Programming with Arrows

John Hughes

Department of Computer Science and Engineering,
Chalmers University of Technology,
S-41296 Sweden.

1 Introduction

1.1 Point-free programming

Consider this simple Haskell definition, of a function which counts the number of occurrences of a given word `w` in a string:

```
count w = length . filter (==w) . words
```

This is an example of "point-free" programming style, where we build a function by composing others, and make heavy use of higher-order functions such as `filter`. Point-free programming is rightly popular: used appropriately, it makes for concise and readable definitions, which are well suited to equational reasoning in the style of Bird and Meertens [2]. It's also a natural way to assemble programs from components, and closely related to connecting programs via pipes in the UNIX shell.

Now suppose we want to modify `count` so that it counts the number of occurrences of a word in a *file*, rather than in a string, and moreover prints the result. Following the point-free style, we might try to rewrite it as

```
count w = print . length . filter (==w) . words . readFile
```

But this is rejected by the Haskell type-checker! The problem is that `readFile` and `print` have side-effects, and thus their types involve the `IO` monad:

```
readFile :: String -> IO String
print :: Show a => a -> IO ()
```

Of course, it is one of the *advantages* of Haskell that the type-checker can distinguish expressions with side effects from those without, but in this case we pay a price. These functions simply have the wrong types to compose with the others in a point-free style.

Now, we can write a point-free definition of this function using combinators from the standard `Monad` library. It becomes:

```
count w = (>>=print) .
          liftM (length . filter (==w) . words) .
          readFile
```