\ k)) \\
\lfloor \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \rfloor_{\mathsf{Cont}_A} &= \lambda k.\ \lfloor t_1 \rfloor_{\mathsf{Cont}_A}\ (\lambda x.\ \lfloor t_2 \rfloor_{\mathsf{Cont}_A}\ k)
\end{aligned}$$

Can be refined in many ways. For example, if the source language has typed effects, the translation can be refined with several monads.

For example: “Lightweight monadic programming in ML”,
Nikhil Swamy, Nataliya Guts, Daan Leijen, Michael Hicks, 2011

## Monads: denotational semantics

Denotational semantics: defining the meaning of programs (or proofs, etc) as mathematical objects.

Types as sets, as directed-complete partial orders, as game arenas...

Modern structuring approach: categories.

Category theory: the mathematical theory of composition.

(Started around Mac Lane in the 1940s, replaced universal algebra).

The minimum we need:

- reminder on categories, functors, natural transformations
- interpreting $\lambda$-terms in categories
- monads
- monadic interpretation of $\lambda$-terms
- Kleisli categories

## Reminder: Category

A category $\mathcal{C}$ is given by

- a set of objects $\mathrm{Obj}(\mathcal{C})$
  (we write $A \in \mathcal{C}$ for $A \in \mathrm{Obj}(\mathcal{C})$)
- for each pair of objects $A, B \in \mathrm{Obj}(\mathcal{C})$, a set of morphisms $\mathcal{C}(A, B)$
  (we write $f : A \to B$ for $f \in \mathcal{C}(A, B)$)

with a monoid-like structure on morphisms:

- each $A \in \mathcal{C}$ has an identity morphism $\mathrm{Id}_A : A \to A$
- each $f : A \to B$ and $g : B \to C$ have a composition $(f; g) : A \to C$
  (composition may be written $f; g$ or $g \circ f$)
- identity morphisms are identities for composition: for $f : A \to B$ we have $(\mathrm{Id}_A; f) = f = (f; \mathrm{Id}_B)$.

## Category: examples

Categories generalize sets: a set is a “discrete” category

(only identity morphisms).

Categories generalize ordered sets $(S, \leqslant)$: orders with non-trivial “justifications”.

Categories generalize monoids: a monoid is a single-object category.

Set-like categories: the category of sets (and functions), of (sets and) relations, of groups (and morphisms), etc. The category of (small) categories, of course.

Categories combinators, for example: product of categories $\mathcal{C} \times \mathcal{D}$.

Our working category: objects are types, morphisms are one-variable terms $A(B, :) = \{x : A \vdash e : B\}$

## Categories with structure

We define families of categories whose objects and morphisms come with additional structure.

Monoidal categories, cartesian categories, closed categories, categories with coproducts, etc.

(skipped in this course)

Denotational semantics of $\lambda$-calculi:

What is the minimal, natural categorical structure required to interpret terms as morphisms?

This meaning can then be instantiated to sets, domains, games, graphs, types...

Provided one has equipped them with the necessary structure.

A shared language for structure.

## Reminder: Functors

A functor $F : \mathcal{C} \to \mathcal{D}$ is given by:

- a function from $\mathrm{Obj}(\mathcal{C})$ to $\mathrm{Obj}(\mathcal{D})$, also written $F$;
- for each $A, B \in \mathrm{Obj}(\mathcal{C})$, a function from $\mathcal{C}(A, B)$ to $\mathcal{D}(F(A), F(B))$, also written $F$, that respects identities and compositions:
  - $F(\mathrm{Id}_A) = \mathrm{Id}_{F(A)}$
  - $F(f; g) = F(f); F(g)$

Instances:

- between sets: functions
- between ordered sets: monotone functions
- between graphs: graph morphisms
- between types: “positive”/covariant parametrized types ((List _), $(\_ \times B)$... not $(\_ \to B)$)
- from types to sets: compositional semantics
- ...

## Reminder: Natural transformations

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\theta : F \to G$ is given by a family of morphisms $\theta\{A\} : F\ A \to G\ A$ (for all $A \in \mathrm{Obj}(C)$) such that the following diagram commutes for any $f : A \to B$:

$$\begin{tikzcd}
F\ A \arrow[r, "F(f)"] \arrow[d, "\theta\{A\}"] & F\ B \arrow[d, "\theta\{B\}"] \\
G\ A \arrow[r, "G(f)"] & G\ B
\end{tikzcd}$$

For example,

## Reminder: Natural transformations

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\theta : F \to G$ is given by a family of morphisms $\theta\{A\} : F\ A \to G\ A$ (for all $A \in \mathrm{Obj}(C)$) such that the following diagram commutes for any $f : A \to B$:

$$\begin{tikzcd}
F\ A \arrow[r, "F(f)"] \arrow[d, "\theta\{A\}"] & F\ B \arrow[d, "\theta\{B\}"] \\
G\ A \arrow[r, "G(f)"] & G\ B
\end{tikzcd}$$

For example, head : $\forall\{A\}.\mathrm{List}\ A \to \mathrm{Option}\ A$ verifies:

## Reminder: Natural transformations

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\theta : F \to G$ is given by a family of morphisms $\theta\{A\} : F\ A \to G\ A$ (for all $A \in \mathrm{Obj}(C)$) such that the following diagram commutes for any $f : A \to B$:

$$\begin{tikzcd}
F\ A \arrow[r, "F(f)"] \arrow[d, "\theta\{A\}"] & F\ B \arrow[d, "\theta\{B\}"] \\
G\ A \arrow[r, "G(f)"] & G\ B
\end{tikzcd}$$

For example, head : $\forall\{A\}.\mathrm{List}\ A \to \mathrm{Option}\ A$ verifies:

$$\begin{tikzcd}
\text{List Int} \arrow[r, "\text{List.map string\_of\_int}"] \arrow[d, "\text{head}\{\text{Int}\}"] & \text{List String} \arrow[d, "\text{head}\{\text{String}\}"] \\
\text{Option Int} \arrow[r, "\mathit{Option.map\ string\_of\_int}"] & \text{Option String}
\end{tikzcd}$$

## Example of structure: monoidal category

The only additional structure we see in this course – it helps to define monads.

A monoidal category $\mathcal{C}$ is a category $\mathcal{C}$ equipped with

- a functor $\otimes : \mathcal{C} \times \mathcal{C} \to C$ – the product
- an object $1_{\mathcal{C}} \in \mathcal{C}$ – the unit object
- natural isomorphisms (two inverse natural transformations)
  - $A \otimes (B \otimes C) \simeq (A \otimes B) \otimes C$
  - $1 \otimes A \simeq A \simeq A \otimes 1$

  with some coherence diagrams (skipped in this course).

## Assumptions

We mentioned that one-variable terms $x : A \vdash e : B$ form a category.

How to represent multi-variable terms $\Gamma \vdash e : B$?

Answer: assume some product structure $A \otimes B$ on our categories.

$x_\Gamma : \bigotimes_{x:A \in \Gamma} A \vdash e : B$

Various notions of products exist
(monoidal product, cartesian product..).

They correspond to different structural rules on context.
(Ordered, linear, etc.)

In this course we skip this discussion
and require the two following operations:

- projection: $\mathrm{proj}_{(x:A) \in \Gamma} : (\bigotimes_{x_i:A_i \in \Gamma} A_i) \to A$
- input duplication: $\mathrm{keep}_A \ (f : A \to B) : A \to A \otimes B$

## Interpretation

We interpret derivations $\Gamma \vdash e : A$ as morphisms $[\![e]\!] : [\![\Gamma]\!] \to [\![A]\!]$.

Interpreting type formers ($A \times B$, $A \to B$...) requires extra categorical structure.

(Skipped in this lecture.)

$$
\begin{array}{lll}
[\![A]\!] & & \text{assumed} \\
[\![\Gamma]\!] & := & \bigotimes_{(x:A)\in\Gamma} [\![A]\!] \\
\left[\!\!\left[\dfrac{(x:A)\in\Gamma}{\Gamma \vdash x : A}\right]\!\!\right] & := & \mathrm{proj}_{(x:A)\in\Gamma} : [\![\Gamma]\!] \to [\![A]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : B}\right]\!\!\right] & := &
\end{array}
$$

## Interpretation

We interpret derivations $\Gamma \vdash e : A$ as morphisms $[\![e]\!] : [\![\Gamma]\!] \to [\![A]\!]$.

Interpreting type formers ($A \times B$, $A \to B$...) requires extra categorical structure.

(Skipped in this lecture.)

$$\begin{array}{lcl}
[\![A]\!] & & \text{assumed} \\
[\![\Gamma]\!] & := & \bigotimes_{(x:A)\in\Gamma} [\![A]\!] \\
\left[\!\!\left[\dfrac{(x:A)\in\Gamma}{\Gamma \vdash x : A}\right]\!\!\right] & := & \mathrm{proj}_{(x:A)\in\Gamma} : [\![\Gamma]\!] \to [\![A]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : B}\right]\!\!\right] & := & (\mathrm{keep}_{[\![\Gamma]\!]}\ [\![e_1]\!]; [\![e_2]\!]) : [\![\Gamma]\!] \to [\![B]\!]
\end{array}$$