To do

1.8
You're halfway through

Construction

Reduction, Functions and Lists

14 comments

Reduction

A model of program execution

- A programmer needs a concrete model for how a program is executed.
- For imperative programs, we can execute statement by statement, keeping track of the values of variables (the stack) and where we are in the program (the program counter).
- Functional programs don't have statements!
- The mechanism for executing functional programs is *reduction*.

Reduction

Reduction is the process of converting an expression to a simpler form. Conceptually, an expression is reduced by simplifying one *reducible expression* (called "redex") at a time. Each step is called a reduction, and we'll use `-- >` to show the result.

```
  3 + (4*5)
--  >
  3 + 20
--  >
23
```

Reduction is important because it is the sole means of execution of a functional program. There are no statements, as in imperative languages; all computation is achieved purely by reducing expressions.

Unique reduction path

When a reduction is performed, there is only one possible answer. In this example, the computation has only one possible path:

```
  3 + (5 * (8-2))
--  >
  3 + (5 * 6)
--  >
  3 + 30
--  >
33
```

There is only one possible reduction path in that example, because in each step the current expression contains only one redex.

Multiple reduction paths

If an expression contains several redexes, there will be several reduction paths.

```
(3+4) * (15-9)
-- >
  7 * (15-9)
-- >
  7 * 6
-- >
 42

(3+4) * (15-9)
-- >
  (3+4) * 6
-- >
  7 * 6
-- >
  42
```

The result doesn't depend on reduction path!

A fundamental theorem (the Church-Rosser theorem):

Every terminating reduction path gives the same result

This means that

- Correctness doesn't depend on order of evaluation.
- The compiler (or programmer) can change the order freely to improve performance, without affecting the result.
- Different expressions can be evaluated in parallel, without affecting the result. *As a result, functional languages are leading contenders for programming future parallel systems*.

Functions

Haskell is a functional language so the function concept is essential to the language. A function takes one or more arguments and computes a result. Given the same arguments, the result will always be the same. This is similar to a mathematical function and it means that in Haskell there are no side-effects. There are two fundamental operations on functions: function definition (creating a function) and function application (using a function to compute a result).

Function definitions

- In Haskell, many functions are pre-defined in a standard library called the *prelude*.
- In due course, we'll learn how to use many of these standard functions.

Defining a function

- But the essence of functional programming is defining your own functions to solve your problems!
- A function is defined by an *equation*.

```
f = \x -> x+1  -- lambda function
-- or
f x = x+1 -- named function
```

This is equivalent to $f(x) = x + 1$ in mathematical notation.

- The left hand side of the equation looks like a variable – and that's what it is
- The right hand side is an expression that uses the local variables listed in parentheses and defines the result of the expression.

Function application

How function application works

- A function definition is an equation, e.g. `f = \x → x + 1`
- The left hand side gives the name of the function;
- The right hand side (the "body") is an expression giving the formal parameters, and the value of the application. The expression may use the parameters.
- An application is an expression like `f 31`, where 31 is the argument.

- The application is evaluated by replacing it with the body of the function, where the formal parameters are replaced by the arguments.

Example of application

```
f  = \x - > x+1
  f 3
--  > {bind x=3}
  (x+1) where x=3
--  > {substitute 3 for x}
  3+1
--  >
4
```

Multiple arguments and results

Functions with several arguments

A function with three arguments:

```
add3nums = \x y z -> x + y + z
```

To use it,

```
10 + 4* add3nums 1 2 3
= {- put extra parentheses in to show structure -}
  10 + ( 4* (add3nums 1 2 3) )
  -- >
  10 + (4*(1+2+3) )
  -- >
  10 + (4*6)
  -- >
  10 + 24
  -- >
  34
```

Lists

A key datastructure: the list

- A list is a single value that contains several other values.
- Syntax: the elements are written in square parentheses, separated by commas.

```
['3', 'a']
[2.718, 50.0, -1.0]
```

Function returning several results

- Actually, a function can return only one result.
- However, lists allow you to package up several values into one object, which can be returned by a function.
- Here is a function (minmax) that returns both the smaller and the larger of two numbers:

```
minmax = \x y -> [min x y, max x y]
minmax 3 8  -- > [3,8]
minmax 8 3  -- > [3,8]
```

The elements are evaluated lazily

You can write a constant list

```
mylist = [2,4,6,8]
```

But the elements can be expressions. They are evaluated only when they are used. Suppose you define:

```
answer = 42
yourlist = [7, answer+1, 7*8]
```

Then

```
yourlist -- > [7, 43, 56]
```

But as long as you do not access the expression, it is not evaluated.

Constructing lists

Append: the (++) operator

- The (++) operator takes two existing lists, and gives you a new one containing all the elements.
- The operator is pronounced *append*, and written as two consecutive + characters.

```
[23, 29] ++ [48, 41, 44] -- > [23, 29, 48, 41, 44]
```

A couple of useful properties:

- The length of the result is always the sum of the lengths of the original lists.
- If xs is a list, then $[] + +xs = xs = xs + +[]$.

Sequences

- Sometimes it's useful to have a sequence of numbers.
- In standard mathematical notation, you can write $0, 1, \ldots, n$.
- Haskell has a sequence notation for lists.
- Write the sequence in square brackets, with start value, the operator `..`, and end value.
- `[0 .. 5] -- > [0,1,2,3,4,5]`
- `[100 .. 103] -- > [100,101,102,103]`
- The elements are incremented by 1

Sequences aren't limited to numbers

- There are many *enumerable types* where there is a natural way to increment a value.
- You can use sequences on any such type.
- Characters are enumerable: there is a successor to each character.

For example:

```
['a' .. 'z']
-- >
['a','b','c','d','e','f','g','h','i','j','k', 'l','m','n','o','p','q','r','s','t','u','v', 'w','x','y','z']
```

is a list of characters;

```
['0' .. '9']
-- >
['0','1','2','3','4','5','6','7','8','9']
```

is a list of characters (which happen to be the digit characters);

```
[0 .. 9]
-- >
[0,1,2,3,4,5,6,7,8,9]
```

is a list of numbers.

List comprehensions

- A list comprehension is a high level notation for specifying the computation of a list
- The compiler automatically transforms a list comprehensions into an expression using a family of basic functions that operate on lists

- List comprehensions were inspired by the mathematical notation *set comprehension*.

- Examples of set comprehensions:
  - A set obtained by multiplying the elements of another set by 3 is $\{3 \times x \mid x \leftarrow \{1, \ldots, 10\}\}$.
  - The set of even numbers is $\{2 \times x \mid x \leftarrow N\}$.
  - The set of odd numbers is $\{2 \times x + 1 \mid x \leftarrow N\}$.
  - The cross product of two sets A and B is $\{(a, b) \mid a \leftarrow A, b \leftarrow B\}$.

Examples of list comprehensions

```
[3*x | x <- [1..10]]
-- >
[3,6,9,12,15,18,21,24,27,30]

[2*x | x <- [0..10]]
-- >
[0,2,4,6,8,10,12,14,16,18,20]

[2*x + 1 | x <- [0..10]]
-- >
[1,3,5,7,9,11,13,15,17,19,21]

[[a,b] | a <- [10,11,12] , b <- [20,21]]
-- >
[[10,20],[10,21],[11,20],[11,21],[12,20],[12,21]]
```

Operating on lists

Indexing a list

- We can index a list by numbering the elements, starting with 0.
- Thus a canonical form of a list with n elements is $[x_0, x_1, ..x_{n-1}]$.
- The !! operator takes a list and an index, and returns the corresponding element.

```
[5,3,8,7]  !! 2    -- > 8
[0 .. 100]  !! 81  -- > 81
['a'..'z'] !! 13  -- > 'n'
```

- If the index is negative, or too large, *undefined* is returned.
- For robust programming, we need to ensure either that all expressions are well defined, or else that all exceptions are caught and handled.
- Later, we'll look at how to follow both of those approaches.

head and *tail*

- There are standard library functions to give the head of a list (its first element) or the tail (all the rest of the list)
- The result of applying *head* or *tail* to the empty list is *undefined*.

```
head :: [a] -> a
head [4,5,6] -- > 4
tail :: [a] -> [a]
tail [4,5,6] -- > [5,6]
```

- Recommendation: avoid using (head) and (tail), because you want to avoid undefined values so your programs are robust. Unless you're doing something really sophisticated, you're better off with pattern matching. There are, however, some cases where they are appropriate.

Lists are lazy

We have mentioned before that Haskell is "lazy", meaning that it only evaluates expressions when they are required for the evaluation of another expression.This behaviour extends to lists, so we can actually define infinite lists using sequences, for example `[1 .. ]` is the list of all positive integers. Another example is the `primes` function (from the Data.Numbers.Primes package) which returns an infinite list of prime numbers. A consequence of laziness in lists is that you can define lists containing very complex and time consuming expressions, and as long as you never access them they will not be evaluated. The same is true for an incorrect expression, for example defining `xs = [1,2,xs !! 5,4]` will not result in an error as long as you don't access the third element.

Keep in mind that lists are also immutable. As a result, if you define `xs2 = xs ++ xs` and try to access the third element `xs2 !! 2` will still result in an error because `xs` has not been modified:

```
xs2 !! 2 -- > *** Exception: Prelude.(!!): index too large
```

Interestingly, if we change the definition of `xs` to `xs = [1,2,xs2 !! 5,4]`, then both `xs !! 2` and `xs2 !! 2` will return `2`:

```
xs = [1,2,xs2 !! 5,4]
xs2 = xs ++ xs
xs2 !! 2 -- > 2
xs !! 2 -- > 2
```

This is a consequence of Haskell's expression evaluation through reduction: the order of the expressions does not matter.