Understanding Monads via C++

Jeff Cohen — ACCU 05/08/2019

What is a Monad?

~~A monad is~~
~~a simple application~~
~~of~~
~~algebraic category~~
~~theory.~~

Here, monads will be explained by writing the equivalent C++!

But first: Haskell's view of monads

```
class Functor f where
    fmap   :: (a -> b) -> f a -> f b
```

A type class declaration (with type kind variable f)

Sort of like abstract virtual methods (with additional type variables a and b)

```
class Functor f => Applicative f where
    pure   :: a -> f a
    (<*>)  :: f (a -> b) -> f a -> f b
```

Every Applicative is also a Functor (think C++ template requires clause)

Declare an operator named <*>

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

    return = pure
```

Lifts a value into the monad (think C++ constructor)

Bind operator: applies a computation to the value contained within the monad

Default implementation

A simple monad: Maybe

```
data Maybe a = Nothing | Just a
```

Equivalent to std::optional<a>
(but "Maybe a" is not a template)

```
instance Functor Maybe where
    fmap _ Nothing    = Nothing
    fmap g (Just x)   = Just (g x)
```

```
instance Applicative Maybe where
    pure              = Just
    Nothing  <*> _    = Nothing
    (Just g) <*> mx   = fmap g mx
```

```
instance Monad Maybe where
    Nothing  >>= _    = Nothing
    (Just x) >>= f    = f x
```

Type classes are a higher-order type, just like C++ concepts.

Instance declarations are sort of like a concept_map, if C++20 still had concept_maps, and *if they supported runtime polymorphism.*

Haskell is not object-oriented. It doesn't have classes in the C++ sense, with object instances that have a vtbl pointer. Instead, Haskell "unbundles" vtbls from values via its type class mechanism.

Example: Safe Division

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

f w x y z = do v1 <- w `safediv` x
               v2 <- v1 `safediv` y
               v2 `safediv` z

main = do
    putStrLn (show (f 100 1 2 5))
        -- Just 10
    putStrLn (show (f 100 0 2 5))
        -- Nothing
```

Safe division returns Nothing if the divisor is zero.

Function f divides w by, successively, x, y and z. Upon encountering a zero, no further division takes place and Nothing is returned.

The do block *looks* like a procedural series of assignment statements, but nothing is further from the truth. It is syntactic sugar for a series of monadic bind (>>=) function calls.

Note that function types are optional. If omitted, it is inferred by the compiler, both from its implementation and from how it's used—including how the return value is used. This will compile even if all function types are omitted.

Do Blocks

```
f w x y z = do v1 <- w `safediv` x
               v2 <- v1 `safediv` y
               v2 `safediv` z


f w x y z = (w  `safediv` x) >>= \v1 ->
            (v1 `safediv` y) >>= \v2 ->
            (v2 `safediv` z)


instance Monad Maybe where
    Nothing  >>= _  = Nothing
    (Just x) >>= f  = f x
```

These two implementations of f are semantically identical.

Each "assignment" in the do block is turned into a call of the monadic >>= (bind) function. The right-hand argument is a lambda that encapsulates all remaining code in the do block.

The implementation of >>= may execute the lambda as many times as it likes, including zero times. For the Maybe monad, it is executed if and only if the monad's value is not Nothing.

The Maybe monad in C++

```
template<typename T, typename F>
auto operator>>(std::optional<T> value, F f) -> decltype(f(T())) {
  if (value)
    return f(*value);
  else
    return decltype(f(T())){};
}

void f(std::optional<int> v) {
  auto r = v >> [](auto x) { return std::optional<std::string>{std::to_string(x)}
             >> [](auto y) { return std::optional<std::string>{y + y}
             >> [](auto z) { return std::optional<int>{std::stoi(z, nullptr) / 2}; };
       }; };

  if (r)
    std::cout << "Just " << *r << "\n";
  else
    std::cout << "Nothing\n";
}
```

Yeah, that was a thing…

- And yet, it still fails to capture the full semantics of Haskell.
- Templates were used in the name of maximum type inferencing, but not only did that fall short—return types could not be inferred—standard Haskell does not have templates!
- How can we properly represent type classes, i.e. runtime parametric polymorphism, in C++?

Runtime Polymorphism

```haskell
showFirst :: (Show a) => [Maybe a]->[Char]
showFirst [] = "nothing!"
showFirst (x:xs) =
    case x of
        Nothing -> showFirst xs
        Just y  -> show y
```

Show is a standard type class, with instances defined for numerous types. It defines a function, show, that converts a value to its printable representation.

showFirst traverses a list of Maybe values, looking for the first something and showing it. It constrains its argument to Maybe values that are instances of Show (The constraint would be inferred if the function type was omitted).

This is not a template. Only one instance of this function is compiled to machine code. Values do not have a vtbl. So how is runtime polymorphism achieved?

C++ to the Rescue!

```
struct MaybeBase {
    enum { Nothing, Just } tag;

    virtual void *Value() = 0;
};

template<typename T>
struct Maybe: public MaybeBase {
    T value;

    virtual void *Value() {
        return &value;
    }
};

struct TC_Show {
    std::string (*show)(void *v);
};
```

```
std::string
showFirst(std::vector<MaybeBase *> &xs,
          TC_Show &tc_show) {
    for (auto x : xs) {
        if (x->tag == MaybeBase::Nothing)
            continue;
        return tc_show.show(x->Value());
    }
    return "nothing!";
}

std::string Show_Int(void *v) {
    int num = *reinterpret_cast<int*>(v);
    return std::to_string(num);
}

TC_Show show_Int = { Show_Int };
```

List Monad

```haskell
instance Monad [] where
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]

    return x = [x]
```

The list monad's bind operator executes its function argument once for each element of the list.

Note that the function is expected to return a list. These returned lists are concatenated together via the list comprehension.

The monadic return function lifts a value into the list monad by encapsulating it in a one-element list.

State Monad

```
-- "State" is the type of the
-- internal state
newtype ST a = S (State -> (a,State))

app :: ST a -> State -> (a,State)
app (S st) x = st x

instance Monad ST where
    -- (>>=) :: ST a -> (a->ST b) -> ST b
    st >>= f = S (\s -> let (x,s') =
                     app st s in app (f x) s')

    -- return :: a -> ST a
    return x = S (\s -> (x,s))
```

The state monad encapsulates “mutable” state. The bind operator provides access to a value derived from that state, while also computing a new state via its function argument.

Note that the state is **not** modified in place. A new state value is generated each time. The value produced by the bind operator is a tuple comprised of both the value derived from the old state and the newly updated state. An old state value may be re-used as many times as you like, because it is never modified.

Monads sequence evaluation, crucial for reading and updating state.

State Monad Example

```
type State = Int
newtype ST a = S (State -> (a,State))

postincr = S (\n -> (n, n + 1))
double = S (\n -> (n * 2, n * 2))

app (S st) x = st x

main =
    let v = do postincr -- (4,5)   (7,8)
               double   -- (10,10) (16,16)
               postincr -- (10,11) (16,17)
    in do putStrLn (show ((app v) 4))
          putStrLn (show ((app v) 7))

-- prints (10,11)
--        (16,17)
```

The do blocks, here, do not actually compute a value; they composes a series of state mutating functions into a single, giant function.

The expression ((app v) 4) first extracts that giant function, then applies it to the value 4. One post-increment, doubling, and post-increment later, that 4 turns into 10, with the final state being 11. Likewise, 7 turns into 16, with a final state of 17.

It is possible to thread state in complex ways, for example while visiting every node of a tree.

Another State Monad Example

```
data Tree a = Leaf a |
              Node (Tree a) (Tree a)
              deriving Show

label :: Tree a -> ST (Tree (Int,a))
label (Leaf x) = do n <- postincr
                    return (Leaf (n,x))
label (Node l r) = do l' <- label l
                      r' <- label r
                      return (Node l' r’)

tree = Node (Node (Leaf 'a') (Leaf 'b'))
            (Leaf 'c')

main =
    let tree' = fst (app (label tree) 0)
    in putStrLn (show tree')
```

The function label copies the tree that is its argument, labeling any Leaf nodes that it finds with successively larger integers. The counter is threaded as before via postincr inside the ST (Tree (Int,a)) monad.

The original tree has type Tree Char; label doesn’t care what the value type, a, is as it only copies the values.

The function fst returns the first value of a tuple, in this case extracting the labeled tree from the final state.

The deriving Show clause in the data declaration auto-generates a suitable type class instance for the new type, allowing show tree' to work. The output is:

```
Node (Node (Leaf (0,'a')) (Leaf (1,'b')))
     (Leaf (2,'c'))
```

IO Monad

```
instance Monad IO where
    -- (>>=) :: IO a -> (a->IO b) -> IO b
    mx >>= f = ... -- internally defined

    -- return :: a -> IO a
    return x = ... -- internally defined

-- The main function must have this type:
main :: IO ()
```

In one sense, the IO monad is just like the state monad—it encapsulates state—but with one huge difference: the state that it encapsulates is “the real world.” That state **must** be updated in place!

It is through the IO monad that a pure, effect-free Haskell program is able produce effects. These effects **cannot escape** the IO monad and thus contaminate the functional purity of the rest of the code. *The type system guarantees it*.

Not surprisingly, the >>= and return functions cannot be implemented in standard Haskell.

The main function must return the IO monad. The runtime, in “using” this value, calls the function created and returned by main, thus effectively executing the program.

The Purpose of Monads

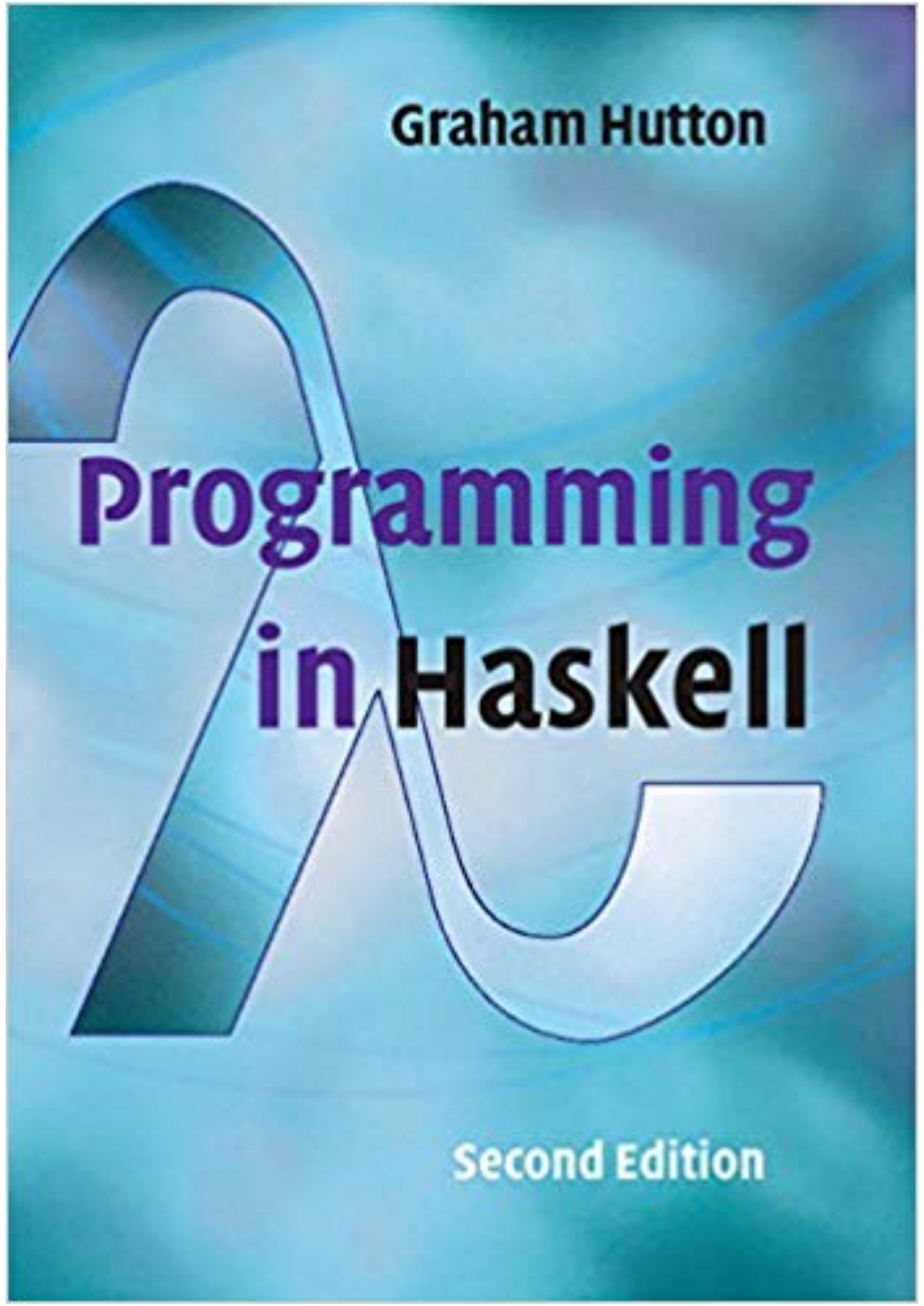
- Monads were invented to solve a set of problems that Haskell created for itself:
    - By being a pure, functional language with absolutely no side-effects.
    - All values, without exception (outside the IO monad), are immutable.
    - Lazy (non-strict) evaluation. An expression is generally not evaluated unless and until its value is needed.
    - As a consequence, expression evaluation order is not defined by syntax.
- Monads solves these problems by:
    - Explicitly sequencing evaluation, imposing an evaluation order.
    - Encapsulating mutable state in such a way that functional purity is maintained.

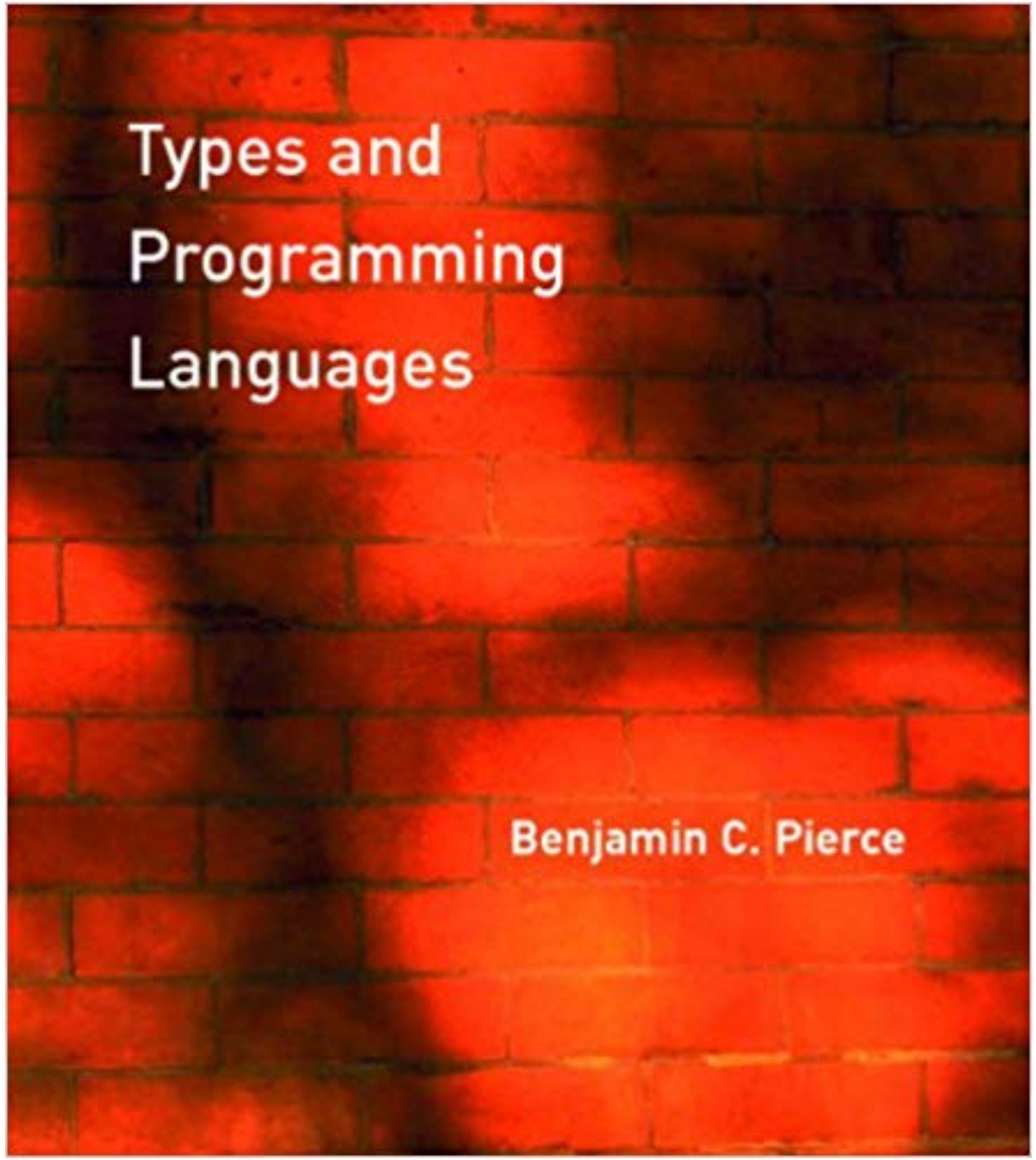
Why Should C++ Care About Monads?

- After all, C++ does not suffer from Haskell’s self-imposed problems.
  - It’s drowning in side-effects.
  - Objects and variables are ecstatically mutable.
  - Evaluation order is (mostly) defined by syntax.
- But monads are also a powerful mechanism for abstracting computation.
- Consider the Maybe monad, how a Nothing value short-circuits subsequent evaluation, all **without any boilerplate code**.
- Short-circuits? Wait. Doesn’t C++ already have two operators, && and ||, that do that?

Because C++ Already Has Them!

- What a twist! C++ already has monads!
- Exceptions are another monad. Consider all the boilerplate code *that* eliminates!
- But those monads are ad-hoc and specialized. Haskell provides a general solution.
- There is at least one proposal to add a better monadic solution to C++: http://wg21.link/p0798 (Monadic operations for std::optional).


Graham Hutton
Programming in Haskell
Second Edition


Types and Programming Languages
Benjamin C. Pierce

Questions