Exploring Languages with Interpreters and Functional Programming
Chapter 4

H. Conrad Cunningham

1 September 2018

Contents

Copyright (C) 2016, 2017, 2018, H. Conrad Cunningham
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 September 2018 is a recent version of Firefox from Mozilla.

4 First Haskell Programs

4.1 Chapter Introduction

The goals of this chapter are to

- introduce the definition of Haskell functions using examples
- illustrate the use of the `ghci` interactive REPL (Read-Evaluate-Print Loop) interpreter

4.2 Defining Our First Haskell Functions

Let's look at our first function definition in the Haskell language, a program to implement the factorial function for natural numbers.

The Haskell source file `Factorial.hs` holds the Haskell function definitions for this chapter. The test script is in source file `TestFactorial.hs`; it is discussed further in Chapter 10 on Software Testing,

4.2.1 Factorial function specification

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments. We can define *fact* using the product operator as follows:

$$fact(n) = \prod_{i=1}^{i=n} i$$

For example,

$$fact(4) = 1 \times 2 \times 3 \times 4.$$

By definition

$$fact(0) = 1$$

which is the *identity* element of the multiplication operation.

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

$$fact'(n) = 1,\ if\ n = 0$$
$$fact'(n) = n \times\ fact'(n-1),\ if\ n \geq 1$$

Since the domain of *fact'* is the set of natural numbers, a set over which induction is defined, we can easily see that this recursive definition is well defined.

- For $n = 0$, the base case, the value is simply 1.

- For $n \geq 1$, the value of *fact'*(n) is recursively defined in terms of *fact'*($n-1$). The argument of the recursive application decreases toward the base case.

In the Review of Relevant Mathematics appendix, we prove that *fact*(n) = *fact'*(n) by mathematical induction.

The Haskell functions defined in the following subsections must compute *fact*(n) when applied to argument value $n \geq 0$.

4.2.2 Factorial function using if-then-else: `fact1`

One way to translate the recursive definition *fact'* into Haskell is the following:

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
              n * fact1 (n-1)
```

- The first line above is the *type signature* for function `fact1`. In general, type signatures have the syntax *object* `::` *type*.

  Haskell type names begin with an uppercase letter.

  The above defines object `fact1` as a function (denoted by the `->` symbol) that takes one argument of type integer (denoted by the first `Int`) and returns a value of type integer (denoted by the last `Int`).

  Haskell does not have a built-in natural number type. Thus we choose type `Int` for the argument and result of `fact1`.

  The `Int` data type is a bounded integer type, usually the integer data type supported directly by the host processor (e.g. 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer (-2^{29} to 2^{29}).

- The declaration for the function `fact1` begins on the second line. Note that it is an equation of the form

  fname parms `=` *body*

  where *fname* is the function's name, *parms* are the function's parameters, and *body* is an expression defining the function's result.

  Function and variable names begin with lowercase letters optionally followed by a sequence of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

  A function may have zero or more parameters. The parameters are listed after the function name without being enclosed in parentheses and without commas separating them.

The parameter names may appear in the *body* of the function. In the evaluation of a function *application* the actual argument values are substituted for parameters in the *body*.

- Above we define the *body* function `fact1` to be an `if-then-else` *expression*. This kind of expression has the form

  `if` *condition* `then` *expression1* `else` *expression2*

  where

  condition is a Boolean expression, that is, an expression of Haskell type `Bool`, which has either `True` or `False` as its value

  expression1 is the expression that is returned when the condition is `True`

  expression2 is the expression (with the same type as *expression1*) that is returned when the condition is `False`

  Evaluation of the `if-then-else` expression in `fact1` yields the value `1` if argument `n` has the value `0` (i.e. `n == 0`) and yields the value `n * fact1 (n-1)` otherwise.

- The `else` clause includes a recursive application of `fact1`. The whole expression `(n-1)` is the argument for the recursive application, so we enclose it in parenthesis.

  The value of the argument for the recursive application is less than the value of the original argument. For each recursive application of `fact` to a natural number, the argument's value thus moves closer to the termination value `0`.

- Unlike most conventional languages, the *indentation is significant* in Haskell. The indentation indicates the nesting of expressions.

  For example, in `fact1` the `n * fact1 (n-1)` expression is nested inside the `else` clause of the `if-then-else` expression.

- This Haskell function does not match the mathematical definition given above. What is the difference?

  Notice the domains of the functions. The evaluation of `fact1` will go into an "infinite loop" and eventually abort when it is applied to a negative value.

In Haskell there is *only* one way to form more complex expressions from simpler ones: *apply* a function.

Neither parentheses nor special operator symbols are used to denote function application; it is denoted by simply listing the argument expressions following the function name. For example, a function `f` applied to argument expressions `x` and `y` is written in the following *prefix* form:

```
f x y
```

However, the usual prefix form for a function application is not a convenient or natural way to write many common expressions. Haskell provides a helpful bit of syntactic sugar, the *infix* expression. Thus instead of having to write the addition of `x` and `y` as

```
add x y
```

we can write it as

```
x + y
```

as we have since elementary school. Here the symbol `+` represents the addition function.

Function application (i.e. juxtaposition of function names and argument expressions) has higher precedence than other operators. Thus the expression `f x + y` is the same as `(f x) + y`.

4.2.3 Factorial function using guards: `fact2`

An alternative way to differentiate the two cases in the recursive definition is to use a different equation for each case. If the Boolean *guard* (e.g. `n == 0`) for an equation evaluates to true, then that equation is used in the evaluation of the function. A guard is written following the `|` symbol as follows:

```
fact2 :: Int -> Int
fact2 n
    | n == 0    = 1
    | otherwise = n * fact2 (n-1)
```

Function `fact2` is equivalent to the `fact1`. Haskell evaluates the guards in a top-to-bottom order. The `otherwise` guard always succeeds; thus it's use above is similar to the trailing `else` clause on the `if-then-else` expression used in `fact1`.

4.2.4 Factorial function using pattern matching: `fact3` and `fact4`

Another equivalent way to differentiate the two cases in the recursive definition is to use *pattern matching* as follows:

```
fact3 :: Int -> Int
fact3 0 = 1
fact3 n = n * fact3 (n-1)
```

The parameter pattern `0` in the first *leg* of the definition only matches arguments with value 0. Since Haskell checks patterns and guards in a top-to-bottom order, the `n` pattern matches all nonzero values. Thus `fact1`, `fact2`, and `fact3` are equivalent.

To stop evaluation from going into an "infinite loop" for negative arguments, we can remove the negative integers from the function's domain. One way to do this is by using guards to narrow the domain to the natural numbers as in the definition of `fact4` below:

```
fact4 :: Int -> Int
fact4 n
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

Function `fact4` is undefined for negative arguments. If `fact4` is applied to a negative argument, the evaluation of the program encounters an error quickly and returns without going into an infinite loop. It prints an error and halts further evaluation.

We can define our own error message for the negative case using an `error` call as in `fact4'` below.

```
fact4' :: Int -> Int
fact4' n
    | n == 0    =  1
    | n >= 1    =  n * fact4' (n-1)
    | otherwise = error "fact4' called with negative argument"
```

In addition to displaying the custom error message, this also displays a stack trace of the active function calls.

4.2.5 Factorial function using built-in library function: `fact5`

The four definitions we have looked at so far use recursive patterns similar to the recurrence relation *fact'*. Another alternative is to use the library function `product` and the list-generating expression `[1..n]` to define a solution that is like the function *fact*:

```
fact5 :: Int -> Int
fact5 n = product [1..n]
```

The list expression `[1..n]` generates a *list* of consecutive integers beginning with `1` and ending with `n`. We study lists in a later chapter.

The library function `product` computes the product of the elements of a finite list.

If we apply `fact5` to a negative argument, the expression `[1..n]` generates an empty list. Applying `product` to this empty list yields 1, which is the identity element for multiplication. Defining `fact5` to return 1 is consistent with the function *fact* upon which it is based.

Which of the above definitions for the factorial function is better?

Most people in the functional programming community would consider `fact4` (or `fact4'`) and `fact5` as being better than the others. The choice between them depends upon whether we want to trap the application to negative numbers as an error or to return the value 1.

4.2.6 Testing

Chapter 10 discusses testing of the Factorial module designed in this chapter. The test script is `TestFactorial.hs`.

4.3 Using the Glasgow Haskell Compiler (GHC)

See the Glasgow Haskell Compiler Users Guide for information on the Glasgow Haskell Compiler (GHC) and its use.

GHCi is an environment for using GHC interactively. That is, it is a REPL (Read-Evaluate-Print-Loop) command line interface using Haskell. The "Using GHCi" chapter of the User Guide describes its usage.

Below, we show a GHCi session where we load source code file (module) `Factorial.hs` and apply the factorial functions to various inputs. The instructor ran this in a Terminal session on an iMac running macOS 10.13.4 (High Sierra) with `ghc` 8.4.3 installed.

1. Start the REPL.

   ```
   bash-3.2$ ghci
   GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
   ```

2. Load module `Fact` that holds the factorial function definitions. This assumes the `Factorial.hs` file is in the current directory. The `load` command can be abbreviated as just `:l`.

   ```
   Prelude> :load Factorial
   [1 of 1] Compiling Factorial        ( Factorial.hs, interpreted )
   Ok, one module loaded.
   ```

3. Inquire about the type of `fact1`.

   ```
   *Factorial> :type fact1
   fact1 :: Int -> Int
   ```

4. Apply function `fact1` to 7, 0, 20, and 21. Note that the factorial of 21 exceeds the `Int` range.

   ```
   *Factorial> fact1 7
   5040
   ```

```
*Factorial> fact1 0
1
*Factorial> fact1 20
2432902008176640000
*Factorial> fact1 21
-4249290049419214848
```

5. Apply functions `fact2`, `fact3`, `fact4`, and `fact5` to 7.

```
*Factorial> fact2 7
5040
*Factorial> fact3 7
5040
*Factorial> fact4 7
5040
*Factorial> fact5 7
5040
```

6. Apply functions `fact1`, `fact2`, and `fact3` to -1. All go into an infinite recursion, eventually terminating with an error when the runtime stack overflows its allocated space.

```
*Factorial> fact1 (-1)
*** Exception: stack overflow
*Factorial> fact2 (-1)
*Factorial> fact3 (-1)
*** Exception: stack overflow
```

7. Apply functions `fact4` and `fact4'` to -1. They quickly return with an error.

```
*Factorial> fact4 (-1)
*** Exception: Factorial.hs:(54,1)-(56,29):
  Non-exhaustive patterns in function fact4
*Factorial> fact4' (-1)
*** Exception: fact4' called with negative argument
CallStack (from HasCallStack):
  error, called at Factorial.hs:64:17 in main:Factorial
```

8. Apply function `fact5` to -1. It returns a 1 because it is defined for negative integers.

```
*Factorial> fact5 (-1)
1
```

9. Set the `+s` option to get information about the time and space required and the `+t` option to get the type of the returned value.

```
*Factorial> :set +s
*Factorial> fact1 20
2432902008176640000
```

```
(0.00 secs, 80,712 bytes)
*Factorial> :set +t
*Factorial> fact1 20
2432902008176640000
it :: Int
(0.05 secs, 80,792 bytes)
*Factorial> :unset +s +t
*Factorial> fact1 20
2432902008176640000
```

10. Exit GHCi.

```
:quit
Leaving GHCi.
```

Suppose we had set the environment variable `EDITOR` to our favorite text editor in the Terminal window. For example, on a Mac OS system, your instructor might give the following command in shell (or in a startup script such as `.bash_profile`):

```
export EDITOR=Aquamacs
```

Then the `:edit` command within GHCi allows us to edit the source code. We can give a filename or default to the last file loaded.

```
:edit
```

Or we could also use a `:set` command to set the editor within GHCi.

```
:set editor Aquamacs
...
:edit
```

See the Glasgow Haskell Compiler Users Guide for more information about use of GHC and GHCi.

4.4 What Next?

In this chapter, we looked at our first Haskell functions and how to execute them using the Haskell interpreter.

In the next chapter, we continue our exploration of Haskell by examining its built-in types.

4.5 Exercises

1. Reimplement functions `fact4` and `fact5` with type `Integer` instead of `Int`. Using `ghci`, execute these functions for values -1, 7, 20, 21, and 50 using `ghci`.

2. Develop both recursive and iterative (looping) versions of a factorial fuunction in an imperative language (e.g. Java, C++, Python 3, etc.)

4.6 Acknowledgements

In Summer 2016, I adapted and revised much of this work in from Chapter 3 of my *Notes on Functional Programming with Haskell* [Cunningham 2014] and incorporated it into Chapter 2, Basic Haskell Functional Programming, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs (this chapter); previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction.

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.

4.7 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.

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

4.8 Terms and Concepts

Factorials, function definition and application, recursion, function domains, `error`, `if`, guards, basic types (`Int`, `Integer`, `Bool`, `Char`, functions, tuples, lists, `String`), `type` aliases, library (Prelude) functions, REPL, `ghci` commands and use.