The Curse of the Excluded Middle

“Mostly functional” programming does not work

Erik Meijer

There is a trend in the software industry to sell “mostly functional” programming as the silver bullet for solving problems developers face with concurrency, parallelism (manycore), and, of course, Big Data. Contemporary imperative languages could continue the ongoing trend, embrace closures, and try to limit mutation and other side effects. Unfortunately, just as “mostly secure” does not work, “mostly functional” does not work either. Instead, developers should seriously consider a completely fundamentalist option as well: embrace pure lazy functional programming with all effects explicitly surfaced in the type system using monads.

Like dieters falling for magic 10-minute-miracle exercise gadgets, developers seem ready to fall for easy solutions to the latest crises in their field. Recently, many are touting “nearly functional programming” and “limited side effects” as the perfect weapons against the new elephants in the room: concurrency and parallelism. Few want to accept that the benefits necessarily come at the cost of the convenience of latent effects in common operations such as I/O, just as dieters would rather not admit that the benefits of exercise necessarily come at the cost of time and sweat.

Just like “mostly secure,” “mostly pure” is wishful thinking. The slightest implicit imperative effect erases all the benefits of purity, just as a single bacterium can infect a sterile wound. On the other hand, radically eradicating all effects—explicit and implicit—renders programming languages useless. This is the curse of the excluded middle: you must confront effects seriously by either (a) accepting that programming is ultimately about mutating state and other effects, but for pragmatic reasons tame effects as much as possible; or (b) abolishing all implicit imperative effects and making them fully explicit in the type system, but for pragmatic reasons allow occasional explicit effects to be suppressed.

THE PROBLEM

Imperative programs describe computations by repeatedly performing implicit effects on a shared global state. In a parallel/concurrent/distributed world, however, a single global state is an unacceptable bottleneck, so the foundational assumption of imperative programming that underpins most contemporary programming languages is starting to crumble. Contrary to popular belief, making state variables immutable comes nowhere close to eliminating unacceptable implicit imperative effects. Operations as ordinary as exceptions, threading, and I/O all cause as much hardship as simple mutable state. Consider the following C# example (thanks to Gavin Bierman) that filters an array to retain all values between 20 and 30:

```

static bool LessThanThirty(int x) {
    Console.Write("{0}? Less than 30;", x); return x < 30;
}
static bool MoreThanTwenty(int x) {
    Console.Write("{0}? More than 20;", x); return x > 20;
}

var q0 = new[]{ 1, 25, 40, 5, 23 }.Where(LessThanThirty);
var q1 = q0.Where(MoreThanTwenty);
foreach (var r in q1){ Console.WriteLine("{0}";r); }

```

Because `Where` is lazy (or uses deferred execution), the effects in the predicates used in `q0` and `q1` are interleaved; hence evaluating `q1` prints all values between 20 and 30 as if the predicates were intersected:

```

1? Less than 30; 1? More Than 20; 25? Less than 30; 25? More Than 20; [25];40? Less than
30; 5? Less than 30; 5? More Than 20; 23? Less than 30; 23? More Than 20; [23];

```

The average programmer would surely expect `q0` to filter out all values above 30 before `q1` starts and removes all values smaller than 20, because that's the way the program was written, as evidenced by the semicolon between the two statements. Any cross-dependencies between the predicates come as a nasty surprise.

Here is another example: mixing laziness with exceptions. If an exception is thrown in the body of a closure passed to the `Select` (map) function, because of deferred execution the exception is not thrown inside the scope of the try-catch handler. Instead, the exception is thrown once evaluation is forced by the `foreach` loop, with no handler in sight:

```

var xs = new[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
IEnumerable<int> q;
try { q = xs.Select(x=>1/x); } catch { q = new int[]; }
foreach(var z in q){ Console.WriteLine(z); // throws here }

```

Effects interfere with language features in other ways, such as the interplay between closures and the disposable pattern. The next example opens a file for reading and captures the file variable in a closure that escapes the lexical scope of the using block. In C# the `using` statement causes the variable initialized at the entry of the block to be automatically disposed of when control flow reaches the end of the block. Hence, by the time the closure is called, the file will have been disposed of, causing a surprising exception far away in time and space from where the exception-throwing code resides in the static source code:

```
Func<string> GetContents;
using(var file = FileSystem.OpenTextFileReader(@"my file")) {
    GetContents = ()=>file.ReadToEnd();
}
Console.WriteLine(GetContents()); // surprise! an exception
```

The lexical scope of try-catch and using blocks does not mix well with the dynamic extent of side-effecting closures.

DOWN THE RABBIT HOLE

If the examples with laziness and closures look farfetched, let's look at a method that in printf debugging style traces its return value to the console. As innocent as this may look, it is now unsafe for a compiler to perform optimizations as simple as common-subexpression elimination:

```
string Ha() { var ha = "Ha"; Console.Write(ha); return ha; }

// prints HaHa
var haha = Ha()+Ha();

// prints Ha
var ha = Ha();
var haha = ha+ha;
```

But wait—it is actually much worse. Simply creating a new object is an observable side effect. Even when called with the same arguments, a constructor will return a different result each time, as can be observed via the GetHashCode or ReferenceEquals method, among others:

```
var a = new Object();
//58225482, for instance
Console.WriteLine(a.GetHashCode());

var b = new Object();
// 54267293, for instance
Console.WriteLine(b.GetHashCode());

Debug.Assert(a != b);
```

To make constructors pure, you must insist on value semantics for all objects, which implies the elimination of all shared mutable state in objects. Now the essence of object-oriented programming is lost: encapsulation of state and behavior in a single unit.

These examples illustrate another harmful consequence of implicit imperative effects: they foreclose many common optimizing transformations. Since side effects implicitly alter the program's global environment, it is usually unfeasible to isolate and localize effects. As a result, imperative

programs are mostly noncompositional, making it hard for both programmers and compilers to reason about them.

Bad as this is, surely it is enough just to abolish state mutation to make the code pure. No! Unfortunately, it is not enough simply to make all fields in a class `readonly` and disallow assignments to local variables in order to tame the state monster. The C ω program here shows that threads can easily simulate state, even though there are no assignments or mutable variables anywhere to be seen in the code. The private channel `Value(T current)` carries the state of the cell, piggybacking on the hidden mutable state of the thread's message queue within the cell:

```
class Cell<T> {
  Cell<T>(T init){ Value(init); }
  T Get() & async Value(T current){ return current; }
  async Set(T @new) & async Value(T old){ Value(@new); }
}
```

You can define a mutable `Point` class using `readonly` fields of type `Cell`:

```
class Point {
  readonly Cell<int> x = new Cell<int>(0);
  readonly Cell<int> y = new Cell<int>(0);
}
```

Passing state around over a private channel is embraced as the *foundation* of active objects or actors in Erlang. Here is the Erlang version of the mutable `Cell` program above, using a tail-recursive function `cell(Value)` to hold the state of the `Cell`:

```
new_cell(X) -> spawn(fun() -> cell(X) end).
cell(Value) ->
  receive
    {set, NewValue} -> cell(NewValue);
    {get, Pid}      -> Pid!{return, Value}, cell(Value);
    {dispose}       -> {}
  end.
set_cell(Cell, NewValue) -> Cell!{set, NewValue}.
get_cell(Cell) ->
  Cell!{get, self()},
  receive
    {return, Value} -> Value
  end.
dispose_cell(Cell) -> Cell!{dispose}.
```

Note how this Erlang actor basically encodes an object with dynamic method dispatch using the pattern-matching, message-sending, and recursion primitives of the language, which you may

happily leverage to implement mutable references, sabotaging the fact that the Erlang language does not natively expose mutable state.

These examples are just the tip of the iceberg. The fundamental problem with side effects is that there are many uncontrollable ways to observe them, and even worse, it is often possible to simulate one effect with another. Adding recursion to pure combinatorial circuits makes it possible to build flip-flops that provide mutable state. You can prove that state and delimited continuations suffice to simulate any effect,² and that unchecked exceptions can simulate continuations.⁴ You cannot be careful enough in dealing with effects.

The examples just presented show how tricky and subtle side effects can be, popping up where they are least expected. Maybe, then, you might ask, “With care and discipline can you transform an imperative language into a pure one by avoiding features that cause problems?” To do this, you would have to remove *all* of its intrinsic effects. Even evaluation is an effect, and you must give up control of it to the compiler and runtime. The consequence is that such ultra-pure programs cannot interact with the user or the environment, cannot perform network I/O, react to user-interface events, read data from files, get the time of day, or generate random numbers. This is a terrible dilemma: If pure programs cannot leave any trace of ever being executed, then how can they be useful?

Though the situation seems bleak, fortunately there are several ways out of the pit. They all involve tastefully adding features instead of blindly removing them, as John Hughes observed in 1984 in his seminal paper, “Why Functional Programming Matters.”¹

FUNDAMENTALIST FUNCTIONAL PROGRAMMING

Pure functional programming is programming with mathematical functions. This means that the only way to express dependencies among values is by applying functions to arguments and harvesting values returned. Calling a function with the same arguments will return the same result every time. There is no way to keep a secret, hide a value in a little place to be picked up later, directly say do this before that, spin up a thread, throw an unchecked exception, or just print something to the console. This may seem rigid and fundamentalist. It is. But it is also powerful and enabling.

To understand how fundamentalist functional programming might help solve the concurrency problem, it is important to understand that it is not just imperative programming without side effects, which, as we have seen, is useless. Rather, it harnesses the fundamentalist language of mathematical functions to express and encapsulate rich effects compositionally using monads. To build an intuitive model of monads, let’s look at perhaps the simplest possible generic method:

```
T Identity<T>(T me) { ... }
```

The type signature of this method claims that *for all types T*, given an argument of type T, the method will return a value of type T. That is not the whole truth; the imperative type signature is *insouciant*, permissive, too loose. It expresses only an approximation of the actual *mathematical* type of the method. It does not account for any side effects that might be hiding in its execution, such as the fact that it eagerly evaluates its argument; it might inspect the generic type parameter (so it does not work uniformly with all types T); it might throw exceptions, perform I/O, spin up threads,

allocate new objects in the heap, and take a lock; it can access `this`, etc.

The trick of pure fundamentalist functional languages is to have obsessive-compulsive types and expose all effects explicitly in the type signatures using monads, distinguishing between the types of pure values and the types of computations on values that may entail effects.

INFORMAL INTRODUCTION TO MONADS

For a value of type `a`, let's write an effectful computation that can deliver a value of this type in the notation of a function application `(M a)`, the Haskell syntax for generic types. One way to envision this effectful computation is with a machine for producing output values of type `a`, while explicitly denoting the effect `M` caused by the computation of the output value. Think of a factory in the real world that needs electricity and pollutes the environment as a side effect of producing goods. Note that a value of type `M a` is just a promise to produce a value of type `a`, and no effect is yet performed. To actually do something with a monadic value, there are two standard combinators that work on such effectful computations:

- The *infix application function* `(ma>>= \a->f(a))`, commonly called *bind*, executes the computation `ma` to expose its effects, calling the resulting value `a`, and passes that to function `f`. Evidently, the result of this application `(f a)` should again be a potentially side-effecting computation; hence, the type signature for *bind* looks like this:

```
(>>=):: M a -> (a -> M b) -> M b.
```

- The *injection function* `(return a)` injects a pure value into a computation. Its type signature is:
`return :: a -> M a.`

In practice, each effect usually comes with so-called *nonproper morphisms*. These domain-specific operations are unique to that effect (see examples later in this article).

You can now abstract and formalize all effectful computations of the form above as an algebraic structure called a *monad* that supports the two generic operations for creating and propagating effects, *return* and *bind* `(>>=)`:

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

The mother of all monads in Haskell is the I/O monad, which represents all computations that have a global effect. As such, it comes with a large number of nonproper morphisms, most of which look familiar to imperative programmers, such as operations to read and write to the console, spinning up threads, and throwing exceptions. Once effects are explicit in, say, the I/O monad, operations such as allocating mutable variables, and reading and writing to them, must be lifted into the I/O monad.

```
data IO a -- no need to know how this is implemented
```

```
putChar :: Char -> IO ()
getChar :: IO Char
```

```
newIORef    :: a          -> IO (IORef a)
readIORef   :: IORef a    -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

```
forkIO :: IO a -> IO ThreadID
```

From the type of `forkIO`, you can immediately see that spinning up a thread is a side-effecting operation, which ensures that any encoding of mutable cells using threads will be in the I/O monad as well. This prevents programmers from believing they are defining immutable types where they are not. Note that the I/O monad does not prevent the global state to be updated simultaneously from several threads, as the type of `forkIO` shows.

The real power of monads comes from the fact that computations are themselves just values that can be passed around as first-class citizens within the pure host language. This enables programmers to write new abstractions (domain-specific and custom-control structures) using the monadic primitives and nonproper morphisms.

For example, you can easily define a function that takes a list of side-effecting computations and executes each, collecting the results in a list:

```
sequence      :: Monad m => [m a] -> m [a]
sequence []   = return []
sequence (ma:mas) = ma >=> (\a -> sequence mas >=> (\as -> a:as))
```

A pure fundamentalist functional language is a very convenient host language for internal DSLs since it is just executable denotational semantics. As such, pure functional languages come a long way toward fulfilling the vision of P.J. Landin's influential paper published in 1966, "The Next 700 Programming Languages."³

The additional beauty of making effects explicit in the type system is that you can distinguish among various effects and freely introduce new ones. For example, you can define the monad (`STM a`) of transactional memory.⁶ It contains nonproper morphisms for allocating, reading, and writing transactional variables:

```
newTVar    :: a          -> STM (TVar a)
readTVar   :: TVar a     -> STM a
writeTVar  :: TVar a -> a -> STM ()
```

The `retry` operation aborts the current transaction and waits until any transactional variables are changed, or it attempts to run another transactional block when chained via the `orElse` operator.

```
retry  :: STM a
orElse :: STM a -> STM a -> STM a
```

Finally, the function `atomic` injects the transaction monad into the general I/O monad. From that point on, no more transactional operations can be performed on the value.

```
atomic :: STM a -> IO a
```

Since there is no (implicit) leakage from I/O to STM, the STM monad captures all the mechanisms needed to perform transactions without having to worry about rolling back arbitrary effects, since ordinary Haskell computations are assumed to be pure.

Side effects are not only hard to tame, but they also are often sneaking in the back door. Even in the supposedly pure Haskell, there is a seemingly inconspicuous function called `unsafePerformIO :: IO a -> a` that tells the compiler to “forget” about the side effects involved in evaluating its argument. This “almost function” is supposed to facilitate encapsulating benevolent side effects in an otherwise pure computation; `unsafePerformIO`, however, opens a Pandora’s box because it subverts the Haskell type system, wrecking language guarantees and allowing any type to be cast to any other type:

```
unsafeCast :: a -> b
unsafeCast x = unsafePerformIO (do{
    writeIORef castref x; readIORef castref
})
castref = unsafePerformIO (do{ newIORef undefined })
```

These examples show some of the power of treating side-effecting computations as values using monads, but the question of the average developer can deal with them remains unanswered. We believe the answer is a resounding yes, as confirmed by the fact that monads are also just what make LINQ tick. The LINQ Standard Query Operators are essentially monadic operators. For example, the `SelectMany` extension method directly corresponds to the Haskell `bind (>>=)` introduced earlier:

```
(>>=) :: m a -> (a -> m b) -> m b
M<T> SelectMany(this M<S> src, Func<S, M<T>> f)
```

A major difference between polymorphism in Haskell and generics in CLR (Common Language Runtime), Java, or other object-oriented languages is that the Haskell definition of monads relies on higher-kinded polymorphism—that is, the monad class (interface) is parameterized by a type constructor instead of by a type. Usually, generics allow parameterization only over types—for example, `List<T>` and `Array<T>`—but do not allow parameterization over the container type `M` as in `M<T>` and then instantiate `M` with `List` or `Array`. As a consequence, the LINQ standard sequence operators that encode monads must rely on syntactical patterns, essentially overloads, instead of proper generics.

The ease with which you can define reusable abstractions over effects is why people often call Haskell the world’s best imperative language. Maybe the real crisis that the industry now faces is

much worse than the perceived pain of adopting pure fundamentalist functional programming. The enthusiastic reception of LINQ is an indication that people are ready for change.

ALTERNATIVES

Common belief is that the fundamentalist approach to programming is too much of a paradigm shift for ordinary programmers, and that the way forward is to make existing imperative languages more pure by taming effects. The advocated approach is to tame effects in the dual way: assume that all methods have ambient effects (are in the I/O monad), except for those that are marked with special modifiers such as `threadsafe`, `pure`, `immutable`, `readonly`, distinguishing between `val` and `var`, etc., that signal the *absence* of events. While at first sight this may seem to be less obtrusive to the developer, this is arguably not the case since once in a pure context, you cannot call an impure function and hence must transitively annotate any method called from a pure method as well—just as adding an effect somewhere deep inside a pure expression in Haskell requires a complete refactoring to monadic style.

One problem with purity annotations is that they are not extensible—that is, users cannot define new “non-effects.” As we have seen in the examples of monadic composition, it is essential to support users defining their effects and nonproper morphisms whenever necessary.

Second, purity annotations typically pertain to *functions*, whereas in Haskell, effects are not tied to functions, but to *values*. In Haskell, a function of type $f :: A \rightarrow IO\ B$ is a pure function that given a value of type *A* returns a side-effecting computation, represented by a value of type $IO\ B$. Applying the function *f*, however, does not cause any immediate effects to happen. This is rather different from marking a function as being pure. As shown here, attaching effects to values enables programmers to define their own control structures that, for example, take lists of side-effecting computations into a side-effecting computation that computes a list.

There are many other proposals for reasoning about effects in imperative languages, such as linear or uniqueness types, ownership types, or, most recently, separation logic.⁵ These all require a high degree of sophistication from both users and tools, however. This is extremely heavy mathematical machinery, particularly when compared with the simple equational reasoning of monads and pure functional programming, and thus does not really simplify the lives of ordinary programmers. One should not need a Ph.D. in theoretical computer science to be able to hack and reason about code.

Taming effects to make an imperative language pure is as painful for developers as being fundamentalist and making a pure language imperative.

CONCLUSION

The idea of “mostly functional programming” is unfeasible. It is impossible to make imperative programming languages safer by only partially removing implicit side effects. Leaving one kind of effect is often enough to simulate the very effect you just tried to remove. On the other hand, allowing effects to be “forgotten” in a pure language also causes mayhem in its own way.

Unfortunately, there is no golden middle, and we are faced with a classic dichotomy: the curse of the excluded middle, which presents the choice of either (a) trying to tame effects using purity annotations, yet fully embracing the fact that your code is still fundamentally effectful; or (b) fully embracing purity by making all effects explicit in the type system and being pragmatic by introducing nonfunctions such as `unsafePerformIO`. The examples shown here are meant to

convince language designers and developers to jump through the mirror and start looking more seriously at fundamentalist functional programming.

REFERENCES

1. Hughes, J. 1989. Why functional programming matters. *Computer Journal* 32(2): 98-107. 10.1093/comjnl/32.2.98
2. Filinski, A. 1994. Representing monads. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages* (POPL). ACM Press: 446-457.
3. Landin, P. J. 1966. The next 700 programming languages. *Communications of the ACM* 9(3): 157-166. 10.1145/365230.365257
4. Lillibridge, M. 1999. Unchecked exceptions can be strictly more powerful than call/cc. *Higher-Order and Symbolic Computation* 12(1): 75-104. 10.1023/A:1010020917337
5. O'Hearn, P. W. 2012. A primer on separation logic (and automatic program verification and analysis). *Software Safety and Security; Tools for Analysis and Verification. NATO Science for Peace and Security Series* 33: 286-318.
6. Oram, A., Wilson, G. 2007. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIK MEIJER (emeijer@applied-duality.com) is the founder of Applied Duality, Inc. and professor of Cloud Programming at TUDelft. He is perhaps best known for his contributions to programming languages such as Haskell, C#, Visual Basic, and Hack, and his work on LINQ and the Rx Framework.

© 2014 ACM 1542-7730/14/0400 \$10.00