MPRI 2.4

“Side effects” in Programming (and the rest)

Gabriel Scherer

Inria informatics mathematics

2021

Introduction
Example
Reasoning about effects
Flavours of effects
Against purity

Monads

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.

1 Introduction

Example: Five Easy Pieces on a calculator

Reasoning about effects

Flavours of effects

Against purity

2 Monads

Introduction
Example
Reasoning about effects
Flavours of effects
Against purity
Monads

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 Some [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
```

1 Introduction
Example: Five Easy Pieces on a calculator
Reasoning about effects
Flavours of effects
Against purity

2 Monads

Effects break some equational reasoning.

Reordering

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

Valid for Option, Count and Cfg, but not Rng

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

(De)duplicating

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

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

Erasing

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

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

1 Introduction

Example: Five Easy Pieces on a calculator

Reasoning about effects

Flavours of effects

Against purity

2 Monads

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)
```

1 Introduction

Example: Five Easy Pieces on a calculator

Reasoning about effects

Flavours of effects

Against purity

2 Monads

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 ⇒ commutative, write-only state

Slogan: avoid accidental effects.

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

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 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
```