Chapter 8

First-class effects

8.1 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 which 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 which 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*.

8.2 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.

8.2.1 The monad interface

The monad interface can be defined as an OCaml signature (Section 5.1.1):

```
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

[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 which prints `"ocamlc"` or `"ocamlopt"` according to which compiler is used:

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

arguments. 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
```

8.2.2 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 a little 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.)

8.2.3 Example: a state monad

The monad interface is not especially useful in 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
  include MONAD
  type state
  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 which 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 `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 which 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 which 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 which 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 6.1.2) so that we can abstract over the `state` type:

```
module State (S : sig type t end)
  : STATE with type state = S.t =
struct
  type state = S.t
  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'
  let get s = (s, s)
  let put s' _ = (s', ())
  let runState m ~init = m init
end
```

8.2.4 Example: fresh names

How might we use `State` to write an effectful function? Let's consider a function which 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.

8.2.5 Example: an exception monad

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

```
module type ERROR =
sig
  type error
  include MONAD
  val raise : error → _ t
  val _try_ : 'a t → catch:(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
 ~catch:(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)
  : ERROR with type error = E.t =
struct
  type error = E.t
  type 'a t =
     Val : 'a → 'a t
   | Exn : error → 'a t
  let return v = Val v
  let raise e = Exn e
  let (≫=) m k =
    match m with
      Val v → k v
    | Exn 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 which returns a value, while `raise` constructs a computation which raises an exception. The behaviour of ≫= depends on its first argument. If the first argument is a computation which returns a value then that value is passed to the second argument and the computation continues. If, however, the first argument is a computation which raises an exception then the result of ≫= 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 which 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 which may raise exceptions from functions which 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 which when run will either raise an exception or return a value of type `'a`.

8.3 Parameterised monads

Up to this point we have not gained a great deal by using monads to write effectful functions. Since OCaml has both state and exceptions as primitive effects, we could give simpler and more efficient implementations of all the functions that we've seen so far without using monads.

The benefits of the monadic style become clear when it becomes necessary to use effects that are not primitive in OCaml. In some cases (Exercise 3) these effects can be implemented by adding additional value members to `MONAD`, just as we added `get` and `put` to support state, and `raise` to support exceptions. However, in order to implement some effects with special typing rules we need to extend the `MONAD` interface in a different direction, by adding parameters to the type `t`.

We will now consider a refinement of the monad interface which makes it possible to capture the changes that take place as a computation runs. Here is the *parameterised monad* interface:

```
module type PARAMETERISED_MONAD =
sig
```

```
  type ('s,'t,'a) t
  val return : 'a → ('s,'s,'a) t
  val (>>=) : ('r,'s,'a) t →
       ('a → ('s,'t,'b) t) →
             ('r,'t,'b) t
end
```

The `PARAMETERISED_MONAD` interface has the same three members as the `MONAD` interface of Section 8.2.1, but includes additional type parameters. The type `t` of computations has three parameters. The first parameter, `'s`, represents the state in which the computation starts; the second parameter, `'t`, represents the state in which the computation ends, and the final parameter `'a` represents the result of the computation. The instantiations of the two additional type parameters in the types of `return` and >>= reflect the behaviour of those operations. In the type of `return` the start and end parameter have the same type `'s`, indicating that a computation constructed with `return` does not change the state of the computation. The type of >>= reflects the ordering between the computations involved. The first argument to >>= is a computation which starts in state `'r` and finishes in state `'s`; the second argument constructs a computation which starts in state `'s` and finishes in state `'t`. Combining the two computations using >>= produces a computation which starts in state `'r` and finishes in state `'t`, indicating that it runs the effects of the first argument of >>= before the effects of the second argument.

The laws for parameterised monads are the same as the laws for monads, except for the types of the expressions involved.

8.3.1 Example: a parameterised monad for state

Perhaps the simplest example of parameterised monads is to improve the typing of computations involving state. By interpreting the additional type parameters as the type of a reference cell we can define a more powerful state monad in which the type of the cell can change over the course of a computation. As we saw in Chapter 3, OCaml does not support polymorphic references, so our enhanced state monad will allow us to build computations that it is not possible to write in an imperative style.

Here is an interface for a parameterised state monad:

```
module type PSTATE =
sig
 include PARAMETERISED_MONAD
 val get : ('s,'s,'s) t
 val put : 's → (_,'s,unit) t
 val runState : ('s,'t,'a) t → init:'s → 't * 'a
end
```

The `PSTATE` interface extends `PARAMETERISED_MONAD` with functions `get`, `put` and `runState`, much like the `STATE` interface which extended `MONAD` (Section 8.2.3). The type parameters of each operation indicate what effect the operation has on the state of the reference cell. Since `get` retrieves the contents of the cell, leaving it unchanged, the start and end parameters match both each other and the result type. The type of `put` also reflects its behaviour: since `put` overwrites the reference cell with the value passed as argument, the end state of the computation matches the type `'s` of the parameter. Finally, `runState` runs a computation which turns a reference of type `'s` into a reference of type `'t`; it is therefore parameterised by an initial state of type `'s` and returns a value of type `'t`, along with the result of the computation.

Here is an implementation of `PSTATE`:

```
module PState : PSTATE =
struct
  type ('s, 't, 'a) t = 's → 't * 'a
  let return v s = (s, v)
  let (≫=) m k s = let s', a = m s in k a s'
  let get s = (s, s)
  let put s' _ = (s', ())
  let runState m ~init = m init
end
```

As with `State` (Section 8.2.3) the type `t` of computations is defined as a function from the initial state of the computation to a pair of the final state and the result. Since the type state is allowed to vary, the initial and final state types are now type parameters rather than the fixed type `state` used in the unparameterised `State` monad. Since we no longer need to parameterise the whole definition by a single state type it is not necessary for `PState` to be a functor.

Comparing `PState` with the `State` implementation of Section 8.2.3 reveals that the implementations are otherwise identical; only the types have changed.

Programming with polymorphic state

The parameterised state monad makes it possible to construct a variety of computations. We will use it to build a simple typed stack machine.

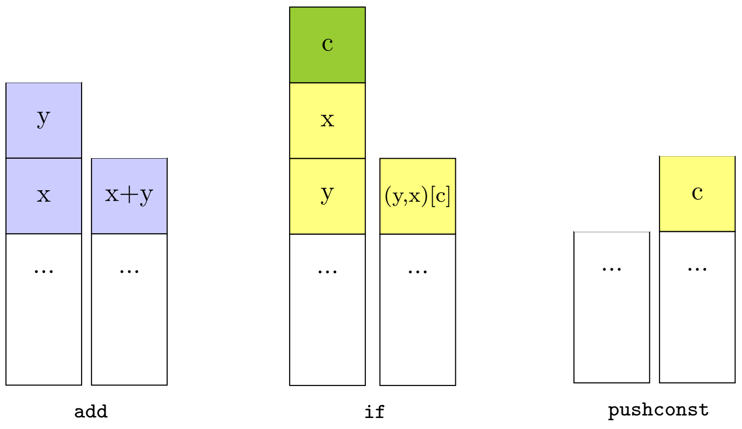

Figure 8.1: Stack machine operations

Figure 8.1 shows the three operations of the stack machine and their effects on the stack. The first instruction, `add`, replaces the top two elements on the stack with their sum. The second instruction, `if`, removes the top three elements from the stack and adds either the second or the third element back according to whether the top element was **true** or **false**. The third instruction, `pushconst`, adds a value to the top of the stack.

We can give the types of the stack operations as a module signature[2]:

```
module type STACK_OPS =
sig
  type ('s,'t,'a) t
  val add : unit → (int * (int * 's),
                            int * 's, unit) t
  val _if_ : unit → (bool * ('a * ('a * 's)),
                                    'a * 's, unit) t
  val push_const : 'a → ('s,
                     'a * 's, unit) t
end
```

The type `('s, 't, 'a) t` represents stack machine programs which transform a stack of type `'s` into a stack of type `'t` and return a result of type `'a`. The type of each operation shows how the operation changes the type of the stack. The `add` operation turns a stack of type `int * (int * 's)` into a stack of type `int * 's` — that is it replaces the top two (i.e. leftmost) integers on the stack with a single integer, leaving the rest of the stack unchanged. The `_if_` operation turns a stack of type `bool * ('a * ('a * 's))` into a stack of type `'a * 's` — that is, it removes a `bool` value from the top of the stack and removes one of the two values below the `bool`, leaving the rest of the stack unchanged. The `push_const` value turns a stack of type `'s` into a stack of type `'a * 's` — that is, it adds a value to the top of the stack.

We can combine the stack operations with the parameterised monad signature to build a signature for a stack machine:

```
module type STACKM = sig
 include PARAMETERISED_MONAD
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val stack : unit → ('s, 's, 's) t
 val execute : ('s,'t,'a) t → 's → 't * 'a
end
```

Just as in the `STACK_OPS` signature, a computation of type `('s, 't, 'a) t` is a stack machine program which transforms a stack of type `'s` into a stack of type `'t`. As with the monads we've seen already, the `return` function builds a trivial computation and the $\gg=$ function builds a computation from the two computations passed as arguments. More precisely, `return` builds a program that leaves the stack untouched, and $\gg=$ builds a program that runs its two arguments in sequence. In addition to the stack machine operations there is one additional primitive computation, `stack`, which makes it possible to observe the

[2]The `unit` arguments to `add` and `_if_` save us from running into problems with the value restriction.

state of the stack at a particular point in the computation: calling `stack` builds a computation which returns the current stack as its result. The `execute` function plays the same role as `runState` for the `State` monad and `_try_` for the `Error` monad — that is, it runs a computation. More precisely, `execute` takes a computation representing a stack machine program together with an input stack and returns a pair of the output stack together with the result of the computation.

We can implement `StackM` using the parameterised state monad `PState`:

```
module StackM : STACKM =
struct
  include PState

  let add () =
    get ≫= fun (x,(y,s)) →
    put (x+y,s)

  let _if_ () =
    get ≫= fun (c,(t,(e,s))) →
    put ((if c then t else e),s)

  let push_const k =
    get ≫= fun s →
    put (k, s)

  let stack () = get

  let execute c s = runState ~init:s c
end
```

The implementation is a straightforward use of the parameterised state monad. A computation in the stack machine monad is interpreted directly as a computation in the parameterised state monad, with the reference cell holding the stack. Each of the instruction operations (`add`, `_if_` and `push_const`) builds a computation which retrieves the current stack, performs an appropriate transformation, and stores the updated stack. The `stack` function uses `get` to retrieve the current stack, and `execute` is implemented directly using `runState`.

8.4 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 which 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.

8.5 Applicatives

Applicatives[3] offer a second interface to effectful computation which is less powerful and therefore, from a certain perspective, more general than monads.

8.5.1 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 8.1). Like **let**, the monadic >>= operation both sequences computations and makes the result of one computation available for constructing other computations (Section 8.4). 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 $\texttt{e}_1$ and $\texttt{e}_2$ 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

[3]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 165) use the name "idioms" under which applicatives were originally introduced.

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**[4]:

```
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.

8.5.2 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 which 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 8.2.1) reveals a number of minor differences — the operations are called `pure` and <*> (pronounced “apply”) rather than `return` and ≫=, and the function argument to <*> comes first rather than second — and one significant difference. Here is the type of ≫=:

```
    '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 which 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.

[4]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.

8.5.3 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,...,c_n are primitive computations which do not involve the computation constructors `pure` and `<*>`.

8.5.4 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 8.5.3.

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 8.5.3, 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
```

8.5.5 Applicatives and monads

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

```
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 8.5.4:

```
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)
```

8.5.6 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
  include APPLICATIVE
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end =
struct
  type state = S.t
  module M = State(S)
  include Applicative_of_monad(M)
  let (get, put, runState) = M.(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.

8.5.7 Example: fresh names

We have converted the `State` monad to an corresponding applicative (Section 8.5.6). Can we write an applicative analogue to the `label_tree` function of Section 8.2.4?

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
  include APPLICATIVE
  val fresh_name : string t
  val run : 'a t → 'a
end =
struct
  module M = State(struct type t = int end)
  include 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 8.2.4 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.

8.5.8 Composing applicatives

Section 8.5.5 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 7), 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 9).

8.5.9 Example: the dual applicative

As we have seen (p142), 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 8.5.6 is based on the `Applicative_of_monad` functor (Section 8.5.5), 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 which 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
  include APPLICATIVE
  val fresh_name : string t
  val run : 'a t → 'a
end =
struct
  include Dual_applicative(NameA)
  let (fresh_name, run) = NameA.(fresh_name, run)
end
```

As we saw when applying the `Applicative_of_monad` functor in Section 8.5.6, 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")
```

8.5.10 Example: the phantom monoid applicative

We saw in Section 8.5.5 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 8.2.4), but not using `APPLICATIVE` (Section 8.5.7). 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
  include APPLICATIVE with type 'a t = int
  val count : 'a t
  val run : 'a t → int
end
 =
struct
  type 'a t = int
  let pure _ = 0
  let count = 1
  let (<*>) = (+)
  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 5.1.4), 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 8.8.

8.6 Parameterised applicatives

Section 8.3 introduced the notion of *parameterised* monads. The same idea extends naturally to applicatives. Here is a signature for parameterised applicatives, in which the type `t` is given two additional parameters `'s` and `'t` to represent the start and end states of a computation:

```
module type PARAMETERISED_APPLICATIVE =
sig
  type ('s,'t,'a) t
  val pure : 'a → ('s,'s,'a) t
  val (<*>) : ('r,'s,'a → 'b) t
          → ('s,'t,'a) t
          → ('r,'t,'b) t
end
```

The instantiation of the first two parameters echos the definition of `PARAMETERISED_MONAD` : `pure` computations have the same start and end states, while computations built using <*> combine a computation that changes state from `'r` to `'s` and a computation that changes state from `'s` to `'t` to build a computation that changes state from `'r` to `'t`.

As in the unparameterised case, it is possible to define a functor that builds a parameterised applicative from a parameterised monad (Exercise 5).

8.6.1 Example: optimising stack machines

How should we choose whether to expose a particular computational effect (such as state, naming, or exceptions) as an applicative or as a monad? In some cases the decision is easy, since certain applicatives do not correspond to any monad (Section 8.5.10), so using the `MONAD` interface is not possible. However, it is often the case that a particular computational effect can be exposed under either interface (Section 8.5.5). One consideration is the degree of expressive power that we would like to expose to the user. If we would like to offer the user as much power as possible then the `MONAD` interface is clearly the better choice (Section 8.5.7). However, as we shall now see, exposing a less powerful interface can sometimes make it possible to implement an effect more efficiently.

We introduced parameterised monads by showing how to implement a typed stack machine which operates by transforming a stack represented as a nested tuple. Since programs are typically run than once we we might like to optimise our stack machine programs to run more efficiently. For example, we can tell without running the program, and without knowing anything about the current state of the stack, that the following program fragment will end with `7` at the top of the stack:

```
push_const 3 ≫= fun () →
push_const 4 ≫= fun () →
add ()
```

and so we might like to transform the instruction sequence into the following more efficient fragment:

```
push_const 7
```

Unfortunately, the implementation of `STACKM` in Section 8.3.1 is not very amenable to optimisations of this sort. The difficulty is that we do not have a concrete representation of the stack machine instructions — indeed, it is diffulct to see how we such a representation is possible with monads, since the dynamic dependencies supported by the `MONAD` interface (Section 8.4) allow the instructions to be synthesised as the stack machine program runs. In particular, we can write a computation using `StackM` that uses the `stack` function to observe the stack, and then decides which instruction to run next based on the result:

```
  stack () ≫= fun (top, (next, _)) →
  if top < next
    then add ()
    else push_const true ≫= fun () → _if_ ()
```

Optimising our stack machine programs before they run will become much easier if we switch from a higher-order representation of instructions to a first-order representation, which requires switching from the powerful `MONAD` interface with its dynamic dependencies to the less powerful `APPLICATIVE` interface, which cannot be used to write programs like the above.

We will start with a representation of individual instructions. The `instr` type is a GADT whose constructors have types which are derived directly from the types of the `STACK_OPS` interface:

```
type (_, _) instr =
   Add : (int * (int * 's),
                int * 's) instr
 | If : (bool * ('a * ('a * 's)),
                      'a * 's) instr
 | PushConst : 'a → ('s,
                'a * 's) instr
```

A program consists of a sequence of instructions. Since each instruction has a different type we define a custom list type `instrs` rather than using OCaml's `list`:

```
type (_, _) instrs =
   Stop : ('s, 's) instrs
 | ::   : ('s1, 's2) instr * ('s2, 's3) instrs → ('s1, 's3) instrs
```

As with `instr`, the two type parameters for `instrs` represent the state of the stack before and after the instructions run. The `Stop` constructor represents an empty sequence of instructions which leaves the state of the stack unchanged. The type of the `::` constructor shows that instructions are run left-to-right: first the left argument (a single instruction) changes the state of the stack from `'s1` to `'s2`, then the right argument (a sequence of instructions) changes the state of the stack from `'s2` to `'s3`.

Here is an example program:

```
let program =
  PushConst 3 :: PushConst 4 :: PushConst 5 ::
  PushConst true :: If :: Add :: Stop
```

The type of this program is `('s, int * 's) instrs` — that is it transforms a stack of type `'s` by adding an `int` to the top.

Rather than deal with concrete programs directly, we will build an interface for constructing programs based on `APPLICATIVE`. We can build a suitable interface by combining `PARAMETERISED_APPLICATIVE` with `STACK_OPS`:

```
module type STACKA = sig
 include PARAMETERISED_APPLICATIVE
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't * 'a
 val stack : unit → ('s, 's, 's) t
end
```

In order to implement `STACKA` we need a couple of auxiliary functions. The first function, `@.`, appends two instruction lists:

```
let rec (@.) : type a b c. (a,b) instrs → (b,c) instrs → (a,c) instrs
   =
 fun l r → match l with
```

```
    Stop → r
  | x :: xs → x :: (xs @. r)
```

The implementation of `@.` looks just like the implementation of an append function for regular lists; only the types are different.

The second function, `exec`, executes a list of instructions:

```
let rec exec : type r s. (r, s) instrs → r → s =
  fun instrs s → match instrs, s with
    Stop, _ → s
  | Add :: instrs, (x, (y, s)) → exec instrs (x + y, s)
  | If :: instrs, (c, (t, (e, s))) → exec instrs ((if c then t else e),
     s)
  | PushConst c :: instrs, s → exec instrs (c, s)
```

The function is defined by simultaneous matching over the instruction list and the stack. The type refinement (Section 7.1.1) that takes place when the instructions are matched assigns different types to the stack in each branch.

Here is an implementation of `STACKA`:

```
let const x _ = x

module StackA : STACKA =
struct
  type ('s, 't, 'a) t = ('s, 't) instrs * ('s → 'a)
  let pure v = (Stop, const v)
  let add () = (Add :: Stop, const ())
  let _if_ () = (If :: Stop, const ())
  let push_const c = (PushConst c :: Stop, const ())
  let stack () = (Stop, (fun s → s))
  let (<*>) (f, g) (x, y) =  (f @. x, (fun s → g s (y (exec f s))))
  let execute (i, k) s = (exec i s, k s)
end
```

The type `t` is a pair of an instruction list and a function over stacks. As with `STACKM` the three type parameters to `t` represent the type of the stack before a computation runs, the type of the stack after the computation runs, and the result of the computation.

The `pure` function takes an argument `v` and builds an empty instruction list and a function which ignores the input stack, simply returning `v`. The three instruction functions `add`, `_if_` and `push_const`, build singleton lists of instructions, along with functions which ignore the input stack and return the unit value `()`. The `stack` function makes it possible to observe the stack during the execution of a computation. Calling `stack` builds an empty instruction list together with a function which extracts the current value of the stack as the result of the computation. The <*> function combines two computations `(f,g)` and `(x,y)`. The resulting instruction list is the concatenation of the instruction lists `f` and `x`. The second component of the pair is a function which passes the input stacks for `f` and `x` to `g` and `y`, and then combines the results by application.

Finally, `execute` runs a computation `(i, k)` by executing the set of instructions `i` against an input stack `s` to build the output stack and additionally passing `s` to `k` to determine the computation's result.

Optimising the stack machine Now that we have a concrete (first-order) representation of instructions, it is easy to optimize programs as we construct them. Here is a function, `optimize`, which transforms a list of instructions to reduce addition of known constants:

```
let rec optimize : type r s a. (r, s, a) t → (r, s, a) t = function
    PushConst x :: PushConst y :: Add :: ops →
      optimize (PushConst (x + y) :: ops)
  | op :: ops → op :: optimize ops
  | Stop → Stop
```

The type of `optimize` gives us some reassurance that the optimization is sound, since both the input and the output instruction lists transform a stack of type `r` to a stack of type `r`.

Here is a second implementation of <∗> which applies `optimize` to simplify the newly-constructed argument list:

```
  let (<*>) (f, g) (x, y) =  (optimize (f @. x),
                                   (fun s → g s (y (exec f s))))
```

It would not be difficult to add new optimizations for other instruction sequences. We will return to the question of stack machine optimization in Chapter 11, where we will use staging to further improve the performance of stack machine programs.

8.6.2 Applicatives and monads: interfaces and implementations

Figures 8.2 and 8.3 summarise the relationship between applicatives and monads.

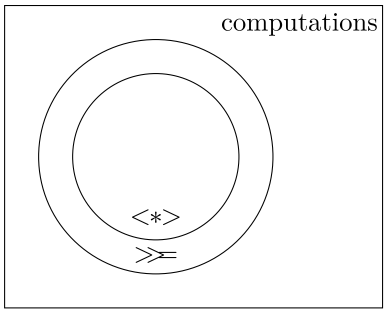

Figure 8.2: Monadic computations include applicative computations

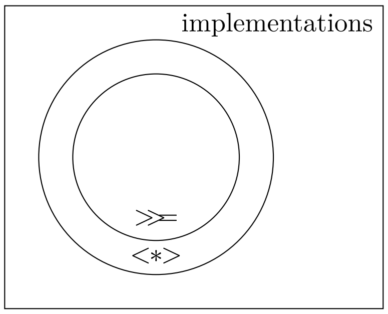

Figure 8.3: Applicative implementations include monadic implementations

Figure 8.2 is about computations. As we have seen (Section 8.5.5), 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 8.5.7).

Figure 8.3 is about implementations. As we have seen (Section 8.5.5), 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 8.5.10).

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.

8.7 Arrows

Applicatives and monads lie at fairly extreme points on a spectrum of expressiveness. With monads, computations can depend in arbitrary ways on other computations: the result of one computation can determine which values to pass to another computation, or which computation should run next, or whether a subsequent computation should run at all. With applicatives, none of these dependencies are possible: the results of computations may be combined with pure functions, but the result of one computation is never available as input for another computation.

Arrows offer a third interface to computation which is intermediate between applicatives and monads in expressive power. With arrows, as with monads, the result of each computation is made available as input to subsequent computations. For example, as with monads, it is possible to use the result of a call to get to construct the argument of a subsequent call to put. However, as with applicatives, the computations themselves are fixed in advance: there is no way to use the result of a call to get to determine whether to call put or get as the next computation.

Here is the arrow interface as an OCaml signature:

```
module type ARROW =
sig
 type ('a, 'b) t
 val arr : ('a → 'b) → ('a, 'b) t
 val (>>>) : ('a, 'b) t → ('b, 'c) t → ('a, 'c) t
 val first : ('a, 'b) t → ('a * 'c, 'b * 'c) t
end
```

Similarly to APPLICATIVE and MONAD, the ARROW signature includes a type t for representing computations, a function arr for constructing trivial computations that perform no effects, and a function >>> for combining computations. In order to carefully control the use of dependencies, the computation type t is parameterised by both an input type 'a and a result type 'b.

It is instructive to compare the type of >>> with the applicative <*> and the monadic >>=.

The applicative <∗> operator combines two computations, of types `('a → 'b) t` and `'a t`, to build a new computation of type `'b t`. The results of the input computations, of type `'a → 'b` and `'a`, are combined to produce the output result, of type `'b`:

```
val (<*>): ('a → 'b) t → 'a t  → 'b t
```

The monadic ≫= operator combines a computation of type `'a t` with a function of type `'a → 'b t` which receives the result of the computation and and uses it to construct a new computation:

```
val (≫=): 'a t → ('a -> 'b t)  → 'b t
```

The arrow ⋙ operator combines two computations, of type `('a, 'b) t` and `('b, 'c) t`, passing the result of the first computation as input to the second computation:

```
val (⋙): ('a,'b) t → ('b,'c) t  → ('a, 'c) t
```

As with <∗>, the two arguments to ⋙ are already-constructed computations, not functions which build computations from previous results. However, ⋙ is more powerful than <∗>, since it passes the result of the first computation to the second rather than simply combining the results.

The final operation in the `ARROW` interface, `first`, provides a way to adjust computations so that they pass additional data alongside their inputs and outputs. For example, given a computation `f` of type `(int, float * bool) t`, a computation `g` of type `(float, string) t`, and a computation `h` of type `(string * bool, unit) t`, the following computation uses `first` to pass the `bool` result from `f` through to `h`:

```
f ⋙ first g ⋙ h
```

8.7.1 Arrow laws and normal forms

As with monads and applicatives, an implementation of the `ARROW` interface must satisfy a number of laws in order to be considered an arrow. Figure 8.4 shows the nine arrow laws, which make use of some simple auxiliary functions defined in Figure 8.5.

The arrow laws can be used to rewrite any computation built from the arrow operations into a normal form, making it easy to determine whether two computations are equivalent. Here is one such normal form, which makes the results of every sub-computation available to all subsequent sub-computations:

```
((arr f₁ ⋙ c₁) &&& arr id)
  ⋙
((arr f₂ ⋙ c₂) &&& arr id)
  ⋙
  ...
  ⋙
((arr fₙ ⋙ cₙ) &&& arr id)
   ⋙
 arr g
```

where the functions `dup`, `swap`, `second` and `&&&` are defined as follows:

```
                 arr id ⋙ f   ≡  f
                 f ⋙ arr id   ≡  f
             (f ⋙ g) ⋙ h   ≡  f ⋙ (g ⋙ h)
                  arr (g ∘ f)   ≡  arr f ⋙ arr g
                first (arr f)   ≡  arr (f × id)
               first (f ⋙ g)   ≡  first f ⋙ first g
     first f ⋙ arr (id × g)   ≡  arr (id × g) ⋙ first f
           first f ⋙ arr fst   ≡  arr fst ⋙ f
 first (first f) ⋙ arr assoc   ≡  arr assoc ⋙ first f
```

Figure 8.4: Arrow laws

```
let id x = x
let (×) f g (x,y) = (f x, g y)
let assoc ((x,y),z) = (x,(y,z))
let (∘) f g x = f (g x)
```

Figure 8.5: Arrow laws: auxiliary definitions

```
let dup x = (x,x)
let swap (x,y) = (y,x)
let second f = arr swap ⋙ first g ⋙ arr swap
let (&&&) f g = arr dup ⋙ first f ⋙ second g
```

8.7.2 Arrows and monads

Since arrows are strictly less powerful than monads, any monad instance can be used to construct an arrow instance that hides some of the expressive power. The following functor accepts an implementation of `MONAD` and builds a related implementation of `ARROW`, using `return` to construct `arr`, ≫= to construct ⋙, and a combination of ≫= and `return` to construct `first`.

```
module Arrow_of_monad (M: MONAD) :
  ARROW with type ('a, 'b) t = 'a -> 'b M.t =
struct
  type ('a, 'b) t = 'a -> 'b M.t
  let arr f x = M.return (f x)
  let (⋙) f g x = M.(f x ≫= fun y -> g y)
  let first f (x,y) =
    M.(f x ≫= fun z -> return (z, y))
end
```

As usual, it can be shown that if `M` satisfies the monad laws then `Arrow_of_monad (A)` satisfies the arrow laws (Exercise 18).

It is instructive to compare the definitions of ⋙ and `first` with the **let** notation. Here is a definition analogous to ⋙, in which the result `y` of calling `f` is made available only as a parameter to the following computation `g`:

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

And here is a definition analogous to `first`, in which the second component `y` of a pair is threaded alongside the computation `f`, which acts only on the first component:

```
let first f (x,y) =
  let z = f x in
     (z, y)
```

The following Section (8.7.2) illustrates with an example how changing from a monad to an arrow removes the ability to express computations with dynamic dependencies.

Arrows and higher-order programming

Section 8.4 introduced the following definition of `uncurryM`, which builds and runs a computation `g` that results from another computation `f`:

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

This kind of higher-order dynamic dependence, where the result of one computation determines the next computation to run, cannot be expressed with arrows. The second argument to the monadic ≫= is a function that accepts the result of running the computation passed as the first argument and uses that result to construct the next computation. In contrast, the corresponding arrow operation ⋙ combines computations which have already been constructed, so there is no opportunity to choose the second computation based on the result of the first.

Example: the phantom monoid arrow

If the arrow interface is less powerful than the monad interface, then why would we ever choose to use arrows rather than monads? In fact, although the less expressive interface offers less to clients (i.e. to code which uses the arrow operations), it requires less of implementers (i.e. to code which defines the arrow operations), and so there is a greater variety of arrow implementations than monad implementations.

For example, here is an implementation the `ARROW` interface, analogous to the `Phantom_monoid_applicative` applicative interface of Section 8.5.10, whose operations are defined straightforwardly in terms of the monoid operations, and which does not use its input and result types.

```
module Phantom_monoid_arrow (M: MONOID)
  : ARROW with type ('a, 'b) t = M.t =
struct
  type ('a, 'b) t = M.t
  let arr _ = M.zero
  let (⋙) f g = M.(f ++ g)
  let first f = f
end
```

8.7.3 Arrows and applicatives

Since arrows are strictly more powerful than applicatives, any arrow instance can be used to construct an applicative instance that hides some of the expressive power. The following functor accepts an implementation of `ARROW` and builds a related implementation of `APPLICATIVE`, using `arr` to construct `pure` and ⋙ and `first` to construct <∗>.

```
module Applicative_of_arrow (A: ARROW) :
  APPLICATIVE with type 'a t = (unit, 'a) A.t =
struct
  type 'a t = (unit, 'a) A.t
  let pure x = A.arr (fun () -> x)
  let (<*>) f p =
    A.(f ⋙ arr (fun g -> ((), g)) ⋙
       first p ⋙ arr (fun (y, g) -> (g y)))
end
```

As usual, if `A` satisfies the arrow laws then `Applicative_of_arrow(A)` satisfies the arrow laws (Exercise 19).

The next section illustrates how changing from an arrow to an applicative removes the ability to pass the result of one computation as input to another.

Example: fresh names

Section 8.5.7 illustrated a difference in expressive power between monads and applicatives: the `fresh_name` operation which could be written in terms of the `get` and `put` operations of the `State` monad had to be added as a primitive operation to the corresponding applicative.

Although the arrow interface is less powerful than the monad interface, it offers more than the applicative interface. In particular, it supports passing the result of one computation as input to another, which is precisely what is needed to define `fresh_name` in terms of `get` and `put`. Here is an implementation of an arrow analogue to the `State` monad, with operations for reading and writing a single reference cell:

```
module State_arrow (S: sig type t end) :
sig
  include ARROW
  val get : (unit, S.t) t
  val put : (S.t, unit) t
end =
struct
  module M = State(S)
  include Arrow_of_monad(M)
  let get, put = M.((fun () -> get), put)
end
```

And here is an implementation of `fresh_name` in terms of `get`, `put`, and the arrow operations:

```
module IState_arrow = State_arrow(struct type t = int end)

let fresh_name : (unit, string) State_arrow.t =
  get ⋙ arr (fun s → (s+1, s)) ⋙
  first put ⋙ arr (fun ((), s) → sprintf "x%d" s)
```

As before, we can use `fresh_name` to define a function that traverses a tree, replacing each label with a fresh name:

```
let rec label_tree :
  'a. 'a tree -> (unit, string tree) IState_arrow.t =
  function
    Empty -> arr (fun () -> Empty)
  | Tree (l, v, r) ->
    label_tree l ⋙
    arr (fun l -> ((), l)) ⋙
    first fresh_name ⋙
    arr (fun (n, l) -> ((), (n, l))) ⋙
    first (label_tree r) ⋙
    arr (fun (r, (n, l)) -> Tree (l, n, r))
```

Reversing effects

As we have seen, an interface that offers more to clients often demands more of implementers. The arrow interface is more expressive than the applicative interface, but as a result there are fewer arrow implementations than applicative implementations. For example, there is no arrow transformer corresponding to the `Dual_applicative` functor of Section 8.5.9, which reverses the effects of an applicative. The `ARROW` interface exposes the dependencies between arrow computations in the input and result types of each computation, and as a result it is not possible to reorder computations.

8.7.4 Arrows, applicatives and monads: interfaces and implementations

Section 8.6.2 summarised the relationship between applicatives and monads. Figures 8.6 and 8.7 extend the summary with arrows, which are intermediate in expressiveness between applicatives and monads.

Figure 8.6 is about computations. As we have seen, every computation which can be expressed using `APPLICATIVE` can be expressed using `ARROW` (Section 8.7.3), and every computation which can be expressed using `ARROW` can be expressed using `MONAD` (Section 8.7.2). Furthermore, there are some computations that can be expressed using `MONAD` that cannot be expressed using `ARROW`, such as `uncurryM` (Section 8.7.2), and some computations that can be expressed using `ARROW` that cannot be expressed using `APPLICATIVE`, such as `fresh_name` (Section 8.7.3).

Figure 8.7 is about implementations. As we have seen, for every implementation of `MONAD` there is a corresponding implementation of `ARROW` (Section 8.7.2), and for every implementation of `ARROW` there is a corresponding implementation of `APPLICATIVE` (Section 8.7.3). Furthermore, there are some implementations of `APPLICATIVE`, such as `Dual_applicative`, which do not correspond to any implementation of `ARROW` (Section 8.7.3) and some implementations of `ARROW`, such as

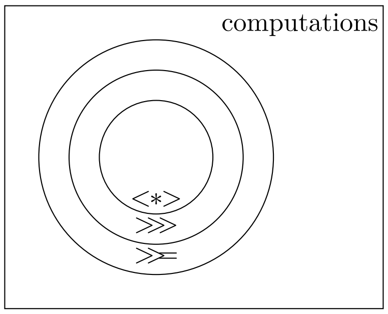

Figure 8.6: Monadic computations include arrow computations; arrow computations include applicative computations

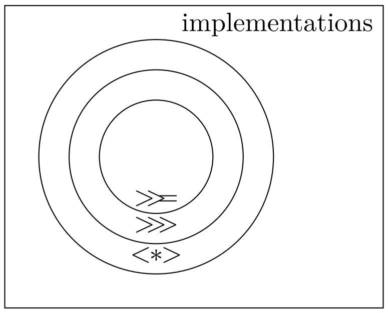

Figure 8.7: Applicative implementations include arrow implementations; arrow implementations include monadic implementations

Phantom_arrow, which do not correspond to any implementation of MONAD (Section 8.7.2).

8.8 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 8.5.10:

```
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:

$$\texttt{zero} \mathbin{++} m_1 \mathbin{++} m_2 \mathbin{++} \ldots \mathbin{++} m_n$$

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 15.)

8.9 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 8.5.10) 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. [★] The functor `Applicative_of_monad` (Section 8.5.5) builds an applicative implementation from a monad implementation. Define an analogous function `Parameterised_applicative_of_parameterised_monad` that builds a parameterised applicative implementation from a parameterised monad implementation.

6. [★★] 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.

7. [★★] The `Compose` functor of Section 8.5.8 builds an applicative by composing two arbitrary applicatives. Show that there is no analogous functor that builds a monad by composing two arbitrary monads.

8. [★★★] The normal form for applicatives (Section 8.5.3) 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` which turns any implementation of `APPLICATIVE` into a second `APPLICATIVE` implementation which constructs computations in normal form. Complete the implementation of `Normal_applicative`, including the functions `lift` and `observe` which convert between the normalised representation and the underlying applicative:

```
module Normal_applicative(A: APPLICATIVE) :
sig
  include APPLICATIVE
  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 pure, <*>, lift and observe *)
end
```

9. [★★] Show that if the arguments to the `Compose` functor of Section 8.5.8 satisfy the applicative laws then the output module also satisfies the laws.

10. [★★] 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`.

11. [★] 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`.

12. [★★★] 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.

13. [★★★] 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?

14. [★★] Show that if `M` is a monoid implementation satisfying the monoid laws then `Phantom_monoid_applicative(M)` (Section 8.5.10) is an applicative implementation satisfying the applicative laws.

15. [★★★] 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`?

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

17. [★★★] Applicative implementations can be built from monad implementations in two ways: either directly using the `Applicative_of_monad` functor, or in two steps by first using the `Arrow_of_monad` functor and then applying the `Applicative_of_arrow` to the result. Do these two routes result in the same applicative?

18. [★★] Show that if `M` is a monad implementation satisfying the monad laws then `Arrow_of_monad(M)` (Section 8.7.2) is an arrow implementation satisfying the arrow laws.

19. [★★] Show that if `A` is an arrow implementation satisfying the arrow laws then `Applicative_of_arrow(A)` (Section 8.7.3) is an applicative implementation satisfying the applicative 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 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 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)

- Hughes introduced arrows as a more general variant of monads in the following paper:

  Generalising monads to arrows
  John Hughes
  Science of Computer Programming (1998)

- 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 8.5.8, 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)