Haskell for OCaml programmers

Raphael Poss

March 2014

modified: 2022-05-02
slug: haskell-for-ocaml-programmers
category: Programming
tags: haskell, ocaml, functional programming, language comparison, analysis, programming languages

This post mirrors OCaml for Haskellers from Edward Z. Yang (2010).

Contents

Prologue 2

Why you should learn Haskell 2

Straightforward equivalences 4

Recursive definitions 6

Program structure 7

Program and effect composition 9

Print debugging 10

Mutable variables 11

Arrays 12

Type classes in a nutshell 12

Rolling your own recipe system 14

Haskell limitations 15

External links **16**
Searching for more information 16
Further reads 16
References 17

Copyright and licensing **17**

> **Note**
> The latest version of this document can be found online at https://dr-knz.net/haskell-for-ocaml-programmers.html. Alternate formats: Source, PDF.

Prologue

Why write a new post when a clever reader could simply "read Edward's post backwards"?

It's about the different audience, really.

My experience is that programmers well-versed in Haskell, or who learn Haskell as first language, tend to have a strong background in mathematics and logical reasoning. For this audience, the abstract equivalences between Haskell and OCaml are trivial and do not bear repeating. For them, a post like Edward's that mosty focuses on the "limitations" of OCaml compared to Haskell, and provides succinct high-level descriptions of OCaml's unique features, is sufficient.

In contrast, an OCaml programmer is often someone who has learned OCaml by iterative widening of their programming toolbox from other languages, perhaps without particular sensitivity to the aesthetics of abstraction. I once belonged to such an audience; I have taught programming to it, too.

For practical OCaml programmers, I found that **it often works better to approach Haskell from its operational perspective**, rather than the formal/equational approach typically taken in Haskell tutorials. The merits of Haskell's powerful abstraction compositions, in this context, are a matter best left to self-discovery, later in the learning process.

For example, Haskell tutorials often focus early on non-strict evaluation and strong typing. This is an unfortunate starting angle for a seasoned OCaml programmer who already knows Haskell's type system very well (OCaml has mostly the same) and who is likely to care immediately about a clear time and space cost model for his/her code, an expectation broken at first sight by non-strict evaluation.

Why you should learn Haskell

There are two set of important features found in Haskell and not in OCaml: *killer features* and *acquired tastes*.

Killer features are those that can immediately enable better productivity, without incurring a feeling of loss from leaving OCaml's world:

- layout-based code structuring;

```
(* ML: begin/end needed to scope
   nested pattern matches *)
match v1 with
| 0 -> begin
        match v2 with
        | 1 -> "hello"
        | 2 -> "world"
       end
| 3 -> "yay"
```

```
{- HS: indentation disambiguates
   nested pattern matches -}
case v1 of
 0 -> case v2 of
       1 -> "hello"
       2 -> "world
 3 -> "yay"
```

- declarations after use;

```
v = sum 20
  where  {- no equivalent in OCaml -}
    sum n = n * (n - 1) / 2
```

- readable type interfaces directly next to declarations;

```
(* in mod.mli *)
val f : int -> int -> int
(* in mod.ml *)
let f = (+)
```

```
{- in Mod.hs -}
f :: Int -> Int -> Int
f = (+)
```

- operator and function overloading;

```
(* no ad-hoc polymorphism,
   (+) vs. (+.) *)
begin
  print_string
     (string_of_int (3 + 2));
  print_string
     (string_of_float (3.0 +. 0.14));
end
```

```
{- show and (+) are overloaded -}

do
 putStr
     (show (3 + 2))
 putStr
     (show (3.0 + 0.14))
```

- more flexibility on operator and function names;

```
let (^.) a b = a ^ "." ^ b
let v = "hello" ^. "world"

(* only punctuation can serve as
   infix operators *)

(* "." is not a valid prefix *)
```

```
(^.) a b = a ++ "." ++ b
v = "hello" ^. "world"

dot a b = a ^. b
v2 = "hello" `dot` "world"

(.) f g x = f (g x)
```

- configurable operator associativity and precedence.

```
let (/~) a b = a / b
let (-~) a b = a - b

(* all operators starting with the
   same character have the same
   associativity and precedence *)

let v = 12 /~ 3 -~ 1 /~ 2
   (* = (12 /~ 3) -~ (1 /~ 2) *)
   (* = 4 -~ 0 *)
   (* = 4 *)
```

```
(/~) a b = a / b
(-~) a b = a - b

infixr 7 -~   {- higher = tighter -}
infixr 6 /~   {- r for right-assoc -}

v = 12 /~ 3 -~ 1 /~ 2
   {- = 12 /~ ((3 -~ 1) /~ 2)  -}
   {- = 12 /~ (2 /~ 2) -}
   {- = 12 /~ 1 -}
   {- = 12 -}
```

"Acquired tastes" are Haskell features that a) can yield a significant productivity improvement in the long run b) can yield terrible performance and/or unreadable code when used naively and thus c) will require patience and practice until you start feeling comfortable using them:

- pure arrays;
- function definitions by equations over infinite data structures;
- type classes and custom Functor / Applicative / Monoid instances.

Some examples are given later below.

Straightforward equivalences

Naming of types:

```
int float  char string bool userDefined (* OCaml *)
Int Double Char String Bool UserDefined {- Haskell -}
```

Operators:

```
=   <>  ^   @   @@  +.  /.  *.  +.  (* OCaml *)
==  /=  ++  ++  $   +   /   *   +   {- Haskell -}

land lor lxor [la]sl [la]sr lnot       (* OCaml *)
.&.  .|. xor  shiftL shiftR complement {- Haskell -}
```

Functions:

```
let f x y = x + y
let g = fun x y -> x + y

let rec fact n =
    if n = 1 then 1
    else n * fact (n-1)

let rec fact' = function
  | 1 -> 1
  | n -> n * fact' (n-1)
```

```
f x y = x + y
g = \x y -> x + y

fact n = {- no "rec" needed -}
    if n = 1 then 1
    else n * fact (n-1)

{- equational: order matters -}
fact' 1 = 1
fact' n = n * fact' (n-1)
```

Unit:

```
(* val f : unit -> int *)
let f () = 3
```

```
f :: () -> Int
f () = 3
```

Pattern match and guards:

```
match e with
 | 0 -> 1
 | n when n > 10 -> 2
 | _ -> 3
```

```
case e of
 0 -> 1
 n | n > 10 -> 2
 _ -> 3
```

Lists:

```
let rec len = function
   | [] -> 0
   | x::xs -> 1 + (len xs)
let v = len ([1; 2] @ [2; 1] @ 3::4::[])
(* v = 6 *)
```

```
len [] = 0
len (x:xs) = 1 + (len xs)

v = len ([1, 2] ++ [2, 1] ++ 3:4:[])
{- v = 6 -}
```

Parametric types:

```
(* type parameters use an apostrophe *)
(* tuple types defined with "*" *)
type 'a pair = P of ('a * 'a)

(* concrete parameter
   before abstract type *)
type t = int pair
```

```
{- type parameters use small letters -}
{- tuple types defined by commas -}
data Pair a = P (a, a)

{- concrete parameter
   after abstract type -}
type T = Pair Int
```

Algebraic data types:

```
(* predefined *)
type 'a option =
     | None
     | Some of 'a

(* custom *)
type 'a tree =
     | Leaf of 'a
     | Node of 'a tree * 'a tree
```

```
{- predefined -}
data Maybe a =
     | Nothing
     | Just a

{- custom -}
data Tree a =
     | Leaf a
     | Node (Tree a, Tree a)
```

Generalized algebraic data types:

```
(* available from OCaml 4.x *)
type _ term =
  | Int : int -> int term
  | Add : (int -> int -> int) term
  | App : ('b -> 'a) term * 'b term
          -> 'a term

let rec eval : type a. a term -> a =
function
  | Int n    -> n
  | Add      -> (fun x y -> x+y)
  | App(f,x) -> (eval f) (eval x)

(* two : int *)
let two =
    eval (App (App (Add, Int 1), Int 1))
```

```
{- compile with -XGADTs -}
data Term a where
  Int :: Int -> Term Int
  Add :: Term (Int -> Int -> Int)
  App :: (Term (b -> a), Term b)
         -> Term a

eval :: Term a -> a

eval (Int n)    = n
eval (Add)      = (\x y -> x + y)
eval (App(f,x)) = (eval f) (eval x)

two :: Int
two =
    eval (App (App (Add, Int 1), Int 1))
```

Text I/O, string representation and interpretation (yay to overloading):

```
string_of_int string_of_float          (* OCaml *)
show          show                     {- Haskell -}

int_of_string float_of_string          (* OCaml *)
read          read                     {- Haskell -}

print_char print_string print_endline  (* OCaml *)
putChar    putStr       putStrLn       {- Haskell -}

input_line input_char                  (* OCaml *)
getLine    getChar                     {- Haskell -}
```

Functional goodies:

```
(* "@@" is low-priority application *)
print_endline @@ string_of_int 3

(* can be defined in OCaml *)
val flip : ('a -> 'b -> 'c)
           -> 'b -> 'a -> 'c
let flip f = fun x y -> f y x

val (@.) : ('b -> 'c) -> ('a -> 'b)
           -> 'a -> 'c
let (@.) f g = fun x -> f (g x)
```

```
{- "$" is low-priority application -}
putStrLn $ show 3

{- built-in in Haskell -}
flip :: (a -> b -> c)
        -> b -> a -> c
flip f = \x y -> f y x

(.) :: (b -> c) -> (a -> b)
       -> a -> c
(.) f g = \x -> f (g x)
```

Recursive definitions

In OCaml, recursive definitions are expressed with the keyword `rec`. In Haskell, *all definitions are recursive*, including those of variables. This means that one cannot use the same identifier to bind to successive definitions in the same scope:

```
let f n =
    (* this defines a fresh "n" *)
    let n = n - 1 in

    n
```

```
f n =
    {- must use different identifiers -}
    let n' = n - 1 in

    n'
```

Otherwise, the Haskell compiler will generate a circular data definition for `n`, which is usually not the intended result!

Program structure

In OCaml, a *program* is defined by the concatenation of all the statements in the source code; the control entry point at run-time is the first statement in the top-level module. There is no function with a special role as entry point. Execution proceeds by evaluating all statements in order.

```
let f =
    print_endline "start";
    fun () -> "world"

let p1 () =
  print_endline "hello"
let p2 () =
  print_endline (f ())

p1 (); p2 ()

(* prints "start", "hello", "world" *)
```

In Haskell, a "program" must be defined by a *constant* (not function) with the special name "**main**". The run-time system starts execution by first constructing the data representations of all top-level definitions; including the definition of the constant "main", which stores a *data structure* akin to a "list of statements", or "recipe". At that point, there cannot be any effects performed yet. Then the run-time system continues execution by reducing the recipe referenced by main, step by step from the head, performing the effects described by the statements therein.

To build such a "list of statements" or "recipe", Haskell provides a shorthand syntax using the **do** keyword and newline/semicolon separators:

```
{- no side-effects in top-level definitions -}
f () = "world"

p1 = do {- p1 is a constant recipe -}
  putStrLn "hello"
p2 = do {- p2 is another constant recipe -}
  putStrLn (f ())

{- entry program must be called "main" -}
{- it is also a constant recipe -}
main = do p1; p2
```

In OCaml, all statements in source files are executed when the corresponding modules are imported. In Haskell, source code files provide only definitions; there are no side-effects performed during module imports.

```
(* in t.ml *)

let _ =
    print_endline "hello"

(* in main.ml *)
open T
let main =
    print_endline "world"

(* prints "hello", then "world" *)
```

```
{- in T.hs -}
module T where
_ = do
    putStrLn "hello"

{- in main.hs -}
import T
main = do
    putStrLn "world"

{- prints only "world", not "hello" -}
```

This is because only the constant recipe constructed by the "main" definition will be evaluated for side-effects at run-time, after all top-level definitions are constructed.

In OCaml, a file named "`blah.ml`" defines a module named "Blah". The module name is derived from the file name. In Haskell, a **module** specification in the file specifies the module name:

```
module Blah where ...
```

By convention, programmers also name the file containing "`module Blah where ...`" with filename `Blah.hs`. However, it is possible to split a Haskell module definition in multiple source files with different names, or define multiple Haskell modules in the same source file[1].

Module use:

```
open Char

(* "lowercase" imported by
   "open" from Char *)
let lc = lowercase

(* no explicit import needed
   to use qualified module names *)
let lsz  = List.length
```

```
import Char

{- "toLower" imported by
   "import" from Char -}
lc = toLower

{- need "import qualified" to use -}
import qualified Data.List

lsz = Data.List.length
```

It is possible to import a module in Haskell (OCaml's "open") without importing all the names it defines. To include only specific identifiers, place them between parentheses after the module name; to exclude specific identifiers, use "`hiding`":

```
import Char (toUpper,toLower)
import Char hiding (isAscii)
```

The set of predefined functions and operators in OCaml is provided by the module `Pervasives`. In Haskell, they are provided by module `Prelude`. It is possible to disable predefined functions using `import Prelude hiding (...)`.

Contrary to OCaml, it is not possible in Haskell to import/open a module locally to a function. Also, there are no direct Haskell equivalents for OCaml's parametric modules (functors). See Haskell limitations below for suggestions.

[1] This opportunity is specified in the Haskell Language Report, the official specification of the Haskell language. However, in practice, its most popular implementation GHC only properly handles modules where the filename matches the `module` specification therein.

Program and effect composition

Composition of pure functions in Haskell is conceptually the same as in OCaml. OCaml predefines "`|>`" (reverse pipeline), Haskell predefines "`.`" (composition).

```
let p x = f (g (h x))
let p x = f @@ g @@ h @@ x

(* "|>" predefined in OCaml *)

let p x = x |> h |> g |> f

(* "." not predefined in OCaml *)
let (@.) f g = fun x -> f (g x)

let p x = (f @. g @. h) x
```

```
p x = f (g (h x))
p x = f $ g $ h $ x

{- "|>" not predefined in Haskell -}
(|>) x f = f x
p x = x |> h |> g |> f

{- "." predefined in Haskell -}

p x = (f . g . h) x
```

To compose effectful statements with no result, both OCaml and Haskell use semicolons (for Haskell, inside a do-block). Additionally in Haskell, a newline character is also a valid statement separator in a do-block.

```
val f : int -> unit
val g : int -> unit

let _ =
    (* sequence of unit calls *)
    f 3 ; f 4 ;

    g 2
```

```
f :: Int -> IO ()
g :: Int -> IO ()

main = do
    {- sequence of unit statements -}
    f 3 ; f 4 {- \n separates too -}

    g 2
```

In OCaml, effectful functions that return a value of type `t` are declared with `t` as return type.

In Haskell, there are no effectful functions; only functions that return "lists of statements", or recipes, as described earlier. This creates a conceptual level of indirection: there are no "functions that perform an effect and return a value of type `t`", but rather "functions that return recipes, so that the subsequent evaluation of those recipes performs effects and also produces a value of type `t`".

A function that returns a recipe is typed with return type "`IO t`", which means "a recipe of statements that computes a value of type `t` upon its later evaluation".

To express the production of the computed value inside the do-block, you can use the handy function **return**:

```
(* inc : int -> int  (with effects) *)
let inc n =
    print_endline @@ string_of_int n;
    n + 1

(* fact : int -> int  (with effects) *)
let fact =
    let rec factr r n =
       print_endline @@ string_of_int n;
       if n = 1 then r
       else factr (n * r) (n - 1)

    in factr 1
```

```
inc :: Int -> IO Int
inc n = do
       putStrLn $ show n
       return (n + 1)

fact :: Int -> IO Int
fact =
    let factr r n = do
       putStrLn $ show n
       if n == 1 then return r
       else factr (n * r) (n - 1)

    in factr 1
```

In OCaml, you can bind a variable to the return value of an effectful function with "`let`", as usual. In Haskell, another syntax is defined for this purpose using "`<-`" in a do-block:

```
(* fact : int -> int  (with effects) *)

let _ =
    let v = fact 3;
    print_endline @@ string_of_int v

(* prints 6 *)
```

```
fact :: Int -> IO Int

main = do
    v <- fact 3
    putStrLn $ show v

{- prints 6 -}
```

Haskell's insistence on separating pure functions from "lists of effectful statements that produce values at run-time" creates a plumbing problem that does not exist in OCaml. Say, for example, you have two effectful functions like `inc` and `fact` above, that both take a *value* as argument and print something before producing their result. How to compose them together?

Direct composition works in OCaml (because both return `int`, which matches their input argument type), but not in Haskell. Instead in Haskell we must either explicitly bind the return value of the first effectful definition to a name with "`<-`", or use the operators "`=<<`" and "`>>=`" (piping of effects):

```
(* inc : int -> int   (with effects) *)
(* fact : int -> int  (with effects) *)

let _ =
   let p = (fact (inc 2));
   let q = (fact @@ inc 2);
   let r = (inc 2 |> fact);

   let s = (2 |> inc |> fact)
```

```
inc :: Int -> IO Int
fact :: Int -> IO Int

main = do
   tmp <- inc 2; p <- fact tmp;
   q <- (fact =<< inc 2)
   r <- (inc 2 >>= fact)

   s <- ((return 2) >>= inc >>= fact)
```

Print debugging

The functional engineer's nightmare: "what actual argument values is this function really called with at run-time?"

In OCaml, one can readily interleave "print" statements with the functional code to trace what happens at run-time. In Haskell, there is an extra issue because "print" statements are only available in the context of the special recipes with type `IO a`, for example constructed by the do-block syntax, and these do not fit type-wise in pure functions with regular non-`IO` types.

There are two ways to achieve this in Haskell. The "pure and magic" way, and the "impure but obviously effective" way.

The "pure and magic" way is a function called **trace** in the library module `Debug.Trace`. This function has type `String -> a -> a`, and will print its first argument during evaluation and return its second argument as result. From the language's perspective, this function is pure:

```
import qualified Debug.Trace
fact n =
    let n' = Debug.Trace.trace ("fact: " ++ (show n)) n in
    n' * fact (n' - 1)
```

In some cases, one may want to do other effectful things beside printing text. For example, we may want to print a timestamp or log output to file. Besides the other functions from `Debug.Trace`, you can roll your own tracing utility using the special function `unsafePerformIO`:

```
let f args ... =

   print_string ...;

   (... value computation ...)
```

```
let f args... =
   unsafePerformIO $ do
     putStr ...

     return (... value computation ...)
```

`unsafePerformIO` takes as single argument a recipe as constructed eg. by a do-block. When the enclosing expression is evaluated at run-time, the recipe is first evaluated for effects and then the final value, given to "`return`" within the do-block, becomes the functional value of the enclosing expression. `unsafePerformIO` has type `IO a -> a`.

This is the "impure but obviously effective" way:

```
(* no import needed *)


(* add : int -> int -> int *)
let add a b =

      print_endline @@ "add: "
                ^ (string_of_int a)
                ^ ", "
                ^ (string_of_int b);
      (a + b)

(* v1 : int *)
let v1 = add 1 1

let _ =
      print_string "hello";
      print_int v1;
      print_int @@ add 2 3

(* prints add, hello, add *)
(* v1 is evaluated where defined *)
```

```
import System.IO.Unsafe
       (unsafePerformIO)

add :: Int -> Int -> Int
add a b =
      unsafePerformIO $ do
          putStrLn $ "add: "
                ++ (show a)
                ++ ", "
                ++ (show b)
          return (a + b)

v1 :: Int
v1 = add 1 1

main = do
     putStr "hello"
     putStr . show $ v1
     putStr . show $ add 2 3

{- prints hello, add, add -}
{- v1's def is not an evaluation! -}
```

Despite its name, this construction is quite safe to use and it properly witnesses the evaluation of the enclosing expression at run-time. It is called "unsafe" because it breaks the usual convention that non-`IO` functions are pure. However, in the particular case of print debugging the purity is practically preserved since the I/O operations do not change the value of the function.

It is bad practice (and strongly frowned upon) to write a Haskell function using `unsafePerformIO` that is declared pure via a non-IO type but whose run-time return value is dependent on run-time side effects other than its arguments.

(`unsafePerformIO` is a bit of a taboo. Haskell programmers do not talk about it to "outsiders", in the same way that OCaml programmers do not talk about `Obj.magic`. But the engineer's life would be miserable without it. Just be careful.)

Mutable variables

OCaml provides the `ref` type for mutable references to values. In Haskell, the type `Data.IORef.IORef` does the same:

```
let _ =
    let v = ref "hello";
    let s = !v in print_endline s;
    v := "world";

    let s = !v in print_endline s
```

```
main = do
   v <- newIORef "hello"
   s <- readIORef v; putStrLn s
   writeIORef v "world"

   s <- readIORef v; putStrLn s
```

The Haskell library ArrayRef provides some syntactic sugar to simplify the use of mutable references.

Like in OCaml, the behavior of concurrent access to mutable variables by different threads in Haskell is not properly defined by the language. Just avoid data races on `IORef` and use Haskell's `MVar` type to communicate between threads instead.

Arrays

OCaml's arrays are mutable; although they are less often used, Haskell supports mutable arrays too. Since in-place array updates are effectful, in Haskell they must be used in the context of recipes, eg. inside do-blocks:

```
let _ =
    let a = Array.make 10 42;
    let v = a.(0);
    a.(2) <- v + v;
    print_int a.(2)

(* prints 84 *)
```

```
main = do
    a <- newArray (0,9) 42
    v <- readArray a 0
    writeArray a 2 (v + v)
    putStrLn.show <<= readArray a 2

{- prints 84 -}
```

The Haskell library ArrayRef provides some syntactic sugar to simplify the manipulation of arrays.

Next to mutable arrays, Haskell provides a lot of different pure array types in different libraries. Unfortunately, using pure arrays is quite cumbersome at first. Only later, when you start to grasp that compositions of functions that operate on arrays are "flattened" before execution (thanks to non-strict evaluation), does it begin to make sense. On the plus side, you then can start to write powerful reusable abstractions with arrays. On the down side, your code then becomes unreadable.

Decidely, pure Haskell arrays are an acquired taste, one difficult to share with non-experts.

Type classes in a nutshell

A common task in software engineering is to advertise a set of services using an abstract interface that hides the internal implementation. For this purpose, OCaml programmers can use objects or parametric modules.

Since Haskell provides neither objects nor parametric modules, Haskell programmers rely on another mechanism entirely, called "type classes". Type classes are nothing like modules but they can help for encapsulation given the right mindset.

In a nutshell, Haskell type classes express a programming contract over a set of types (hence the name): that all types in the class, ie. its "instances", are guaranteed to provide some other related functions. Moreover, a class can also provide a default implementation for some of its functions.

The usual example is the `Eq` class, written as follows:

```
class Eq a where
    (==) :: a -> a -> Bool
```

```
(/=) :: a -> a -> Bool
(/=) x y = not (x == y)
```

This definition expresses the following: "for all types `a` in the class `Eq a`, there exist two operators `(==)` and `(/=)` that accept two arguments of type `a` and returns a boolean value. Moreover, the class `Eq` provides a default implementation of `(/=)` that uses the actual implementation of `(==)` by each particular instance."

Once a class is specified, a programmer can do two things with it: *define type instances* of the class, or *define functions over instances* of the class.

To define an instance of a class, you first define a type, then you define how the type belongs to the class. For example, one can first define a new type that implements Peano integers:

```
data Peano = Zero | Succ Peano
```

Then, express that `Peano` is a member of `Eq`:

```
instance Eq Peano where
    (==) Zero Zero = True
    (==) (Succ x) (Succ y) = (x == y)
    (==) _ _ = False
```

Once this definition is visible, automatically any expression of the form "`x /= y`" becomes valid, thanks to the default implementation of `(/=)` already provided by `Eq`.

Given a type class, one can also define functions over any possible instance of that class. For example, given `Eq`, one can write a function which for any three values, returns `True` if at least two are equal:

```
any2 :: (Eq a) => a -> a -> a -> Bool
any2 x y z = (x == y) || (y == z) || (x == z)
```

Notice the prefix "`(Eq a) =>`" in the type signature. This means that "the function `any2` is defined over any type `a` as long as `a` is an instance of `Eq`".

Type classes can inherit all the operations of another class, before it defines its owns. For example, the class `Ord` defines operator `(<=)` (less than or equal), but for this it requires `(==)` defined by `Eq`. This is written as follows:

```
class  (Eq a) => Ord a  where
    (<=)  :: a -> a -> Bool
    {- some other Ord definitions omitted here -}
```

Type classes can be used directly to implement type-safe "container" data structures over arbitrary other types. For example, an abstract interface for an associative array over arbitrary keys can be defined with:

```
class (Ord k) => Map mt k v where
    empty :: mt k v
    insert :: k -> v -> mt k v -> mt k v
    delete :: k -> mt k v -> mt k v
    (!) :: mt k v -> k -> v
```

This definition creates a new type class called "`Map`", whose type instances are all of the form "`mt k v`" (this means that all instances of `Map` must be parametric with two type parameters). It also requires that the 1st type parameter (`k`, the key type) of its instances be a member of class `Ord`. For each instance `mt k v`, `Map` provides four services `empty`, `insert`, `delete` and `(!)`.

Once this definition is visible, it becomes possible to implement one or more concrete associative array data structures, and make them instances of `Map` via suitable `instance` declarations. From the perspective of third-party modules, the only visible services of these concrete implementations are those provided by the class.

Rolling your own recipe system

The previous sections have emphasized how "lists of statements" or "recipes" must be defined as constant data structures by the program, and are later consumed by the run-time system during execution, starting from `main`, to produce effects.

Once you start to become comfortable with composing these recipes using do-blocks, the `<-` binding and the `(>>=)` and `(=<<)` operators, a question often arises: *is it possible to write one own's recipe type and interpreter?*

The answer is yes! To do so, you will need to define:

- a parametric type `M a` that represents a "recipe of statements that eventually produce a value of type `a`";
- a function `return :: a -> M a` that takes as argument a value, and constructs a recipe intended to produce that value later;
- a function `(>>=) :: M a -> (a -> M b) -> M b` that takes as arguments one recipe and a constructor for another recipe, and "connects" the two recipes together;
- optionally, one or more effectful statements of type `M a` that can be used in recipes next to `return`;
- optionally, an interpreter for objects of type `M a`, that performs its effects in the way you see fit.

Once you have defined the first three properly (see below), the do-block syntax presented earlier is automatically extended to your new type, inferring the right types from your custom definition of `return`.

Here are some useful recipe systems already available in Haskell:

- `IO a`: "recipes" evaluated by the Haskell run-time system for global effects during execution. The peculiarity of this one is that its implementation is completely hidden from the language; you can't redefine `IO` directly in Haskell. The effectful statements of this recipe system include `putStr` and `getChar`, already presented above.
- `Maybe a`: "recipes" that eventually provide a value of type `a`, but where the evaluation stops automatically at the first intermediate statement that returns `Nothing`. The effectful statements of this recipe system include `fail`, which forces `Nothing` to be generated during the effectful evaluation.
- `Writer l a`: "recipes" that eventually provide a value of type `a`, but where the evaluation keeps a log of messages generated by the statements. The effectful statements of this recipe system include `tell`, which generates a message for the log during effectful evaluation.
- `State s a`: "recipes" that eventually provides a value of type `a`, but where each intermediate statement can modify an internal value of type `s` (a state). The effectful statements of this recipe system include `get` and `set`, which can access and modify the internal state during effectful evaluation.

About the M-word: There is a word used in the Haskell community which starts with the letter "M" and causes a great deal of confusion and misunderstandings. I wish to avoid using the M-word entirely in this document. I believe that using and understanding the M-word is unnecessary to learn how to write Haskell programs productively.

Nevetherless, you should understand that whenever you read something about the M-word, it really refers to what I explained in this section. When you read "the type `T a` is a M…", it really means that "the type `T a` describes recipes of statements that produce values of type `a` during evaluation" and also that "the type `T a` is an instance of the type class `M…`, which provides the services `return` and `(>>=)`".

Likewise, when you read or hear "let's define a M…", this simply refers to the act of writing a definition for a new custom recipe type and two new functions `return` and `(>>=)`, and then using `instance` to declare an instance of the `M` class with them.

The reason why Haskell programmers *eventually* end up caring a great deal about the M-word, in the same way they end up caring about the `Applicative`, `Functor` and `Monoid` classes, is that there are very good software components that can be defined using only the services of these classes. This means these software components are greatly reusable, because they apply automatically to all types later declared to be instances of these classes.

Haskell limitations

Of course, there is also a price to pay. Haskell was not primarily designed to be a functional language by engineers for engineers. There are three features that OCaml gets "just right" and are unfortunately completely missing in Haskell:

- parametric modules and module composition;
- named and optional function parameters;
- polymorphic variant types.

Haskell practitioners eventually develop some other conventions ("best practices") to achieve parametric modularity without parametric modules. This usually involves mixing and matching the following features:

- preprocessing using the C preprocessor (`ghc -cpp`), which can be used to mimic parametric (but not composable) modules in the same way as in C;
- record types, placing the module's type definition in ghost type parameters of a record type, and the module's methods in its fields;
- type classes, eg. to replace OCaml's `Map` functor and `OrderedType` parameter, instantiated as `IntMap`, `BoolMap`, etc., by `Map` and `Ord` type classes (as suggested above).

Named and optional function parameters, in practice, become less important once the Haskell practioner knows how to fully leverage higher-order functions and non-strict evaluation. A programming technique to implement optional function arguments using record types and default record fields was described by Neil Mitchell in 2008.

Polymorphic variant types (`[> `A | `B]`) are, unfortunately, without direct equivalent in Haskell. Haskell is a "closed world language" where type inference can only succeed once all types from all compilation units are known. However, a language extension from the GHC compiler called Type families can be abused to provide somewhat similar functionality as OCaml's polymorphic variants.

External links

Searching for more information

- The Haskell wiki.
- The Haskell Wikibook.
- Hoogle: an API search engine, able to search by function name or by type signature.
- The `#haskell` IRC channel on FreeNode.
- The GHC Manual.

Further reads

- Paul Hudak, John Peterson, Joseph Fasel, and Reuben Thomas. A Gentle Introduction to Haskell, Version 98. June 2000. Also known as the "official Haskell tutorial".
- Miran Lipovaca. Learn You a Haskell for Great Good!. April 2011. Also known as "the funkiest way to learn Haskell", but a very pleasant read.
- Bryan O'Sullivan, Don Stewart, John Goerzen. Real World Haskell. November 2008. With lots of practical examples.
- Edward Z. Yang, Resource limits for Haskell. April 2013. Explains how to monitor and control space usage in Haskell functions, including during non-strict evaluation on parallel computers.
- Robert Harper, Persistence of Memory, April 2011. Promotes the use of persistent (immutable) data structures, and argues that the common debate about run-time efficiency of mutable vs. immutable data structures is often misdirected.
- Robert Harper, The Point of Lazyness, April 2011. Argues that non-strict evaluation is desirable even in eager languages, at least to manage processes and streams elegantly.
- Lennart Augustsson, More points for lazy evaluation, May 2011. Argues that non-strict evaluation promotes reuse of software components.
- Edward Z. Yang, So you want to add a new concurrency primitive to GHC..., January 2014. Explains how GHC's Haskell still has conceptual issues internally with the notion of mutable memory.
- Andreas Voellmy, Junchang Wang, Paul Hudak and Kazuhiko Yamamoto. Mio: A high-performance multicore IO manager for GHC. September 2013. Explains how GHC implements `IO` internally, and how it was recently extended to better support parallel execution.
- Ben Rudiak-Gould, Alan Mycroft and Simon Peyton Jones. Haskell is Not Not ML. 2006. Suggests that there is an underlying common language behind Haskell and SML, that can run programs written in either. Also discussed by Ehud Lamm here.

References

- Edward Z. Yang, OCaml for Haskellers, October 2010.
- Xavier Leroy et al., The OCaml system release 4.01, September 2013.

Copyright and licensing

Copyright © 2014-2026, Raphael Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.