Chapter 10

First-class effects

10.1 Effects in OCaml

Most of the programs and functions we have considered so far are *pure*: they turn parameters into results, leaving the world around them unchanged. However, most useful programs and many useful functions are not pure: they may modify memory, write to or read from files, communicate over a network, raise exceptions, and perform many other effects. Practical programming languages must support some way of performing effects, and OCaml has support for writing impure functions in the form of language features for mutable memory, for raising and handling exceptions and for various forms of I/O.

However, there are other useful effects besides those that are provided in OCaml, including checked exceptions (as found in Java), continuations (as found in Scheme), nondeterminism (as found in Prolog) and many more. How might we write programs that make use of effects such as these without switching to a different language?

One approach to programming with arbitrary effects is to introduce an interface for describing *computations* — i.e. expressions that perform effects when evaluated. Programming with computations will allow us to simulate arbitrary effects, even those not provided by the language.

The role of let A reasonable starting point for building an interface for computations is to look at how impure programs are written in OCaml. OCaml provides a number of constructs and functions for performing effects — **try** and raise for dealing with exceptions, ref, := and ! for programming with mutable state, and so on. However, in addition to these effect-specific operations, every effectful program involves some kind of *sequencing* of effects. Since the order of evaluation of expressions affects the order in which effects are performed (and thus the observable behaviour of a program), it is crucial to have some way of specifying that one expression should be evaluated before another.

For example, consider the following OCaml expression:

```
f (g ()) (h ())
```

If the functions g and h have observable effects then the behaviour of the program when the first argument g () is evaluated first is different from the behaviour when the second argument h () is evaluated first. In OCaml the order of evaluation of function arguments is unspecified[1], so the behaviour of the program in different environments may vary.

The order of evaluation of two OCaml expressions can be specified using **let**: in the following expression e_1 is always evaluated before e_1:

```
let x = e₁ in e₂
```

To the other functions of **let** — local definitions, destructuring values, introducing polymorphism, etc. — we may therefore add *sequencing*.

10.2 Monads

This sequencing behaviour of **let** can be captured using a *monad*. Monads have their roots in abstract mathematics, but we will be treating them simply as a general interface for describing computations.

10.2.1 The monad interface

The monad interface can be defined as an OCaml signature (Section 6.1.1):

```
module type MONAD =
sig
  type 'a t
  val return : 'a → 'a t
  val (≫=)   : 'a t → ('a → 'b t) → 'b t
end
```

The type t represents the type of computations. A value of type 'a t represents a computation that performs some effects and then returns a result of type 'a; it corresponds to an expression of type 'a in an impure language such as OCaml.

The function return constructs trivial computations from values. A computation built with return simply returns the value used to build the computation, much as some expressions in OCaml simply evaluate to a value without performing any effects.

The ≫= operator (pronounced "bind") combines computations. More precisely, ≫= builds a computation by combining its left argument, which is a computation, with its right argument, which is a function that builds a computation. As the type suggests, the result of the first argument is passed to the second argument; the resulting computation performs the effects of both arguments.

[1]In fact, many constructs in OCaml have unspecified evaluation order, and in some cases the evaluation order differs across the different OCaml compilers. For example, here is a program that prints "ocamlc" or "ocamlopt" according to which compiler is used:

```
let r = ref "ocamlc" in print_endline (snd ((r := "ocamlopt"), !r))
```

The ≫= operator sequences computations, much as **let** sequences the evaluation of expressions in an impure language.

Here is an OCaml expression that sequences the expressions e_1 and e_2, binding the result of e_1 to the name x so that it can be used in e_2:

```
let x = e₁ in e₂
```

And here is an analogous computation written using a monad:

```
e₁ ≫= fun x → e₂
```

10.2.2 The monad laws

In order to be considered a monad, an implementation of the MONAD signature must satisfy three laws. The first law says that return is a kind of left unit for ≫=:

```
  return v ≫= k     ≡     k v
```

The second law says that return is a kind of right unit for ≫=:

```
  m ≫= return     ≡     m
```

The third law says that bind is associative.

```
  (m ≫= f) ≫= g     ≡   m ≫= (fun x → f x ≫= g)
```

The higher-order nature of ≫= makes these laws a little difficult to read and remember. If we translate them into the analogous OCaml expressions things become easier. The first law (a β rule for **let**) then says that instead of using **let** to bind a value we can substitute the value in the body:

```
  let x = v in e     ≡     e[x:=v]
```

The second law (an η rule for **let**) says that a **let** binding whose body is simply the bound variable can be simplified to the right-hand side:

```
  let x = e in x     ≡     e
```

The third law (a commuting conversion for **let**) says that nested **let** bindings can be unnested:

```
  let x = (let y = e₁ in e₂) in e₃
                 ≡
  let y = e₁ in let x = e₂ in e₃
```

(assuming that y does not appear in e_3.)

Top-level functions for MONAD Since there are many different instances of the MONAD interface, it is convenient to define top-level functions that take an implicit MONAD argument:

```
let return {M:MONAD} x = M.return x
let (≫=) {M:MONAD} m k = M.(≫=) m k
```

Using these functions we can write code that works for any instance of MONAD.

10.2.3 Example: a state monad

The monad interface is not especially useful by itself, but we can make it more useful by adding operations that perform particular effects. Here is an interface STATE, which extends MONAD with operations for reading and updating a single reference cell:

```
module type STATE =
sig
  type state
  type 'a t
  module Monad : MONAD with type 'a t = 'a t
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end
```

To the type and operations of MONAD, STATE adds a type state denoting the type of the cell, and operations get and put for reading and updating the cell. The types of get and put suggest how they behave: get is a computation without parameters that returns a result of type state — that is, the contents of the cell; put is a computation which is parameterised by a state value with which it updates the cell, and which returns unit. Since the type t of computations is an abstract type we also add a destructor function runState to allow us to *run* computations. The runState function is parameterised by the initial state and it returns both the final state and the result of running the computation.

The following implicit instance converts STATE instances to MONAD instances, making it possible to use the top-level return and ≫= functions with instances of STATE:

```
implicit module Monad_of_state{S:STATE} = S.Monad
```

It's useful to define a similar implicit instance each time we define a new extension of MONAD, but we won't write out the instances every time.

The STATE interface makes it possible to express a variety of computations. For example, here is a simple computation that retrieves the value of the cell and then stores an incremented value:

```
get ≫= fun s →
put (s + 1)
```

We might write an analogous program using OCaml's built-in reference type as follows:

```
let s = !r in
    r := (s + 1)
```

This example shows how to *use* the state monad. How might we *implement* STATE? As we shall see, the primary consideration is to find a suitable definition for the type t; once t is defined the definitions of the other members of the interface typically follow straightforwardly. The type of runState suggests a definition: a STATE computation may be implemented as a function from an initial state to a final state and a result:

```
type 'a t = state → state * 'a
```

Then return is a function whose initial and final states are the same:

```
val return : 'a → 'a t
let return v s = (s, v)
```

and ≫= is a function that uses the final state s' of its first argument as the initial state of its second argument:

```
val (≫=)  : 'a t → ('a → 'b t) → 'b t
let (≫=) m k s = let (s', a) = m s in k a s'
```

The get and put functions are even simpler. We can define get as a function that leaves the state unmodified, and also returns it as the result:

```
val get : state t
let get s = (s, s)
```

and put as a function that ignores the initial state, replacing it with the value supplied as an argument:

```
val put : state → unit t
let put s' _ = (s', ())
```

Here is a complete definition for an implementation of STATE. We define it as a functor (Section 7.1.2) so that we can abstract over the state type:

```
module State (S : sig type t end) = struct
  type state = S.t
  type 'a t = state -> state * 'a
  module Monad = struct
    type 'a t = state → state * 'a
    let return v s = (s, v)
    let (≫=) m k s = let s', a = m s in k a s'
  end
  let get s = (s, s)
  let put s' _ = (s', ())
  let runState m init = m init
end
```

10.2.4 Example: fresh names

How might we use State to write an effectful function? Let's consider a function that traverses trees, replacing the label at each branch with a fresh name. Here is our definition of a tree type:

```
type 'a tree =
    Empty : 'a tree
  | Tree : 'a tree * 'a * 'a tree → 'a tree
```

In order to use the State monad we must instantiate it with a particular state type. We'll use int, since a single int cell is sufficient to support the fresh name generation effect

```
module IState = State (struct type t = int end)
```

We can define the fresh_name operation as a computation returning a string in the IState monad:

```
let fresh_name : string IState.t =
  get          >>= fun i →
  put (i + 1) >>= fun () →
  return (Printf.sprintf "x%d" i)
```

The fresh_name computation reads the current value i of the state using get, then uses put to increment the state before returning a string constructed from i. Using fresh_name we can define a function label_tree that traverses a tree, replacing each label with a fresh name:

```
let rec label_tree : 'a.'a tree → string tree IState.t =
  function
    Empty → return Empty
  | Tree (l, v, r) →
    label_tree l >>= fun l →
    fresh_name   >>= fun name →
    label_tree r >>= fun r →
    return (Tree (l, name, r))
```

Labelling an empty tree is trivial, since there are no labels. Labeling a branch involves first labeling the left subtree, then generating a fresh name for the label, then labeling the right subtree, and finally constructing a new node from the labeled subtrees and the fresh name.

It is instructive to see what happens when we inline the definitions of the type and operations of the IState monad: get, put, >>= and return. After reducing the resulting applications we are left with the following:

```
let rec label_tree : 'a.'a tree → int → int * string tree =
  function
    Empty → (fun s → (s,Empty))
  | Tree (l, v, r) →
     fun s0 →
     let (s1, l) = label_tree l s0 in
     let (s2, n) = fresh_name s1 in
     let (s3, r) = label_tree r s2 in
       (s3, Tree (l, n, r))
```

Exposing the plumbing in this way allows us to see how computations in the state monad are executed. Each computation — label_tree l, fresh_name s1 etc. — is a function which receives the current value of the state and which returns a pair of the updated state along with a result. We could, of course, have written the code in this state-passing style in the first place instead of using monads, but passing state explicitly has a number of disadvantages: it is easy to inadvertently pass the wrong state value to a sub-computation, and it is hard to change the program to incorporate other effects.

10.2.5 Example: an exception monad

Let's consider a second extension of the MONAD interface that adds operations for raising and handling exceptions:

```
module type ERROR = sig
  type error
  type 'a t
  module Monad : MONAD with type 'a t = 'a t
  val raise : error → 'a t
  val _try_ : 'a t → (error → 'a) → 'a
end
```

The ERROR signature extends MONAD with a type error of exceptions and two operations. The first operation, raise, is parameterised by an exception and builds a computation that does not return a result, as indicated by the polymorphic result type. The second operation, _try_, is a destructor for computations that can raise exceptions. We might write

```
_try_ c
 (fun exn → e')
```

to run the computation c, returning either the result of c or, if c raises an exception, the result of evaluating e' with exn bound to the raised exception.

How might we implement ERROR? As before, we begin with the definition of the type t. There are two possible outcomes of running an ERROR computation, so we define t as a variant type with a constructor for representing a computation that returns a value and a constructor for representation a computation that raises an exception:

```
type 'a t =
   Val : 'a → 'a t
 | Exn : error → 'a t
```

It is then straightforward to define an implementation of Error. As with State, we define Error as a functor to support parameterisation by the exception type:

```
module Error (E: sig type t end) = struct
  type error = E.t
  module Monad = struct
    type 'a t =
       Val : 'a → 'a t
     | Exn : error → 'a t
    let return v = Val v
    let (>>=) m k =
      match m with
      | Val v → k v
      | Exn e → Exn e
  end
  let raise e = Exn e
  let _try_ m catch =
    match m with
    | Val v → v
    | Exn e → catch e
end
```

The implementations of return and raise are straightforward: return constructs a computation that returns a value, while raise constructs a computation that raises an exception. The behaviour of >>= depends on its first argument. If the first argument is a computation that returns a value then that value is passed to the second argument and the computation continues. If, however, the first argument is a computation that raises an exception then the result of $\gg=$ is the same computation. That is, the first exception raised by a computation in the error monad aborts the whole computation. The _try_ function runs a computation in the error monad, either returning the value or passing the raised exception to the argument function catch as appropriate.

Like the state monad, the error monad makes it possible to express a wide variety of computations. For example, we can write an analogue of the find function from the standard OCaml List module. The find function searches a list for the first element that matches a user-supplied predicate. Here is a definition of find:

```
let rec find p = function
   [] → raise Not_found
| x :: _ when p x → x
| _ :: xs → find p xs
```

If no element in the list matches the predicate then find raises the exception Not_found. Here is the type of find:

```
val find : ('a → bool) → 'a list → 'a
```

We might read the type as follows: find accepts a function of type 'a → bool and a list with element type 'a, and *if it returns a value*, returns a value of type 'a. There is nothing in the type that mentions that find can raise Not_found , since the OCaml type system does not distinguish functions that may raise exceptions from functions that always return successfully.

In order to implement an analogue of find using the ERROR interface we must first instantiate the functor, specifying the error type:

```
module Error_exn = Error(struct type t = exn end)
```

We can then implement the function as a computation in the Error_exn monad:

```
let rec findE p = Error_exn.(function
   [] → raise Not_found
| x :: _ when p x → return x
| _ :: xs → findE p xs)
```

The definition of findE is similar to the definition of find, but there is one difference: since findE builds a computation in a monad, we must use return to return a value.

Here is the type of findE:

```
val findE : ('a → bool) → 'a list → 'a Error_exn.t
```

The type tells us that findE accepts a function of type 'a → bool and a list with element type 'a, just like find. However, the return type is a little more informative: it tells us that findE builds a computation in the Error_exn monad that when run will either raise an exception or return a value of type 'a.

10.3 Monads and higher-order effects

The examples of monadic computations that we've seen so far have been fairly simple. However, the higher-order nature of the ≫= operator makes the monad interface powerful enough to express a wide variety of computations.

Like OCaml's built-in effects, monadic effects are *dynamic*, in the sense that the result of one computation can be used to build subsequent computations. For example, here is a fragment of OCaml that calls a (presumably effectful) function f and passes the result to a second function g:

```
let x = f () in
let y = g x in
   ...
```

and here is an analogous computation written using monads:

```
f ≫= fun x →
g x ≫= fun y →
   ...
```

This is a kind of first-order dynamic dependence, in which results from one computation appear as parameters to another. A second form of dynamic dependence allows the computations themselves, not just their parameters, to be determined by earlier computations. Here is a second OCaml fragment which defines a function uncurry that converts a curried function into a function on pairs:

```
let uncurry f (x,y) =
   let g = f x in
   let h = g y in
     h
```

and here is an analogous computation written using monads:

```
let uncurryM f (x, y) =
   f x ≫= fun g →
   g y ≫= fun h →
     return h
```

In both cases the function g used for the second subcomputation is computed by f x.

It is clear that the monad interface offers a great deal of flexibility to the user. However, by the same token it demands a great deal from the implementer. As we shall see, there are situations where monads are *too* powerful, and both user and implementer are better served by a more restrictive interface.

10.4 Applicatives

Applicatives[2] offer a second interface to effectful computation that is less powerful and therefore, from a certain perspective, more general than monads.

[2]The full name for "applicative" is for "applicative functor", but we'll stick with the shorter name. Several of the papers in the further reading section (page 157) use the name "idioms" under which applicatives were originally introduced.

10.4.1 Computations without dependencies

As we have seen, computations constructed using the MONAD interface correspond to the sort of computations that we can write with **let** ... **in** in OCaml (Section 10.1). Like **let**, the monadic $\gg=$ operation both sequences computations and makes the result of one computation available for constructing other computations (Section 10.3). However, **let** ... **in** is not always the most appropriate construct for combining computations in OCaml. In particular, if there are no dependencies between two expressions e_1 and e_2 then it is sometimes more appropriate to use the less-powerful construct **let** ... **and**. For example, when reading the following OCaml fragment the reader might wonder where the variable b on the second line is bound. Since the variables introduced by first line are in scope in the second line the reader must scan both the first line and the surrounding environment to find the nearest binding for b.

```
let x = f a in
let y = g b in
  ...
```

Since the variable x bound by the first line is not used in the second line the code can be rewritten to use the less-powerful binding construct **let** ... **and**[3]:

```
let x = f a
and y = g b
 in ...
```

Now it is immediately clear that none of the variables bound with **let** are used before the **in** on the last line, easing the cognitive burden on the reader.

10.4.2 The applicative interface

As we have seen, there is a correspondence between computations that use the MONAD interface and programs written using **let** ... **in**. The APPLICATIVE interface captures computations that have no interdependencies between them, in the spirit of **let** ... **and**.

Here is the interface for applicatives:

```
module type APPLICATIVE =
sig
  type 'a t
  val pure : 'a → 'a t
  val (<*>) : ('a → 'b) t → 'a t → 'b t
end
```

Comparing APPLICATIVE with MONAD (Section 10.2.1) reveals a number of minor differences — the operations are called pure and <*> (pronounced "apply") rather than return and $\gg=$, and the function argument to <*> comes first rather than second — and one significant difference. Here is the type of $\gg=$:

[3]Unfortunately the OCaml definition leaves the evaluation order of expressions bound with **let** ... **and** unspecified, so we must also consider whether we are happy for the two lines to be executed in either order. This is rather an OCaml-specific quirk, though, and does not affect the thrust of the argument, which is about scope, not evaluation order.

```
'a t → ('a → 'b t) → 'b t
```

and here is the type of <∗>, with the order of arguments switched to ease comparison:

```
'a t → ('a → 'b) t → 'b t
```

As the types show, the arguments to ≫= are a computation and a function that constructs a computation, allowing ≫= to pass the result of the computation as argument to the function. In contrast, the arguments to <∗> are two computations, so <∗> cannot pass the result of one computation to the other. The different types of ≫= and <∗> result in a significant difference in power between monads and applicatives, as we shall see.

10.4.3 Applicative normal forms

There are typically many ways to write any particular computation. For example, if we would like to call three functions f, g and h, and collect the results in a tuple then we might write either of the following equivalent programs:

```
let (x, y) =
    let x = f ()
    and y = g () in
      (x, y)
and z = h ()
 in (x, y, z)
```

```
let x = f ()
and (y, z) =
    let y = g ()
    and z = h ()
 in (x, y, z)
```

In this case it is fairly easy to see that the programs are equivalent. For situations where determining equivalence is not so easy it is useful to have a way of translating programs into a *normal form* — that is, a syntactically restricted form into which we can rewrite programs using the equations of the language. If we have a normal form then checking equivalence of two programs is a simple matter of translating them both into the normal form then comparing the results for syntactic equivalence.

For programs written with **let** ... **and** we might use a normal form that is free from nested **let**. We can rewrite both the above programs into the following form:

```
let x = f ()
and y = g ()
and z = h ()
 in (x, y, z)
```

Applicative computations also have a normal form. Every applicative computation is equivalent to some computation of the following form:

pure f <∗> c_1 <∗> c_2 <∗> ... <∗> c_n

where $c_1, c_2, \ldots, c_n$ are primitive computations that do not involve the computation constructors pure and <∗>.

10.4.4 The applicative laws and normalization

There are four laws (equations) which implementations of APPLICATIVE must satisfy. These four laws are sufficient to rewrite any applicative computation into the normal form of Section 10.4.3.

The first applicative law says that pure is a homomorphism for application.

```
  pure (f v)   ≡   pure f <*> pure v
```

The second law says that a lifted identity function is a left unit for applicative application.

```
  u   ≡   pure id <*> u
```

The third law says that nested applications can be flattened using a lifted composition operation.

```
  u <*> (v <*> w)   ≡   pure compose <*> u <*> v <*> w
```

(Here compose is defined as **fun** f g x → f (g x).) The fourth law says that pure computations can be moved to the left or right of other computations.

```
  v <*> pure x   ≡   pure (fun f → f x) <*> v
```

In summary, the laws make it possible to introduce and eliminate pure computations, and to flatten nested computations, allowing every computation to be rearranged into the normal form of Section 10.4.3, which consists of an unnested application with a single occurrence of pure.

Let's look at an example. The following computation is not in normal form, since there are two uses of pure, and a nested <*>:

```
pure (fun (x,y) z → (x, y, z))
 <*> (pure (fun x y → (x, y)) <*> f <*> g)
 <*> h
```

We can flatten the nested applications a little by using the third applicative law:

```
pure compose
 <*> pure (fun (x,y) z → (x, y, z))
 <*> (pure (fun x y → (x, y)) <*> f)
 <*> g
 <*> h
```

The adjacent pure computations can be coalesced using the first law:

```
pure (compose (fun (x,y) z → (x, y, z)))
 <*> (pure (fun x y → (x, y)) <*> f)
 <*> g
 <*> h
```

The remaining nested application can be flattened using the third law:

```
pure compose
 <*> pure (compose (fun (x,y) z → (x, y, z)))
 <*> pure (fun x y → (x, y))
 <*> f
 <*> g
 <*> h
```

We now have three adjacent pure computations that can be combined using the first law:

```
pure ((compose (compose (fun (x,y) z → (x, y, z)))) (fun x y → (x, y))
    )
 <*> f
 <*> g
 <*> h
```

Expanding the definition of compose and beta-reducing gives us the following normal form term:

```
pure (fun x y z → (x, y, z))
 <*> f
 <*> g
 <*> h
```

10.4.5 Applicatives and monads

There is a close relationship between applicatives and monads, which can be expressed as a functor:

```
implicit module Applicative_of_monad {M:MONAD} :
  APPLICATIVE with type 'a t = 'a M.t =
struct
  type 'a t = 'a M.t
  let pure = M.return
  let (<*>) f p =
     M.(f >>= fun g →
         p >>= fun q →
         return (g q))
end
```

The Applicative_of_monad functor builds an implementation of the APPLICATIVE interface from an implementation of the MONAD interface, preserving the definition of the type t. The definition of pure is trivial; all the interest is in the definition of <*>, which is defined in terms of both >>= and return. First >>= extracts the results from the two computations which are arguments to <*>; next, these results are combined by applying the first result to the second result; finally, return turns the result of the application back into a computation. We might consider this definition of <*> in terms of >>= as analogous to the way that computations written using **let** ... **and** can be written using **let** ... **in**. For example, we might rewrite the following program

```
let g = f
and q = p
  g q
```

as

```
let g = f in
let q = p in
  g q
```

(provided g does not appear in p, which we can always ensure by renaming the variable.)

The Applicative_of_monad functor shows how we might rewrite computations which use the applicative operations as computations written in terms of return and >>=. For example, here is the normalised applicative computation from Section 10.4.4:

```
pure (fun x y z → (x, y, z)) <*> f <*> g <*> h
```

Substituting in the definitions of pure and <*> from Applicative_of_monad gives us the following monadic computation:

```
  ((return (fun x y z → (x, y, z)) >>= fun u →
   f >>= fun v →
   return (u v)) >>= fun w →
   g >>= fun x →
   return (w x)) >>= fun y →
  h >>= fun z →
  return (y z)
```

This is not as readable as it might be, but we can use the monad laws to reassociate the >>= operations and eliminate the multiple uses of return, resulting in the following term:

```
f >>= fun e →
g >>= fun x →
h >>= fun z →
return (e, x, z)
```

10.4.6 Example: the state applicative

As we have seen, we can use the Applicative_of_monad functor to turn applicative computations into monadic computations. Viewing things from the other side, we can also use Applicative_of_monad to turn implementations of MONAD into implementations of APPLICATIVE. For example, we can build an applicative from the State monad:

```
module StateA(S : sig type t end) :
sig
  type state = S.t
  type 'a t
  module Applicative : APPLICATIVE with type 'a t = 'a t
  val get : state t
  val put : state → unit t
  val runState : 'a t → state → state * 'a
end =
struct
  type state = S.t
  module StateM = State(S)
  type 'a t = 'a StateM.t
  module Applicative =
     Applicative_of_monad{StateM.Monad}
  let (get, put, runState) = StateM.(get, put, runState)
end
```

Besides the monad type and operations we must transport the additional elements — the state type and the operations (get, put and runState — to the new interface.

10.4.7 Example: fresh names

We have converted the State monad to an corresponding applicative (Section 10.4.6). Can we write an applicative analogue to the label_tree function of Section 10.2.4?

Unfortunately, the applicative interface is not sufficiently powerful to write a computation that behaves like label_tree. In fact, we cannot even write the operation fresh_name. Here is the definition of fresh_name again:

```
let fresh_name : string IState.t =
  get           >>= fun i ->
  put (i + 1) >>= fun () ->
  return (Printf.sprintf "x%d" i)
```

The crucial difficulty is the use of the result i of the computation get in constructing the parameter to put. It is precisely this kind of dependency that the monadic >>= supports and the applicative <*> does not.

Instead of defining a fresh name computation using primitive computations get and put, we must make fresh_name itself a primitive computation in the applicative, where we have the full power of the underlying monad available:

```
module NameA :
sig
  type 'a t
  module Applicative : APPLICATIVE with type 'a t = 'a t
  val fresh_name : string t
  val run : 'a t -> 'a
end =
struct
  module M = State(struct type t = int end)
  type 'a t = 'a M.t
  module Applicative Applicative_of_monad(M)
  let fresh_name = M.(
    get           >>= fun i ->
    put (i + 1) >>= fun () ->
    return (Printf.sprintf "x%d" i))
  let run a = let _, v = M.runState a ~init:0 in v
end
```

Once we have defined fresh_name it is straightforward to write an applicative version of label_free:

```
let rec label_tree : 'a tree -> string tree NameA.t =
  function
    Empty -> pure Empty
  | Tree (l, v, r) ->
    pure (fun l name r -> Tree (l, name, r))
      <*> label_tree l
      <*> fresh_name
      <*> label_tree r
```

Comparing this definition with the monadic implementation of Section 10.2.4 reveals a difference in style. While the monadic version has an imperative feel, with the result of each computation bound in sequence, the applicative version retains the functional programming style, with a single pure function on the left of the computation applied to a colllection of arguments.

10.4.8 Composing applicatives

Section 10.4.5 shows how we can build applicative implementations from monad implementations. Another easy way to obtain new applicative implementations is to compose two existing applicatives. Unlike monads (Exercise 6), the composition of any two applicatives produces a new applicative implementation. We can define the composition as a functor:

```
module Compose (F : APPLICATIVE)
               (G : APPLICATIVE) :
  APPLICATIVE with type 'a t = 'a G.t F.t =
struct
  type 'a t = 'a G.t F.t
  let pure x = F.pure (G.pure x)
  let (<*>) f x = F.(pure G.(<*>) <*> f <*> x)
end
```

The type of the result of the Compose functor is built by composing the type constructors of the input applicatives F and G. Similarly, pure is defined as the composition of F.pure and G.pure. The definition of <*> is only slightly more involved. First, G's <*> function is lifted into the result applicative by applying F.pure. Second, this lifted function is applied to the arguments f and x using F's <*>. It is not difficult to verify that the result satisfies the applicative laws so long as F and G do (Exercise 8).

10.4.9 Example: the dual applicative

As we have seen (p144), OCaml's **let** ... **and** construct leaves the order of evaluation of the bound expressions unspecified. This underspecification is possible because of the lack of dependencies between the expressions; if an expression e_2 uses the value of another expression e_1, then it is clear that e_1 must be evaluated before e_2 in an eager language such as OCaml.

The applicative interface, which offers no way for one computation to depend upon the result of another, suggests a similar freedom in the order in which computations are executed. However, unlike OCaml's **let** ... **and** construct, applicative implementations typically fix the execution order — for example, the state applicative of Section 10.4.6 is based on the Applicative_of_monad functor (Section 10.4.5), whose implementation of <*> always executes the first operand before the second.

Although individual applicative implementations do not typically underspecify evaluation order, it is still possible to take advantage of the lack of dependencies between computations to vary the order. The following functor converts an applicative implementation into a dual applicative implementation that executes computations in the reverse order.

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (<*>) f x =
    A.(pure (|>) <*> x <*> f)
end
```

The Dual_applicative functor leaves the argument type A.t unchanged, since computations in the output applicative perform the same types of effect as computations in the input applicative. The pure function is also unchanged, since changing the order of effects makes no difference for pure computations. All of the interest is in the <*> function, which reverses the order of its arguments and uses pure (|>) to reassemble the results in the appropriate order. (The |> operator performs reverse application, and behaves like the expression **fun** y g → g y.)

It is straightforward to verify that the applicative laws hold for the result Dual_applicative(A) if they hold for the argument applicative A (Exercise 4).

We can use the Dual_applicative to convert NameA into an applicative that runs its effects in reverse:

```
module NameA' :
sig
  module Applicative : APPLICATIVE
  val fresh_name : string t
  val run : 'a t → 'a
end =
struct
  module Applicative = Dual_applicative(NameA)
  let (fresh_name, run) = NameA.(fresh_name, run)
end
```

As we saw when applying the Applicative_of_monad functor in Section 10.4.6, we must manually transport the fresh_name and run functions to the new applicative.

Here is an example of the behaviour of NameA and NameA' on a small computation:

```
# NameA.(run (pure (fun x y → (x, y)) <*> fresh_name <*> fresh_name))
    ;;
- : string * string = ("x0", "x1")
# NameA'.(run (pure (fun x y → (x, y)) <*> fresh_name <*> fresh_name))
    ;;
- : string * string = ("x1", "x0")
```

10.4.10 Example: the phantom monoid applicative

We saw in Section 10.4.5 that we can build an implementation of APPLICATIVE from a MONAD instance using the Applicative_of_monad functor. We have also seen that the two interfaces are not strictly equivalent, since there are some computations, such as fresh_name which can be defined using MONAD

(Section 10.2.4), but not using APPLICATIVE (Section 10.4.7). Are there, then, any implementations of APPLICATIVE which do not correspond to any MONAD implementation?

Here is one such example. The Phantom_counter module represents computations which track the number of times a primitive effect count is invoked. (Exercise 2 involves writing a computation using Phantom_counter.)

```
module Phantom_counter :
sig
  type 'a t = int
  module Applicative : APPLICATIVE with type 'a t = 'a t
  val count : 'a t
  val run : 'a t → int
end
 =
struct
 type 'a t = int
  module Applicative = struct
    type 'a t = int
    let pure _ = 0
    let (<*>) = (+)
  end
  let count = 1
  let run c = c
end
```

As the type shows, Phantom_counter implements the APPLICATIVE interface. However, it is not possible to use the Phantom_counter type to implement MONAD . The difficulty comes when trying to write ≫=. Since a value of type 'a Phantom_counter.t does not actually contain an 'a value (that is, the 'a is "phantom" in the sense discussed in Section 6.1.4), there is no way to extract a result from the first operand of ≫= to pass to the second operand. The monad interface promises more than Phantom_counter is able to offer.

In fact, Phantom_counter is one of a family of non-monadic applicatives parameterised by a *monoid*. The monoid interface contains a type t together with constructors zero and ++:

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end
```

We can generalize the definition of Phantom_counter to arbitrary monoids by turning the module into a functor parameterised by MONOID:

```
module Phantom_monoid_applicative (M: MONOID)
  : APPLICATIVE with type 'a t = M.t =
struct
  type 'a t = M.t
  let pure _ = M.zero
  let (<*>) = M.(++)
end
```

We'll return to monoids in more detail in Section 10.5.

10.4.11 Applicatives and monads: interfaces and implementations

Figures 10.1 and 10.2 summarise the relationship between applicatives and monads.

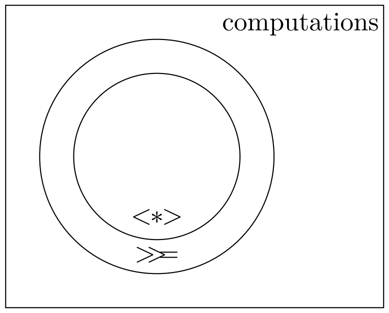

Figure 10.1: Monadic computations include applicative computations

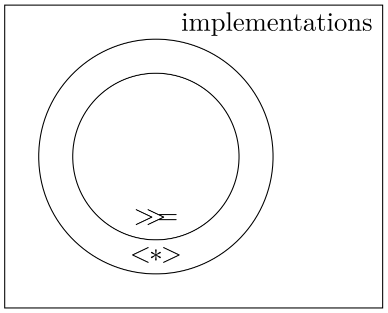

Figure 10.2: Applicative implementations include monadic implementations

Figure 10.1 is about computations. As we have seen (Section 10.4.5), every computation which can be expressed using APPLICATIVE can also be expressed using MONAD. Furthermore, there are some computations that can be expressed using MONAD that cannot be expressed using APPLICATIVE (Section 10.4.7).

Figure 10.2 is about implementations. As we have seen (Section 10.4.5), for every implementation of MONAD there is a corresponding implementation of APPLICATIVE. Furthermore, there are some implementations of APPLICATIVE which do not correspond to any implementation of MONAD (Section 10.4.10).

These inclusion relationships can serve as a guideline for deciding when to use applicatives and when to use monads. When writing computations you should prefer applicatives where possible, since applicatives give the implementer more freedom. On the other hand, when writing implementations you should expose a monadic interface where possible, since monads give the user more power.

10.5 Monoids

We have seen that applicatives offer a less powerful interface to computation than monads. A less powerful interface gives more freedom to the implementer, so it is possible to optimise applicative computations in ways that are not possible for computations written with monads. The question naturally arises, then, whether there are interfaces to effectful computation that are even less powerful than applicatives.

Monoids offer one such interface. Here is the monoid interface, which we first saw in Section 10.4.10:

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end
```

The MONOID interface corresponds approximately to APPLICATIVE with the type parameter removed. There are two constructors: zero, which builds a computation with no effects, and ++, which builds a computation from two computations. As with applicatives, there is a normal form which contains no nesting, allowing each monoid to be rearranged into the following shape:

```
zero ++ m₁ ++ m₂ ++ ... ++ mₙ
```

There are three laws, which say that ++ is associative and that zero is a left and right unit for ++:

```
m ++ (n ++ o) ≡ (m ++ n) ++ o
m ++ zero ≡ m
zero ++ m ≡ m
```

Many familiar data types can be given MONOID implementations, often in several different ways. For example, lists form a monoid, taking the empty list for zero and concatenation for ++, and integers form a monoid under either addition or multiplication.

Interpreted as computations, monoids correspond to impure expressions which do not return a useful value. In OCaml we can sequence such expressions using the semicolon:

```
m;
n;
o;
()
```

Finally, we have seen how to build an implementation of APPLICATIVE from an implementation of MONOID. It is also possible to build an implementation of MONOID from an instance of APPLICATIVE (Exercise 14.)

10.6 Exercises

1. [★★] Define a module satisfying the following signature

   ```
   module TraverseTree (A : APPLICATIVE) :
   sig
    val traverse_tree : ('a → 'b A.t) → 'a tree → 'b tree A.t
   end
   ```

   and use it to give a simpler definition of label_tree.

2. [★] Use Traverse together with Phantom_counter (Section 10.4.10) to build a computation that counts the number of nodes in a tree.

3. [★★] Write a module with the signature

```
MONAD with type 'a t = 'a list
```

and with the following behaviour (demonstrated in the OCaml top-level):

```
# [1;2;3] >>= fun x ->
  ["a";"b";"c"] >>= fun y ->
  return (x, y);;
- : (int * string) ListM.t =
   [(1, "a"); (1, "b"); (1, "c");
    (2, "a"); (2, "b"); (2, "c");
    (3, "a"); (3, "b"); (3, "c")]
```

(*Hint*: start by working out what types `return` and `>>=` should have if `'a t` is defined as `'a list`.)

4. [★★] Show that if `A` is an applicative implementation satisfying the applicative laws then `Dual_applicative(A)` is also an applicative implementation satisfying the applicative laws.

5. [★★] Show that if `M` is a monad implementation satisfying the monad laws then `Applicative_of_monad(M)` is an applicative implementation satisfying the applicative laws.

6. [★★] The `Compose` functor of Section 10.4.8 builds an applicative by composing two arbitrary applicatives. Show that there is no analogous functor that builds a monad by composing two arbitrary monads.

7. [★★★] The normal form for applicatives (Section 10.4.3) can be defined as an OCaml data type:

```
type _ t =
    Pure : 'a → 'a t
  | Apply : ('a → 'b) t * 'a A.t → 'b t
```

where `A` is a module implementing the APPLICATIVE interface. Using this definition it is possible to define a functor `Normal_applicative` that turns any implementation of APPLICATIVE into a second APPLICATIVE implementation that constructs computations in normal form. Complete the implementation of `Normal_applicative`, including the functions `lift` and `observe` that convert between the normalised representation and the underlying applicative:

```
module Normal_applicative(A: APPLICATIVE) :
sig
  type 'a t
  module Applicative : APPLICATIVE with type 'a t = 'a t
  val lift : 'a A.t → 'a t
  val observe : 'a t → 'a A.t
  end =
struct
  type _ t =
      Pure : 'a → 'a t
    | Apply : ('a → 'b) t * 'a A.t → 'b t
  (* add definitions for Applicative, lift and observe *)
end
```

8. [★★] Show that if the arguments to the Compose functor of Section 10.4.8 satisfy the applicative laws then the output module also satisfies the laws.

9. [★★] Show that the Compose functor is associative — that is, show that Compose(F)(Compose(G)(H)) produces the same result as Compose(Compose(F)(G))(H) for any applicative implementations F, G and H.

10. [★] Define an implementation Id of the APPLICATIVE interface that is an identity for composition, so that Compose(Id)(A) and Compose(A)(Id) are equivalent to A.

11. [★★★] Show that Compose(F)(G) is not the same as Compose(G)(F) for all applicatives F and G — i.e. that applicative composition is not commutative.

12. [★★★] Here is an alternative way of defining applicatives:

```
module type APPLICATIVE' =
sig
  type _ t
  val pure : 'a → 'a t
  val map : ('a → 'b) → 'a t → 'b t
  val pair : 'a t → 'b t → ('a * 'b) t
end
```

Show how to convert between APPLICATIVE and APPLICATIVE' using functors. What laws should implementations of APPLICATIVE' satisfy?

13. [★★] Show that if M is a monoid implementation satisfying the monoid laws then Phantom_monoid_applicative(M) (Section 10.4.10) is an applicative implementation satisfying the applicative laws.

14. [★★★] Define a Monoid_of_applicative functor whose input is an APPLICATIVE and whose output is a MONOID. How is it related to the Phantom_monoid_applicative functor? In particular, how is Phantom_monoid_applicative(Monoid_of_applicative(A)) related to A? How is Monoid_of_applicative(Phantom_monoid_applicative(M)) related to M?

15. [★★] Show that if A is an applicative implementation satisfying the applicative laws then Monoid_of_applicative(A) (Exercise 14) is a monoid implementation satisfying the monoid laws.

Further reading

- Applicatives are a convenient basis for building concurrent computations, since the applicative interface does not provide a way to make one computation depend on the result of another. The following paper describes an internal concurrent Facebook service based on applicatives.

  There is no fork: an abstraction for efficient, concurrent, and concise data access
  Simon Marlow, Louis Brandy, Jonathan Coens and Jon Purdy
  International Conference on Functional Programming (2014)

- *Algebraic effects* and *handlers* are a recent refinement of monadic effects which make it easier to compose independently-defined effects. The following paper investigates variants of algebraic effects based on applicatives (idioms) and on another interface to computation, arrows.

  Algebraic effects and effect handlers for idioms and arrows
  Sam Lindley
  Workshop on Generic Programming (2014)

- Parameterised monads are useful in a dependently-typed setting, where the indexes describing the state of a computation can be arbitrary terms rather than types. The following paper presents a safe file-access interface using parameterised monads, and investigates the connection to Hoare logic.

  Kleisli arrows of outrageous fortune Conor McBride
  Journal of Functional Programming (2011)

- The following two papers present a core calculus for another interface to computation, arrows, and compare the relative expressive power of arrows, applicatives and monads.

  The arrow calculus
  Sam Lindley, Philip Wadler, and Jeremy Yallop
  Journal of Functional Programming (2010)

  Idioms are oblivious, arrows are meticulous, monads are promiscuous
  Sam Lindley, Philip Wadler, and Jeremy Yallop
  Mathematically Structured Functional Programming (2008)

- The usefulness of parameterised monads extends well beyond the typed state monad that we have considered in this chapter. The following paper presents parameterised monads in detail with many examples, including typed I/O channels, delimited continuations and session types.

Parameterised notions of computation
Robert Atkey
Journal of Functional Programming (2009)

- As we saw in Section 10.4.8, applicatives can be built by composing simpler applicatives. The following paper shows how to compose three simple applicatives to build a reusable abstraction for web programming.

  The essence of form abstraction
  Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop
  Asian Symposium on Programming Languages and Systems (2008)

- McBride and Paterson's 2008 paper introduced the applicative interface:

  Applicative programming with effects
  Conor McBride and Ross Paterson
  Journal of Functional Programming (2008)

- Parser combinators have become a standard example for using monads to structure programs. The typical presentations of parser combinators are more suited to lazy languages like Haskell than eager languages like OCaml. The following paper is a tutorial introduction to monadic parser combinators.

  Monadic parser combinators
  Graham Hutton and Erik Meijer
  Technical Report, University of Nottingham (1996)

- The higher-order nature of the monadic interface makes it impossible to analyse parsers before running them, leading to inefficiencies and delayed error reporting. Swierstra and Duponcheel structure parsers using an applicative interface to make them more amenable to static analysis. Applicatives aren't discussed explicitly in the paper, since they were only identified as a separate abstraction some years later.

  Deterministic, Error-Correcting Combinator Parsers
  S. Doaitse Swierstra and Luc Duponcheel
  Advanced Functional Programming (1996)

- Wadler introduced monads as a way of structuring functional programs in the early 1990s. The following paper focuses on using monads for I/O in a pure language:

  How to declare an imperative
  Philip Wadler
  International Logic Programming Symposium (1995)

- Although monads do not compose directly, the related concept of *monad transformers*, described in the following paper, can be used to compose monadic effects.

  Monad Transformers and Modular Interpreters
  Sheng Liang, Paul Hudak and Mark P. Jones
  Principles of Programming Languages (1995)