Exploring Languages with Interpreters and Functional Programming Chapter 16

H. Conrad Cunningham

15 October 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

16 Haskell Function Concepts

16.1 Chapter Introduction

The previous chapter introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists.

This chapter continues to examine those concepts and their implications for Haskell programming. It explores strictness, currying, partial application, combinators, operator sections, functional composition, inline function definitions, evaluation strategies, and related methods.

The Haskell module for this chapter is in `FunctionConcepts.hs`.

16.2 Strictness

In the discussion of the fold functions, the previous chapter introduced the concept of strictness. In this section, we explore that in more depth.

Some expressions cannot be reduced to a simple value, for example, `div 1 0`. The attempted evaluation of such expressions either return an error immediately or cause the interpreter to go into an "infinite loop".

In our discussions of functions, it is often convenient to assign the symbol $\perp$ (pronounced "bottom") as the value of expressions like `div 1 0`. We use $\perp$ is a polymorphic symbol—as a value of every type.

The symbol $\perp$ is not in the Haskell syntax and the interpreter cannot actually generate the value $\perp$. It is merely a name for the value of an expression in situations where the expression cannot really be evaluated. It's use is somewhat analogous to use of symbols such as ∞ in mathematics.

Although we cannot actually produce the value $\perp$, we can, conceptually at least, apply any function to $\perp$.

If `f` $\perp = \perp$, then we say that the function is *strict*; otherwise, it is *nonstrict* (sometimes called *lenient*).

That is, a strict argument of a function must be evaluated before the final result can be computed. A nonstrict argument of a function may not need to be evaluated to compute the final result.

Assume that lazy evaluation is being used and consider the function `two` that takes an argument of any type and returns the integer value two.

```
two :: a -> Int
two x = 2
```

The function `two` is nonstrict. The argument expression is not evaluated to compute the final result. Hence, `two` $\perp = 2$.

Consider the following examples.

- The arithmetic operations (e.g. `+`) are strict in both arguments.
- Function `rev` (discussed in a previous chapter) is strict in its one argument.
- Operation `++` is strict in its first argument, but nonstrict in its second argument.
- Boolean functions `&&` and `||` from the Prelude are also strict in their first arguments and nonstrict in their second arguments.

  ```
  (&&), (||) :: Bool -> Bool -> Bool
  False && x = False   -- second argument not evaluated
  True  && x = x

  False || x = x
  True  || x = True    -- second argument not evaluated
  ```

16.3 Currying and Partial Application

Consider the following two functions:

```
add :: (Int,Int) -> Int
add (x,y) = x + y

add' :: Int -> (Int -> Int)
add' x y  = x + y
```

These functions are closely related, but they are not identical.

For all integers `x` and `y`, `add (x,y) == add' x y`. But functions `add` and `add'` have different types.

Function `add` takes a 2-tuple `(Int,Int)` and returns an `Int`. Function `add'` takes an `Int` and returns a function of type `Int -> Int`.

What is the result of the application `add 3`? An error.

What is the result of the application `add' 3`? The result is a function that "adds 3 to its argument".

What is the result of the application `(add' 3) 4`? The result is the integer value `7`.

By convention, function application (denoted by the juxtaposition of a function and its argument) binds to the left. That is, `add' x y = ((add' x) y)`.

Hence, the higher-order functions in Haskell allow us to replace any function that takes a tuple argument by an equivalent function that takes a sequence of simple arguments corresponding to the components of the tuple. This process is called *currying*. It is named after American logician Haskell B. Curry, who first exploited the technique.

Function `add'` above is similar to the function `(+)` from the Prelude (i.e. the addition operator).

We sometimes speak of the function `(+)` as being *partially applied* in the expression `((+) 3)`. In this expression, the first argument of the function is "frozen in" and the resulting function can be passed as an argument, returned as a result, or applied to another argument.

Partially applied functions are very useful in conjunction with other higher-order functions.

For example, consider the partial applications of the relational comparison operator `(<)` and multiplication operator `(*)` in the function `doublePos3`. This function, which is equivalent to the function `doublePos` discussed in an earlier section, doubles the positive integers in a list:

```
doublePos3 :: [Int] -> [Int]
doublePos3 xs = map ((*) 2) (filter ((<) 0) xs)
```

Related to the concept of currying is the *property of extensionality*. Two functions `f` and `g` are extensionally equal if `f x == g x` for all `x`.

Thus instead of writing the definition of `g` as

```
f, g :: a -> a
f x = some_expression

g x = f x
```

we can write the definition of `g` as simply:

```
g = f
```

16.4 Operator Sections

Expressions such as `((*) 2)` and `((<) 0)`, used in the definition of `doublePos3` in the previous section, can be a bit confusing because we normally use these operators in infix form. (In particular, it is difficult to remember that `((<) 0)` returns `True` for positive integers.)

Also, it would be helpful to be able to use the division operator to express a function that halves (i.e. divides by two) its operand. The function `((/) 2)` does not do it; it divides 2 by its operand.

We can use the function `flip` from the Prelude to state the halving operation. Function `flip` takes a function and two additional arguments and applies the argument function to the two arguments with their order reversed.

```
flip' :: (a -> b -> c) -> b -> a -> c  -- flip in Prelude
flip' f x y = f y x
```

Thus we can express the halving operator with the expression `(flip (/) 2)`.

Because expressions such as `((<) 0)` and `(flip (/) 2)` are quite common in programs, Haskell provides a special, more compact and less confusing, syntax.

For some infix operator ⊕ and arbitrary expression `e`, expressions of the form `(e ⊕)` and `( ⊕e)` represent `(( ⊕) e)` and `(flip ( ⊕) e)`, respectively. Expressions of this form are called *operator sections*.

Examples of operator sections include:

> `(1+)` is the successor function, which returns the value of its argument plus 1.
>
> `(0<)` is a test for a positive integer.
>
> `(/2)` is the halving function.
>
> `(1.0/)` is the reciprocal function.
>
> `(:[])` is the function that returns the singleton list containing the argument.

Suppose we want to sum the cubes of list of integers. We can express the function in the following way:

```
sumCubes :: [Int] -> Int
sumCubes xs = sum (map (^3) xs)
```

Above `^` is the exponentiation operator and `sum` is the list summation function defined in the Prelude as:

```
sum = foldl' (+) 0  -- sum
```

16.5 Combinators

The function `flip` in the previous section is an example of a useful type of function called a combinator.

A *combinator* is a function without any free variables. That is, on right side of a defining equation there are no variables or operator symbols that are not bound on the left side of the equation.

For historical reasons, `flip` is sometimes called the `C` combinator.

There are several other useful combinators in the Prelude.

The combinator `const` (shown below as `const’`) is the constant function constructor; it is a two-argument function that returns its first argument. For historical reasons, this combinator is sometimes called the `K` combinator.

```
const' :: a -> b -> a  -- const in Prelude
const' k x = k
```

Example: `(const 1)` takes any argument and returns the value 1.

Question: What does `sum (map (const 1) xs)` do?

Function `id` (shown below as `id’`) is the identity function; it is a one-argument function that returns its argument unmodified. For historical reasons, this function is sometimes called the `I` combinator.

```
id' :: a -> a  -- id in Prelude
id' x = x
```

Combinators `fst` and `snd` (shown below as `fst’` and `snd’`) extract the first and second components, respectively, of 2-tuples.

```
fst' :: (a,b) -> a  -- fst in Prelude
fst' (x,_) = x

snd' :: (a,b) -> b  -- snd in Prelude
snd' (_,y) = y
```

Similarly, `fst3`, `snd3`, and `thd3` extract the first, second, and third components, respectively, of 3-tuples.

TODO: Correct above statement. No longer seems correct. `Data.Tuple.Select` `sel1`, `sel2`, `sel2`, etc.

An interesting example that uses a combinator is the function `reverse` as defined in the Prelude (shown below as `reverse’`):

```
reverse' :: [a] -> [a]          -- reverse in Prelude
reverse' = foldlX (flip' (:)) []
```

Function `flip (:)` takes a list on the left and an element on the right. As this operation is folded through the list from the left it attaches each element as the new head of the list.

We can also define combinators that convert an uncurried function into a curried function and vice versa. The functions `curry'` and `uncurry'` defined below are similar to the Prelude functions.

```
curry' :: ((a, b) -> c) -> a -> b -> c     --Prelude curry
curry' f x y =  f (x, y)

uncurry' :: (a -> b -> c) -> ((a, b) -> c) --Prelude uncurry
uncurry' f p =  f (fst p) (snd p)
```

Two other useful combinators are `fork` and `cross` [Bird 2015]. Combinator `fork` applies each component of a pair of functions to a value to create a pair of results. Combinator `cross` applies each component of a pair of functions to the corresponding components of a pair of values to create a pair of results. We can define these as follows:

```
fork :: (a -> b, a -> c) -> a -> (b,c)
fork (f,g) x = (f x, g x)

cross :: (a -> b, c -> d) -> (a,c) -> (b,d)
cross (f,g) (x,y) = (f x, g y)
```

16.6 Functional Composition

The functional composition operator allows several "smaller" functions to be combined to form "larger" functions. In Haskell, this combinator is denoted by the period (`.`) symbol and is defined in the Prelude as follows:

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Composition's default binding is from the right and its precedence is higher than all the operators we have discussed so far except function application itself.

Functional composition is an associative binary operation with the identity function `id` as its identity element:

```
 f . (g . h) = (f . g) . h
 id . f  = f . id
```

16.7 Function Pipelines

As an example, consider the function `count` that takes two arguments, an integer `n` and a list of lists, and returns the number of the lists from the second argument that are of length `n`. Note that all functions composed below are single-argument functions: `length`, `(filter (== n))`, `(map length)`.

```
count :: Int -> [[a]] -> Int
count n  -- unprimed versions from Prelude
    | n >= 0    = length . filter (== n) . map length
    | otherwise = const 0   -- discard 2nd arg, return 0
```

We can think of the point-free expression `length . filter (== n) . map length` as defining a *function pipeline* through which data flows from right to left.

1. The pipeline takes a polymorphic list of lists as input.

2. The `map length` component of the pipeline replaces each inner list by its length.
3. The `filter (== n)` component takes the list created by the previous step and removes all elements not equal to `n`.
4. The `length` component takes the list created by the previous step and determines how many elements are remaining.
5. The pipeline outputs the value computed by the previous component. The number of lists within the input list of lists that are of length `n`.

Thus composition is a powerful form of "glue" that can be used to "stick" simpler functions together to build more powerful functions. The simpler functions in this case include partial applications of higher order functions from the library we have developed.

As we see above in the definition of `count`, partial applications (e.g. `filter (== n)`), operator sections (e.g. `(== n)`), and combinators (e.g. `const`) are useful as *plumbing* the function pipeline.

Remember the function `doublePos` that we discussed in earlier sections.

```
doublePos3 xs = map ((*) 2) (filter ((<) 0) xs)
```

Using composition, partial application, and operator sections we can restate its definition in point-free style as follows:

```
doublePos4 :: [Int] -> [Int]
doublePos4 = map (2*) . filter (0<)
```

Consider a function `last` to return the last element in a non-nil list and a function `init` to return the initial segment of a non-nil list (i.e. everything except the last element). These could quickly and concisely be written as follows:

```
last' = head . reverse             -- last in Prelude
init' = reverse . tail . reverse -- init in Prelude
```

However, since these definitions are not very efficient, the Prelude implements functions `last` and `init` in a more direct and efficient way similar to the following:

```
last2 :: [a] -> a     -- last in Prelude
last2 [x]     = x
last2 (_:xs) = last2 xs

init2 :: [a] -> [a]   -- init in Prelude
init2 [x]     = []
init2 (x:xs) = x : init2 xs
```

The definitions for Prelude functions `any` and `all` are similar to the definitions show below; they take a predicate and a list and apply the predicate to each element of the list, returning `True` when any and all, respectively, of the individual tests evaluate to `True`.

```
any', all' :: (a -> Bool) -> [a] -> Bool
any' p = or' . map' p    -- any in Prelude
all' p = and' . map' p   -- all in Prelude
```

The functions `elem` and `notElem` test for an object being an element of a list and not an element, respectively. They are defined in the Prelude similarly to the following:

```
elem', notElem' :: Eq a => a -> [a] -> Bool
elem'    = any . (==)   -- elem in Prelude
notElem' = all . (/=)   -- notElem in Prelude
```

These are a bit more difficult to understand since `any`, `all`, `==`, and `/=` are two-argument functions. Note that expression `elem x xs` would be evaluated as follows:

```
     elem’ x xs
⟹   { expand elem’ }
     (any’ . (==)) x xs
⟹   { expand composition }
     any’ ((==) x) xs
```

The composition operator binds the first argument with `(==)` to construct the first argument to `any’`. The second argument of `any’` is the second argument of `elem’`.

16.8 Lambda Expressions

Remember the function `squareAll2` we examined in an earlier section on maps:

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x
```

We introduced the local function definition `sq` to denote the function to be passed to `map`. It seems to be a waste of effort to introduce a new symbol for a simple function that is only used in one place in an expression. Would it not be better, somehow, to just give the defining expression itself in the argument position?

Haskell provides a mechanism to do just that, an anonymous function definition. For historical reasons, these nameless functions are called *lambda expressions*. They begin with a backslash `\{.haskell}` and have the syntax:

\ *atomicPatterns* -> *expression*

For example, the squaring function (`sq`) could be replaced by a lambda expression as `(\x -> x * x)`. The pattern `x` represents the single argument for this anonymous function and the expression `x * x` is its result.

Thus we can rewrite `squareAll2` in point-free style using a lambda expression as follows:

```
squareAll3 :: [Int] -> [Int]
squareAll3 = map' (\x -> x * x)
```

A lambda expression to average two numbers can be written `(\x y -> (x+y)/2)`.

An interesting example that uses a lambda expression is the function `length` as defined in the Prelude—similar to `length4` below. (Note that this uses the optimized function `foldl'` from the standard Haskell `Data.List` module.)

```
length4 :: [a] -> Int    -- length in Prelude
length4  = foldl' (\n _ -> n+1) 0
```

The anonymous function `(\n _ -> n+1)` takes an integer "counter" and a polymorphic value and returns the "counter" incremented by one. As this function is folded through the list from the left, this function counts each element of the second argument.

16.9 Application Operator $

In Haskell, function application associates to the left and has higher binding power than any infix operator. For example, for some function two-argument function `f` and values `w`, `x`, `y`, and `z`

```
w + f x y * z
```

is the same as

```
w + (((f x) y) * z)
```

given the relative binding powers of function application and the numeric operators.

However, sometimes we want to be able to use function application where it associates to the right and binds less tightly than any other operator. Haskell defines the `$` operator to enable this style, as follows:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Thus, for single argument functions `f`, `g`, and `h`,

```
f $ g $ h $ z + 7
```

is the same as

```
(f (g (h (z+7))))
```

and as:

```
(f . g . h) (z+7)
```

Similarly, for two-argument functions `f'`, `g'`, and `h'`,

```
f' w $ g' x $ h' y $ z + 7
```

is the same as

```
((f' w) ((g' x) ((h' y) (z+7))))
```

and as:

```
(f' w . g' x . h' y) (z+7)
```

For example, this operator allows us to write

```
foldr (+) 0 $ map (2*) $ filter odd $ enumFromTo 1 20
```

where Prelude function `enumFromTo m n` generates the sequence of integers from `m` to `n`, inclusive.

16.10 Eager Evaluation Using `seq` and `$!`

Haskell is a lazily evaluated language. That is, if an argument is nonstrict it may never be evaluated.

Sometimes, using the technique called *strictness analysis*, the Haskell compiler can detect that an argument's value will always be needed. The compiler can then safely force eager evaluation as an optimization without changing the meaning of the program.

In particular, by selecting the `-O` option to the Glasgow Haskell Compiler (GHC), we can enable GHC's code optimization processing. GHC will generally create smaller, faster object code at the expense of increased compilation time by taking advantage of strictness analysis and other optimizations.

However, sometimes we may want to force eager evaluation explicitly without invoking a full optimization on all the code (e.g. to make a particular function's evaluation more space efficient). Haskell provides the primitive function `seq` that enables this. That is,

```
seq :: a -> b -> b
x `seq` y = y
```

where it just returns the second argument except that, as a side effect, `x` is evaluated before `y` is returned. (Technically, `x` is evaluated to what is called *head normal form*. It is evaluated until the outer layer of structure such as `h:t` is revealed, but `h` and `t` themselves are not fully evaluated. We examine evaluation in more detail in a later chapter.)

Function foldl, the "optimized" version of foldl can be defined using seq as follows

```
foldlP :: (a -> b -> a) -> a -> [b] -> a  -- Data.List.foldl'
foldlP f z []     = z
foldlP f z (x:xs) = y `seq` foldl' f y xs
                    where y = f z x
```

That is, this evaluates the z argument of the tail recursive application eagerly.

Using seq, Haskell also defines $!, a strict version of the $ operator, as follows:

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

The effect of `f $! x` is the same as `f $ x` except that $! eagerly evaluates the argument x before applying function f to it.

We can rewrite foldl' using $! as follows:

```
foldlQ :: (a -> b -> a) -> a -> [b] -> a  -- Data.List.foldl'
foldlQ f z []     = z
foldlQ f z (x:xs) = (foldlQ f $! f z x) xs
```

We can write a tail recursive function to sum the elements of the list as follows:

```
sum4 :: [Integer] -> Integer  -- sum in Prelude
sum4 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs (acc+x)
```

We can then redefine sum4 to force eager evaluation of the accumulating parameter of sumIter as follows:

```
sum5 :: [Integer] -> Integer -- sum in Prelude
sum5 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs $! acc + x
```

However, we need to be careful in applying seq and $!. They change the semantics of the lazily evaluated language in the case where the argument is nonstrict. They may force a program to terminate abnormally and/or cause it to take unnecessary evaluation steps.

16.11 What Next?

The previous chapter introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists. This chapter continued to examine those concepts and their implications for Haskell programming by exploring concepts and features such as strictness, currying, partial application, combinators, operator sections, functional composition, inline function definitions, and evaluation strategies.

The next chapter looks at additional examples that use these higher-order programming concepts.

16.12 Exercises

1. Define a Haskell function

   ```
   total :: (Integer -> Integer) -> Integer -> Integer
   ```

   so that `total f n` gives `f 0 + f 1 + f 2 + ... + f n`. How could you define it using `removeFirst`?

2. Define a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

3. Define a Haskell function `fmap` that takes a value and a list of functions and returns the list of results from applying each function to the argument value. (For example, `fmap 3 [((*) 2), ((+) 2)]` yields `[6,5]`.)

4. Define a Haskell function `composeList` that takes a list of functions and composes them into a single function. (Be sure to give the type signature.)

16.13 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- chapter 6 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]
- my notes on *Functional Data Structures (Scala)* [Cunningham 2016] which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [Chiusano 2015]

In Summer 2016, I also added the following,a drawing on ideas from [Bird 2015, Ch. 6, 7] and [Thompson 2011, Ch. 11]:

- expanded discussion of combinators and functional composition
- new discussion of the `seq`, `$`, and `$!` operators

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 5.1-5.2 became the basis for new Chapter 15, Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts (this chapter), and previous sections 5.4-5.6 became the basis for new Chapter 17, Higher-Order Function Examples.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

16.14 References

[**Bird 1988**]**:** Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[**Bird 1998**]**:** Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[**Bird 2015**]**:** Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

[**Chiusano 2015**]**:** Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Cunningham 2016**]**:** H. Conrad Cunningham, *Functional Data Structures (Scala)*, 2016. (Lecture notes based, in part, on chapter 3 [Chiusano 2015].)

[**Thompson 2011**]**:** Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

16.15 Terms and Concepts

Strict and nonstrict functions, bottom, strictness analysis, currying, partial application, operator sections, combinators, functional composition, property of extensionality, pointful and point-free styles, plumbing, function pipeline, lambda expression, application operator `$`, eager evaluation operators `seq` and `$!`, head-normal form.