Extracted from:

# The ThoughtWorks Anthology 2

## More Essays on Software Technology and Innovation



The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

# The ThoughtWorks® Anthology 2

## More Essays on Software Technology and Innovation

*Edited by Michael Swaine*

# The ThoughtWorks Anthology 2

## More Essays on Software Technology and Innovation


Farooq Ali
Ola Bini
Brian Blignaut
James Bull
Neal Ford
Martin Fowler
Luca Grulla
Alistair Jones
Aman King
Patrick Kua
Marc McNeill
Julio Maia
Mark Needham
Sam Newman
Rebecca Parsons
Cosmin Stejerean



The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

### Haskell

Of all the functional languages on this list, Haskell can definitely be said to take the functional paradigm the furthest. Haskell is a pure functional programming language, which means the language doesn't support mutability or side effects in any way. Of course, that's a truth with modification, since if it truly didn't support any side effects, you couldn't get it to print anything or take input from a user. Haskell does make it possible to do I/O and things that look like side effects, but in the language model, no side effects are actually occurring.

Another aspect that makes Haskell a fundamentally different language is that it's a lazy language. That means arguments to functions aren't evaluated until they are actually needed. This makes it really easy to do things such as create infinite streams, recursive function definitions, and many other useful things. Since there are no side effects, you usually won't notice that Haskell is lazy, unless you specifically utilize this aspect of the language.

Ever since ML, functional programming languages have branched into two different families of languages—the ones that use static typing and the ones that don't. Haskell is one of the more advanced statically typed functional languages, and its type system can express many things that are hard to express in other languages. However, even though the type system is very capable, it usually doesn't intrude much when actually writing a program. In most cases, you don't have to put types on functions or names; Haskell will use type inference to figure out the correct types by itself.

Haskell does not have a type system with inheritance. Instead, it uses generics to a large degree. A big part of this system is due to something called *type classes*. These classes allow you to add polymorphic behavior to existing types. You can think of type classes as interfaces with implementations that can be added to a class after it's been defined. It's a very powerful feature of Haskell, and once you start using it, you will miss it in other languages.

All in all, Haskell is a very powerful language. It is used by researchers to push the borders in many different areas, which means many new interesting libraries will first be available in Haskell. As an example, Haskell has support for many different concurrency paradigms, including Software Transactional Memory (STM) and nested data parallelism.

It is kind of weird to start a code example of Haskell with a "Hello, World" example, since the things that make it possible to create I/O in Haskell have a tendency to complicate things a bit. But no matter, let's see what it looks like.

MostInterestingLanguages/haskell/hello.hs
```
module Main where

main = do
  hello "Ola"
  hello "Stella"

hello name = putStrLn ("Hello " ++ name)
```

To run this as a stand-alone file, we have to define a main() function inside a module called Main. The do keyword allow us to do several things after each other. Finally, we define hello() to be a function that takes one argument, concatenates that argument with *"Hello "*, and then prints it.

When we compile and run this code, it looks like this:

```
$ ghc -o hello hello.hs
$ ./hello
Hello Ola
Hello Stella
```

Just as with Erlang, Haskell is really good at pattern matching. I haven't mentioned it yet, but Haskell is a whitespace-significant language, which means it uses whitespace to determine structure, just like CoffeeScript and Python. When it comes to pattern matching, this results in quite clean-looking programs. The following creates a data type for representing shapes and then uses pattern matching to calculate the area for different shapes. It also revisits our example of reversing a list by recursion and pattern matching.

MostInterestingLanguages/haskell/patterns.hs
```
module Main where

type Radius = Double
type Side   = Double

data Shape =
           Point
         | Circle    Radius
         | Rectangle Side Side
         | Square    Side
area Point           = 0
area (Circle r)      = pi * r * r
area (Rectangle w h) = w * h
area (Square s)      = s * s
rev [] = []
rev (x:xs) = rev xs ++ [x]

main = do
  print (area Point)
```

```
  print (area (Circle 10))
  print (area (Rectangle 20 343535))
  print (area (Square 20))
  print (rev [42, 55, 10, 20])
```

This gives the following output:

```
$ ghc -o patterns patterns.hs
$ ./patterns
0.0
314.1592653589793
6870700.0
400.0
[20,10,55,42]
```

As you can see, most function definitions in Haskell look a lot like algebraic statements. When defining a data type like Shape, we enumerate all the possibilities and say what data the possibilities must take. Then, when we dispatch based on the data in the call to area(), we also pick out the data contained in the data type.

I mentioned earlier that Haskell is a lazy language. That can be easily demonstrated when defining something that works with infinity, for example.

MostInterestingLanguages/haskell/lazy.hs

```
module Main where

from n = n : (from (n + 1))

main = do
  print (take 10 (from 20))
```

This code looks deceptively simple. The take() function is defined in the Haskell core library. It will take as many elements from the given list as you specify (ten in this case). Our function from() uses the colon to construct a new list. That list is defined as the value of n and then the list you get from calling from() again, with n + 1. In most languages, any time you call this function, it will recurse forever, and that's game over. But Haskell will evaluate from() only enough times to get the values it needs. This is pretty deep stuff and usually takes some time to sink in. Just remember, there is nothing special with the colon operator here. It's just the way Haskell evaluates things.

The result of running the code looks like this:

```
$ ghc -o lazy lazy.hs
$ ./lazy
[20,21,22,23,24,25,26,27,28,29]
```

The final thing I wanted to show about Haskell is something called *type classes*. Since Haskell is not object-oriented and doesn't have inheritance, it becomes really cumbersome to do things such as define a generic function that can print things, test equality, or do a range of other things. Type classes solve this problem, by basically allowing you to switch implementations based on what type Haskell thinks something is. This can be extremely powerful and very unlike anything you've seen in traditional object-oriented languages. So, let's take a look at an example.

MostInterestingLanguages/haskell/type_classes.hs

```
module Main where

type Name = String

data Platypus =
    Platypus Name
data Bird =
    Pochard Name
  | RingedTeal Name
  | WoodDuck Name

class Duck d where
  quack :: d -> IO ()
  walk  :: d -> IO ()

instance Duck Platypus where
    quack (Platypus name) = putStrLn ("QUACK from Mr Platypus " ++ name)
    walk  (Platypus _) = putStrLn "*platypus waddle*"
instance Duck Bird where
    quack (Pochard name) = putStrLn ("(quack) says " ++ name)
    quack (RingedTeal name) = putStrLn ("QUACK!! says the Ringed Teal " ++ name)
    quack (WoodDuck _) = putStrLn "silence... "
    walk _ = putStrLn "*WADDLE*"

main = do
  quack (Platypus "Arnold")
  walk (Platypus "Arnold")
  quack (Pochard "Donald")
  walk (Pochard "Donald")
  quack (WoodDuck "Pelle")
  walk (WoodDuck "Pelle")
```

We have several things going on here. First, we define two data types: one for birds and one for platypuses, which both receive names. Then we create a type class called Duck. We know that if something quacks like a duck and walks like a duck, it is a duck. So, the type class Duck defines two functions called quack() and walk(). These declarations specify only the types of the arguments and what return type is expected. These type signatures specify that

they take a ducklike thing and then print something. After that, we define an instance of the type class for our Platypus. We simply define the functions necessary inside that instance, just as we would have when defining a top-level function in Haskell. Then we do the same thing for our birds, and finally we actually call quack() and walk() on a few different data instances.

When running this example, we see that it behaves exactly as we would want.

```
$ ghc -o type_classes type_classes.hs
$ ./type_classes
QUACK from Mr Platypus Arnold
*platypus waddle*
(quack) says Donald
*WADDLE*
silence...
*WADDLE*
```

Type classes are extremely powerful, and it's hard to do them justice in a small segment like this. Rest assured that once you fully understand type classes, then you are a good way toward mastery of Haskell.

#### Resources

The best place to start learning Haskell is an online book called *Learn You a Haskell for Great Good* (http://learnyouahaskell.com). This book will take you through the paces of Haskell in an easy and entertaining way.

There are several books covering Haskell, all of them approaching from slightly different angles. Many of them are focused on using Haskell from a math or computer science perspective. If you want to learn Haskell for general-purpose programming, the best book is probably *Real World Haskell [OGS08]* by Bryan O'Sullivan, Don Stewart, and John Goerzen. It's available online at http://book.realworldhaskell.org/read.

### Io

Of all the languages in this essay, I think Io is my absolute favorite. It is a very small and powerful language. The core model is extremely regular and very simple, but it gives rise to many strange and wonderful features.

Io is a pure object-oriented programming language, where *pure* simply means that everything in Io is an object. No exceptions. Everything that you touch or work with or that the implementation uses is an object that you can reach in and get hold of. In comparison with Java, C#, Smalltalk, and many other object-oriented languages, Io does not use classes. Instead, it uses something called *prototype-based object orientation*. The idea is that you create a new

object from an existing one. You make changes directly to the object and then use that as a basis for anything else.

Traditional object-oriented languages have two different concepts: classes and objects. In most pure languages, a class is a kind of object. But there is a fundamental difference between them, namely, that classes can hold behavior while objects can't. In Io, methods are objects, just like anything else, and methods can be added to any object. This programming model makes it possible to model things very differently from the way class-based languages require you to work. An additional advantage of prototype-based languages is that they can emulate class-based languages quite well. So, if you want to work with a more class-based model, you are free to do so.

Io is a small language, but it still supports a large chunk of functionality. It has some very nice concurrency features based on coroutines. Using Io actors, it's extremely easy to build robust and scalable concurrent programs.

Another aspect of Io being pure is that the elements that are used to represent Io code are available as first-class objects. This means you can create new code at runtime, you can modify existing code, and you can introspect on existing code. This makes it possible to create extraordinarily powerful metaprogramming programs.

In Io, you define a method just like you assign any other value. You create the method and assign it to a name. The first time you assign a name, you need to use :=, but after that, you can use =. Our "Hello, World" example looks like this:

MostInterestingLanguages/io/hello.io

```
hello := method(n,
  ("Hello " .. n) println)

hello("Ola")
hello("Stella")
```

We concatenate strings using the .. operator and print something by asking it to print itself. The output is highly unsurprising.

```
$ io hello.io
Hello Ola
Hello Stella
```

Io has cooperative multitasking using both actors and futures. Any object in Io can be used as an actor by calling asyncSend() to it, with the name of the method to call. We do have to explicitly call yield to make sure all the code gets to run.

MostInterestingLanguages/io/actors.io

```
t1 := Object clone do(
  test := method(
    for(n, 1, 5,
      n print
      yield))
)

t2 := t1 clone
t1 asyncSend(test)
t2 asyncSend(test)

10 repeat(yield)
"" println

t3 := Object clone do(
  test := method(
    "called" println
    wait(1)
    "after" println
    42))
result := t3 futureSend(test)
"we want the result now" println
result println
```

The first thing this code does is to create a new object called t1 with a test() method that prints the numbers from one to five, yielding in between. We then clone that object into another object and call asyncSend(test)() on both of them, and finally we yield in the main thread ten times.

The second section creates a new object with another test() method that will first print something and then wait for one second, print something else, and then return a value. We can use this object as a transparent future by calling futureSend(test)() to it. The result of that call won't be evaluated until we actually have to use the value to print it, on the last line. This functionality is quite similar to the way Haskell handles lazy values, but we have to explicitly create the future to make this happen in Io.

When running, we get this output:

```
$ io actors.io
1122334455
we want the result now
called
after
42
```

You can see the cooperative nature of the actors in how they print their values between each other. You might also notice that the output from the method we called as a future doesn't get called until the last moment.

Another aspect of Io that is very powerful is its support for reflection and metaprogramming; basically, anything is accessible to look at or change. All code in Io is accessible at runtime, represented in the form of messages. You can do many useful things with them, including creating advanced macro facilities. This small example shows you a bit of the power of this approach, even though the specific example might not be that compelling:

MostInterestingLanguages/io/meta.io

```
add := method(n,
  n + 10)

add(40) println

getSlot("add") println
getSlot("add") message println
getSlot("add") message next println
getSlot("add") message next setName("-")

add(40) println
```

First, this code creates a method to add ten to any argument. We call it to see that it works, and then we use `getSlot()` to get access to the method object without actually evaluating it. We print it and then get the message object inside of it and print that. Messages are chained so that after evaluating one message, Io will follow the next pointer to figure out what to do next. So, we print the next pointer and then change the name of the next message. Finally, we try to add forty again. Basically, this code is changing the implementation of the `add()` method dynamically, at runtime.

And when we run it, we can see that it works.

```
$ io meta.io
50

# meta.io:2
method(n,
    n + 10
)
n +(10)
+(10)
30
```

Io is extremely malleable. Almost everything is accessible and changeable. It is a very powerful language, and it's powerful by having a very small surface

area. It blew my mind when I first learned it, and it continues to blow my mind on a regular basis.

### Resources

Io doesn't have any books written about it, but the introduction guide at http://www.iolanguage.com/scm/io/docs/IoGuide.html gives a nice grounding in the language. After you've worked through it, you should be able to look through the reference documentation and understand what's going on. Since Io is also very introspective, you can usually find out what slots an object has by just asking for it.

Steve Dekorte has several talks online about Io, and the book *Seven Languages in Seven Weeks* by Bruce Tate also has a chapter about Io.