Functional Programming

Parallel Prefix Circuits

Stefan Cross

FPR Coursework Submission, Hilary Term 2014

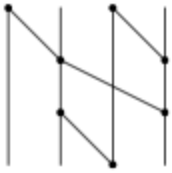
Introduction

Functional programming is an approach to structuring programs that "treats computation as the evaluation of mathematical functions that avoid state and mutable data… emphasising functions that produce results, depending on only their inputs and not the program state… thus eliminating side-effects"[1]. Functional programming contrasts to imperative programming in that we "don't tell the computer what to do - we tell it what stuff is"[2]. The functional programming paradigm is becoming increasingly prevalent, with concepts now commonplace in mainstream languages such as Microsoft's F#, Java's with Scala and Ruby. The application of functional mathematical concepts into programming languages offers improved handling of multi-core processing due to immutability and the concept of referential transparency, amongst many other benefits.

This paper covers the modeling of Parallel Prefix Circuits (PPC's) in the Haskell Programming Language. We will start with a brief overview of PPC's and the problem domain, before progressing through a dozen points that analyse the circuit representation in Haskell and perform various computations on the circuits. Finally we will summarise the findings and outcomes of the assignment and consider the successes and shortcomings of the various implementations. All code can be found at the following Github repository: https://github.com/stefan-cross/ppc

Parallel Prefix Circuits Overview

Prefix computation depends on an associative binary operator $\oplus$. A prefix computation of size n takes as input a list $[x_1,...,x_n]$ of length n and returns as output the list $[x_1, x_1 x_2, ..., x_1 x_2 \cdot \cdot \cdot x_n]$ the 'sums' (with respect to binary operator) of the non-empty prefixes of the input list. Prefix computations are a core component in many parallel algorithms, especially recently those implemented to run on graphical processing units. At first sight, the computation appears necessarily to take linear time, even if computed in parallel, because of the linear data dependency. But because of the assumption of associativity, it is possible to perform the computation in logarithmic time on linearly many processors. Moreover, the computation follows a fixed sequence, oblivious to the actual data, so it is amenable to implementation in hardware.

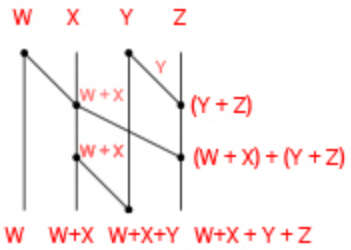


We are given the Brent-Kung circuit, size 4, which is represented by the following term of type *Circuit*:

```
(Fan 2 `Beside` Fan 2) `Above` Stretch [ 2, 2 ] (Fan 2) `Above`
(Id 1 `Beside` Fan 2 `Beside` Id 1)
```

[1] https://en.wikipedia.org/wiki/Functional_programming
[2] Learn You a Haskell for Great Good, (Introduction), Miran Lipovaca 2011

This circuit diagram should be read as follows. The inputs are fed in at the top, and the outputs fall out at the bottom. Each node represents a local computation, combining the values on each of its input wires using and providing copies of the result on each of its output wires.



Circuits can be constructed from a couple of primitives using just three combinators. The two primitives are the *identity* and *fan* circuits of a given size; the first simply copies input to output, whereas the second adds its first input to each of the others.

```
(Fan 2 `Beside` Id 1)  (Id 1 `Beside` Fan 2)
```

Circuits can be placed *Above* or *Beside* each other.

```
(Fan 2 `Beside` Id 1)`Above`(Id 1 `Beside` Fan 2)
```

Another less obvious combinator is Stretch. It is represented as follows:

```
(Stretch [3, 2, 3] Fan 3)
```

*Where the list is interpreted as starting from the 3rd line, then stretching 2, and 3 lines accordingly. The Fan primitive then confirms the number of Fan points. (**It is important to note the assumption of the first number in the list represents the starting point as this assumption is used through out the paper.)***

1. Defining Circuit Width and Depth

The following functions width and depth compute the width and depth of a circuit, assuming that circuits are adequately defined in that if one circuit is above another they both have the same width.

```
> import Data.Char
> type Size = Int -- natural numbers
> data Circuit
>     = Id Size
>     | Fan Size
>     | Beside Circuit Circuit
>     | Stretch [Size] Circuit
>     | Above Circuit Circuit
>     deriving(Show, Eq, Ord)

> width, depth :: Circuit -> Size
> width cir = case cir of
>    (Id i) -> count i
>    (Fan i) -> count i
>    (Beside i j) -> (count (width i)) + (count (width j))
>    (Stretch xs cir) -> (sum xs)
>    (Above i _) -> count (width i)
>    where
>        count i = i

> depth cir = case cir of
>    (Above i j) -> (count' (depth i)) + (count' (depth j))
>    _ -> 1
>    where
>        count' i = i
```

The functional correctness can be confirmed with the following examples, that progress from basic circuits to the more elaborate:

```
*Main> width((Id 1) `Beside` (Id 1))
2
*Main> width((Id 1) `Beside` (Fan 2))
3
*Main> width((Fan 2 `Beside` Fan 2) `Above` Stretch [ 2, 2 ] (Fan 2) `Above`
(Id 1 `Beside` Fan 2 `Beside` Id 1))
4
*Main> depth(Fan 2)
1
*Main> depth((Fan 2 `Beside` Fan 2) `Above` Stretch [ 2, 2 ] (Fan 2) `Above`
(Id 1 `Beside` Fan 2 `Beside` Id 1))
```

3

2. Defining a function for ‘well-sized’

A function to determine if a circuit is ‘well-sized’ that adheres to the size constraints on the *Above* and *Stretch* combinators. The term ‘well-sized’ could be seen as somewhat ambiguous so for clarification, it is taken to mean that the stretch list values match the corresponding *Fan* size and that the width of ‘above’ and ‘below’ circuits match.

```
> wellsized :: Circuit -> Bool
> wellsized cir = case cir of
>    (Id i) -> True
>    (Fan f) -> True
>    (Beside a b) -> (compare' (returnSize a) (returnSize b))
>    (Stretch xs cir) -> (length xs == (returnSize cir))
>    (Above a b) ->  (compare' (returnSize a) (returnSize b))
>    where
>        compare' a b = a == b
>        returnSize cir = case cir of
>            (Id i) -> i
>            (Fan f) -> f
>            (Beside a b) -> (returnSize a) + (returnSize b)
>            (Stretch xs cir) -> (length xs) + (returnSize cir)
>            (Above a _) -> (returnSize a)
```

The functional correctness can be confirmed with the following examples, that progress from basic circuits to the more elaborate:

```
*Main> wellsized (Stretch [3, 2, 3 ] (Fan 3))
True
*Main> wellsized (Stretch [ 2, 2 ] (Fan 3))
False
*Main> wellsized(Id 1 `Above` Id 1)
True
*Main> wellsized(Id 1 `Above` Id 2)
False
*Main> wellsized(Fan 2 `Beside` Fan 2)
True
*Main> wellsized(Id 1 `Beside` Id 1)
True
*Main> wellsized  ((Fan 2 `Beside` Fan 2) `Above`(Fan 2 `Beside` Fan 2))
True
*Main> wellsized  ((Fan 2 `Beside` Fan 2) `Above` (Fan 2 `Beside` Fan 2)
`Above`(Fan 2 `Beside` Fan 2))
True
*Main> wellsized  ((Fan 2 `Beside` Fan 2) `Above` (Fan 2 `Beside` Fan 2)
`Above`(Fan 2 `Beside` Fan 3))
False
```

3. A single total function for safewidth

The previous functions have a number of similarities that can be combined to produce a single total function that returns a *Maybe* value, combining the two; *Just (width c)* when the circuit is in fact well-sized, and *Nothing* otherwise.

```
> safewidth :: Circuit -> Maybe Size
> safewidth cir = if (wellsized cir) then Just(width cir) else Nothing
```

The functional correctness can be confirmed with the following examples:

```
*Main> safewidth (Id 4 `Above` Id 4)
Just 4
*Main> safewidth ((Fan 2 `Beside` Fan 2) `Above` Stretch [ 2, 2 ] (Fan 2)
`Above` (Id 1 `Beside` Fan 2 `Beside` Id 1))
Just 4
```

Note the following two examples are not ‘well-sized’ in that there are differences in the width at different layers of the circuit, hence the returning value of *Nothing*.

```
*Main> safewidth (Id 4 `Above` Id 5)
Nothing
*Main> safewidth ((Fan 2 `Beside` Fan 2) `Above` (Fan 2 `Beside` Fan 2)
`Above`(Fan 2 `Beside` Fan 3))
Nothing
```

4. A function to list circuits side by side

The following functions consider placing a list of circuits side by side. The first function takes the identity primitive circuit (Id w) and places w copies of Id 1 side by side. The following beside prime function; *beside',* takes a list of circuits and returns a series of *Beside* circuits. In order to display the Circuit representation we need to modify the algebraic data type to derive show, this means that we can then print out and display the representation to the command prompt:

```
> beside' :: [Circuit] -> Circuit
> beside' [] = Id 0
> beside' (x:xs) = x `Beside` (beside' xs)

> beside :: Circuit -> Circuit
> beside (Id i)
>     | i > 0 = Id i `Beside` beside (Id (i - 1))
>     | otherwise = Id 0
```

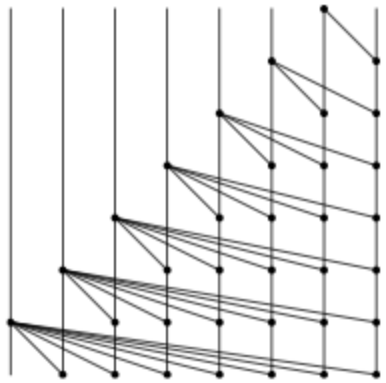
The functional correctness can be confirmed with the following examples:

```
*Main> beside' [(Id 1),(Id 1), (Id 1)]
Beside (Id 1) (Beside (Id 1) (Beside (Id 1) (Id 0)))

*Main> beside (Id 4)
Beside (Id 4) (Beside (Id 3) (Beside (Id 2) (Beside (Id 1) (Id 0))))
```

5. Defining a function to compute fan circuits

One obvious way to construct a parallel prefix circuit is to use lots of fans; for example, here is a naive parallel prefix circuit of size 8:



Given the relatively loose definition of the Stretch combinator in the introduction, it is assumed that the above circuit is represented as such:

```
(Stretch [7,1] (Fan 2)) `Above` (Stretch [6,1,1] (Fan 3) `Above` (Stretch
[5,1,1,1] (Fan 4)) `Above` (Stretch [4,1,1,1,1] (Fan 5)) `Above` (Stretch
[3,1,1,1,1,1] (Fan 6)) `Above` (Stretch [2,1,1,1,1,1,1] (Fan 7)) `Above`
(Stretch [1,1,1,1,1,1,1,1] (Fan 8))
```

Where the Stretch combinator has a list where the first value is the identity of the input line from which it starts, followed by subsequent values of input lines it stretches to, the fan value is thus the sum of the number of points to which the stretch fans out. One could argue that the circuit could be represented in the other following ways that would depend on the comprehension of the Stretch combinator, so we assume the previous definition. Note the use of set list comprehensions in list' and the else statement in scan' for short concise syntax:

```
> scan :: Size -> Circuit
> scan 0 = error "Scan must be positive integer"
> scan 1 = Id 1
> scan 2 = Fan 2
> scan s = scan' s 1 s
>     where
>     asc a = a + 1
>     desc b = b - 1
>     list' s a b = b:[(x `mod` x)+1 | x <- [1..a]]
>     scan' s a b = if b > 2
>         then Stretch (list' s a (desc b)) (Fan (asc a)) `Above`  scan' s
(asc a) (desc b)
>         else Stretch [(x `mod` x)+1 | x <- [1..s]] (Fan s)
```

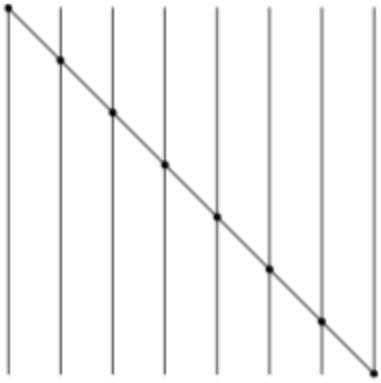
The functional correctness can be confirmed with the following examples:

```
*Main> scan 0
*** Exception: Scan must be positive integer
*Main> scan 1
Id 1
*Main> scan 2
Fan 2
*Main> scan 3
Above (Stretch [2,1] (Fan 2)) (Stretch [1,1,1] (Fan 3))
*Main> scan 4
Above (Stretch [3,1] (Fan 2)) (Above (Stretch [2,1,1] (Fan 3)) (Stretch
[1,1,1,1] (Fan 4)))
*Main> scan 8
Above (Stretch [7,1] (Fan 2)) (Above (Stretch [6,1,1] (Fan 3)) (Above (Stretch
[5,1,1,1] (Fan 4)) (Above (Stretch [4,1,1,1,1] (Fan 5)) (Above (Stretch
[3,1,1,1,1,1] (Fan 6)) (Above (Stretch [2,1,1,1,1,1,1] (Fan 7)) (Stretch
[1,1,1,1,1,1,1,1] (Fan 8)))))))
```

Please note that the final example has the identical semantic meaning to the previously given definition as per the below diagram, however the syntactical differences lay in the use of the above combinators being shown as infix and prefix in the latter.

6. Defining the serial function

Another obvious way is to accumulate values from left to right - for example.



Whilst this circuit is not optimal as it still has maximal depth, it does have the redeeming feature of the minimal number of nodes possible. The following serial function defines the circuit above to be represented as:

```
(Fan 2 `Beside` Id 6) `Above` (Id 1 `Beside` Fan 2 `Beside` Id 5) `Above` (Id
2 `Beside` Fan 2 `Beside` Id 4) `Above` (Id 3 `Beside` Fan 2 `Beside` Id 3)
'Above' (Id 4 `Beside` Fan 2  `Beside` Id 2) `Above`  (Id 5  `Beside` Fan 2
`Beside` Id 1) `Above` ( Id 6 `Beside` Fan 2)
```

The following function defines serial, note the use of pattern matching to cater for smaller sizes and even error handling and then the sub functions in the 'where' declaration and the use of recursion to in effect iterate over the value to compute the desired circuit representation:

```
> serial :: Size -> Circuit
> serial 0 = error "Serial param must be an int greater than 0"
> serial 1 = Id 1
> serial 2 = Fan 2
> serial s = fst s (serial' s 0 s)
>     where
>     fst s cir = (Fan 2) `Beside` (Id (s - 2)) `Above` cir
>     lst s = Id (s - 2) `Beside` Fan 2
>     asc a = a + 1
>     desc b = b - 3
>     serial' s a b = if b > 3 -- one for fst and one for lst
>         then (Id (asc a)) `Beside` (Fan 2) `Beside` Id (desc b) `Above`
serial' s (asc a) (b - 1)
>         else lst s
```

The functional correctness can be confirmed with the following examples:

```
*Main> serial 0
*** Exception: Serial param must be an int greater than 0
*Main> serial 1
Id 1
*Main> serial 2
Fan 2
*Main> serial 3
Above (Beside (Fan 2) (Id 1)) (Beside (Id 1) (Fan 2))
*Main> serial 4
Above (Beside (Fan 2) (Id 2)) (Above (Beside (Beside (Id 1) (Fan 2)) (Id 1))
(Beside (Id 2) (Fan 2)))
*Main> serial 8
Above (Beside (Fan 2) (Id 6)) (Above (Beside (Beside (Id 1) (Fan 2)) (Id 5))
(Above (Beside (Beside (Id 2) (Fan 2)) (Id 4)) (Above (Beside (Beside (Id 3)
(Fan 2)) (Id 3)) (Above (Beside (Beside (Id 4) (Fan 2)) (Id 2)) (Above (Beside
(Beside (Id 5) (Fan 2)) (Id 1)) (Beside (Id 6) (Fan 2)))))))
```

The final example serial 8, again differs from the definition state at the start only in that the Above and Beside combinators are infix in the previous and prefix in the latter examples.

These shown representations also validate against previous width, depth and wellsized functions!

```
*Main> width (Above (Beside (Fan 2) (Id 4)) (Above (Beside (Beside (Id 1) (Fan
2)) (Id 3)) (Above (Beside (Beside (Id 2) (Fan 2)) (Id 2)) (Above (Beside
(Beside (Id 3) (Fan 2)) (Id 1)) (Beside (Id 4) (Fan 2))))))
6

*Main> depth  (Above (Beside (Fan 2) (Id 4)) (Above (Beside (Beside (Id 1)
(Fan 2)) (Id 3)) (Above (Beside (Beside (Id 2) (Fan 2)) (Id 2)) (Above (Beside
(Beside (Id 3) (Fan 2)) (Id 1)) (Beside (Id 4) (Fan 2))))))
5

*Main> wellsized (Above (Beside (Fan 2) (Id 4)) (Above (Beside (Beside (Id 1)
(Fan 2)) (Id 3)) (Above (Beside (Beside (Id 2) (Fan 2)) (Id 2)) (Above (Beside
(Beside (Id 3) (Fan 2)) (Id 1)) (Beside (Id 4) (Fan 2))))))
True
```

7. A definition for Brent-Kung Circuits (optional)

Omitted.

8. Defining a datatype of segments

We say that the type s forms a semigroup if it supports an associative binary operator ⊕:

```
class Semigroup s where
(⊕)::s ➡ s ➡ s -- associative
```

This is like the more familiar type class Monoid, except without the unit element. For example, integers form a semigroup using addition:

```
instance Semigroup Int where
    (⊕) = (+)
```

Here is another semigroup, of ‘kissing segments’. Represent a segment of the natural number line by a pair of integers <i,j> with i <= j. Say that one such segment <i, j> ‘kisses’ another <k,l> precisely when k = j + 1; then they combine to yield the segment <i,l>.

```
> class Semigroup s where
>     (⊕) :: s -> s -> s -- associative
>

> instance Semigroup Int where
>     (⊕) = (+)

> data Segment = Segment (Int, Int) deriving(Show)

> data Segment' =
>    Segment' { i :: Int
>             , j :: Int
>             }
>             deriving(Show)

> instance Semigroup Segment where
>     Segment(i, j) ⊕ Segment(k, l) = if (i <= j) && (k == j + 1) then
Segment(i, l) else error "Invalid segments"

> instance Semigroup Segment' where
>     Segment' i j ⊕ Segment' k l = if (i <= j) && (k == j + 1) then Segment'
i l else error "Invalid segments"
```

The functional correctness can be confirmed with the following examples:

```
*Main> let s = Segment' 12 23
*Main> :t s
```

```
s :: Segment'
*Main> s
Segment' {i = 12, j = 23}

*Main> let x = Segment' 10 11
*Main> let y = Segment' 12 13
*Main>  x ⊕ y
Segment' {i = 10, j = 13

*Main> x
Segment' {i = 10, j = 11}
*Main> y
Segment' {i = 12, j = 14}

*Main>  x ⊕ y
*** Exception: Invalid segments
```

9. Defining a function group

The group function takes a list ws of natural numbers and a list xs of elements such that the sum ws = length xs and partitions xs into segments according to length ws eg

```
group [ 3, 2, 3, 2 ] "functional" = [ "fun", "ct", "ion", "al" ]
```

It is possible to define group with just two equations. The initial function design catered for empty lists with two differing equations, however this can be optimised to have the same function by rearranging the pattern matching. By assuming that we are indeed dealing with two properly defined lists with more than one element then we can process them as we desire by taking the head value of natural numbers list and then grouping and dropping this amount from the string list. This call is made recursively until either list is exhausted in which case we then match on the following wildcard pattern, returning an empty list and thus completing the function call.

```
> group :: [Int] -> [a] -> [[a]]
> group (x:xs) ys = (take x ys) : (group xs (drop x ys))
> group _ _ = []
```

The functional correctness can be confirmed with the following examples:

```
*Main> group [ 3, 2, 3, 2 ] "functional"
["fun","ct","ion","al"]

*Main> group [ 3, 2, 3, 2, 12, 2, 1, 9, 12, 101, 2, 2] "In computer science,
functional programming is a programming paradigm, a style of building the
structure and elements of computer programs, that treats computation as the
evaluation of mathematical functions and avoids state and mutable data."
["In ","co","mpu","te","r science, f","un","c","tional pr","ogramming is"," a
programming paradigm, a style of building the structure and elements of
computer programs, that tr","ea","ts"]

*Main> group [] "Testing"
[]

*Main> group [1, 2, 3] ""
["","",""]
```

10. A function to define apply

The apply function below takes a circuit and a list of the appropriate length whose elements are drawn from a semigroup. If the circuit is indeed a parallel prefix circuit (say, defined by scan, or serial, or brentkung), then apply will compute prefix sums.

```
apply :: Semigroup a => Circuit -> [a] -> [a]
```

A somewhat surprising result is that a single test case serves to prove that a purported parallel prefix circuit is correct. Assuming that the circuit has width w, it takes the list of point segments <1, 1>, <2, 2>, ..., <w, w> to the list of prefixes <1, 1>, <1, 2>, ..., <1, w>, using the semigroup of kissing combination.

```
apply :: Semigroup a => Circuit -> [a] -> [a]
apply cir xs
    | (width cir) == (length xs) = apply' xs
    | otherwise = error "Circuit and segment list length mismatch"
    where
        apply' (x:xs) = x ⊕ (apply' xs)
```

The functional correctness can be confirmed with the following examples:

```
*Main> let cir = (Above (Beside (Fan 2) (Id 1)) (Beside (Id 1) (Fan 2)))
*Main> apply cir [(Segment (1, 1)), (Segment (2, 2)), (Segment (3, 3))]
[(Segment (1, 3)]
```

11. Primitive elimination

It is possible to replicate the circuit representation with just one primitive. The hypothesis being, Fan can replace Identity. Let us consider the possibility of abandoning the Id primitive, One could simulate the Id primitive with Fan such at Fan 1 == Id 1, provided that if one intended to use the Fan primitive to represent Id values that it would required that we combined the Fan value of only 1 with Beside combinators. i.e.

Given that:

```
| = Id 1
```

```
| | = Id 2
```

And Fan 2 is defined as:

```
|\| = Fan 2
```

One could state that:

```
| = Fan 1
```

Represented as:

```
Id 1 == Fan 1
Id 2 == Fan 1 `Beside` Fan 1
Id 3 == Fan 1 `Beside` Fan 1 `Beside` Fan 1
```

Modifying the *Stretch* combinator could also give rise to other possibilities, in that if the declaration did not require the *Fan* circuit to be passed in and that it was given that the sum of the *Stretch* list was suffice to represent the fan value. Also as the first value at the head of the stretch list represents the Id value we could use stretch to represent Id. i.e.

```
Id 1 = Stretch[1] Fan 1
Id 2 = Stretch [2] Fan 1
```

12. A function to define ‘long zip’

The following defines a ‘long zip’ function, which is similar to the standard Haskell function *zipWith* except that it returns a result as long as its longer argument. If one argument is shorter than the other, the last elements of the result are copied from the corresponding elements of the longer list. The binary operator therefore has to take arguments of a common type, and return a result of that type.

```
lzw :: (a -> a -> a) -> [a] -> [a] -> [a]
```

For example,

```
lzw (+) [1,2,3,4] [5,6,7] = [6,8,10,4]
```

```
> lzw :: (a -> a -> a) -> [a] -> [a] -> [a]
> lzw _ [] a = a
> lzw _ a [] = a
> lzw f (x:xs) (y:ys) = f x y : lzw f xs ys
```

The functional correctness can be confirmed with the following examples:

```
*Main> lzw (+) [1,2,3] [2,4,6,8]
[3,6,9,8]
*Main> lzw (+) [1,2,3] [2,4,6,8,10]
[3,6,9,8,10]
```

We can also use other operators such as multiply, max and all other list operators if we desire.

```
*Main> lzw (*) [1,2,3] [2,4,6,8,10]
[2,8,18,8,10]
*Main> lzw (*) [1,2,3] [2,4,6,8,10,100]
[2,8,18,8,10,100]
*Main> lzw (max) [1,2,3] [2,4,6,8,10,100]
[2,4,6,8,10,100]
```

13. Defining a layout function

The following function generates an actual circuit layout in a hardware description language, from a term of type circuit. The essence of the translation is to determine the connections between wires. Note that each circuit can be thought of as a sequence of layers, and connections only go from one layer to the next. This function permits connections in both directions in order to accommodate the previous *Scan* output. It suffices to generate a list of layers, where each layer is a collection of pairs (i, j) with i < j denoting a connection from wire i on this layer to wire j on the next. The ordering of the pairs on each layer is not significant. We count from 0. For example, the Brent–Kung circuit of size 4 given earlier has the following connections: `[[(0, 1), (2, 3)], [(1, 3)], [(1, 2)]]` That is, there are three layers; the first layer has connections from wire 0 to wire 1 and from wire 2 to wire 3; the second a single connection from wire 1 to wire 3; and the third a single connection from wire 1 to wire 2. The following computes such layouts:

```
> type Layout = [Layer]
> type Layer = [Connection]
> type Connection = (Size, Size)

> zipCir :: Circuit -> [[Circuit]]
> zipCir cir = case cir of
>     (Above a b) -> (zipCir a) ++ (zipCir b)
>     (_) -> (layer cir) : []
>     where
>         layer l = case l of
>             (Beside a b) -> (layer a) ++ (layer b)
>             (x) -> x : []

> zipList :: [Int] -> [[Circuit]] -> [[(Int, Circuit)]]
> zipList (y:yx) (xs:xss) = [((y + (extract x) - 2),(x)) | x <- xs] : zipList
yx xss
> zipList _ [] = []
> extract x = case x of
>     (Id x) -> x
>     (Fan x) -> x
>     (Stretch xs x) -> length xs

> layout :: Circuit -> Layout
> layout cir = pack(zipList [0..] (zipCir cir))
>     where
>         pack(xs:xss) = [package x | x <- xs, filterId x] : (pack xss)
>         pack(_) = []
>         filterId (_,Id _) = False
>         filterId _ = True
>         package x = case x of
>             (i, (Fan f)) -> (i, (i + f - 1))
```

```
>              (i, (Stretch ys f)) -> ((head ys) -1, (sum ys)-1)
```

Brent–Kung circuit, as expected:

```
*Main> layout (Above (Above (Beside (Fan 2) (Fan 2)) (Stretch [2,2] (Fan 2)))
(Beside (Beside (Id 1) (Fan 2)) (Id 1)))
[[(0,1),(2,3)],[(1,3)],[(1,2)]]
```

Serial 3 & 10 circuits, as expected:

```
*Main> layout (Above (Beside (Fan 2) (Id 1)) (Beside (Id 1) (Fan 2)))
[[(0,1)],[(1,2)]]
*Main> layout (Above (Beside (Fan 2) (Id 8)) (Above (Beside (Beside (Id 1)
(Fan 2)) (Id 7)) (Above (Beside (Beside (Id 2) (Fan 2)) (Id 6)) (Above (Beside
(Beside (Id 3) (Fan 2)) (Id 5)) (Above (Beside (Beside (Id 4) (Fan 2)) (Id 4))
(Above (Beside (Beside (Id 5) (Fan 2)) (Id 3)) (Above (Beside (Beside (Id 6)
(Fan 2)) (Id 2)) (Above (Beside (Beside (Id 7) (Fan 2)) (Id 1)) (Beside (Id 8)
(Fan 2)))))))))
[[(0,1)],[(1,2)],[(2,3)],[(3,4)],[(4,5)],[(5,6)],[(6,7)],[(7,8)],[(8,9)]]
```

Scan 3 circuit, not as expected:

```
*Main> layout (Above (Stretch [2,1] (Fan 2)) (Stretch [1,1,1] (Fan 3)))
[[(1,2)],[(0,2)]]
```

Unfortunately the combinations of functions is terribly inefficient, this is due to first zipping the circuit, then creating tuples, and then grooming the list for the unwanted Id types and expanding the *Stretch* types. One approach that might ease the overhead is to consider the use of *Just* and *Maybe* in the type declaration. Although not complete the following examples gives some insight into the alternative approach:

```
> type Layout' = [Layer']
> type Layer' = [Connection']
> type Connection' = Maybe (Size, Size)

> layout' :: Circuit -> Layout'
> layout' cir = case cir of
>     (Above a b) -> (layout' a) ++ (layout' b)
>     (_) -> (layer cir) : []
>     where
>         w = width cir
>         layer l = case l of
>             (Beside a b) -> (layer a) ++ (layer b)
>             (_) -> (calc l 0) : []
>         calc a i = case a of
>             (Stretch (xs) f) -> Just ((head xs) -1, (sum xs)-1)
>             (Fan f) -> Just (i, (i + f - 1))
>             (Id i) -> Nothing
```

14. Defining an SVG and Output function

The following functions permit the output of an SVG file of the circuit entered.

```
> type String' = Char

> svg :: (Layout , Size) -> [String']
> svg (lx, s) = concat [createHeader (s - 1), concat(createLine s s),
concat(createPoint lx 0), concat(createFan lx 0), footer]
>     where
>         header = " <svg width='$20' height='$20' viewBox='-10,-10,$20,$20'
xmlns='http://www.w3.org/2000/svg' version='1.1'> \n"
>         line = " <line x1='$00' y1='0' x2='$00' y2='£00' " ++ style ++ "/>
\n"
>         fan = " <line x1='α00' y1='β00' x2='ɣ00' y2='ϱ00' " ++ style ++ "/>
\n" -- Pattern match on Greek symbols
>         point = " <circle cx='$00' cy='£00' r='7' fill='black'
stroke-width='0'/>\n"
>         style = " stroke='black' stroke-width='2' "
>         footer = " </svg> "
>         replace _ _ [] = []
>         replace a b (x:xs)
>             | x == a = b:replace a b xs
>             | otherwise = x:replace a b xs
>         createHeader s = (replace '$' (intToDigit s) header)
>         createLine i c
>             | i >= 0 = (replace '$' (intToDigit i) (replace '£' (intToDigit
c) line)) : createLine (i-1) c
>             | otherwise = []
>         createPoint (xs:xss) i = if length xs > 1
>             then point' xs i : point'' xs i  : createPoint ((tail xs):xss) i
>                 else point' xs i : point'' xs i  : createPoint xss (i + 1)
>         createPoint [] _ = []
>         -- Takes list and int, extracts fst val of tuple, replaces '$' for
int val, point'' works on snd val of tuple where i is layer
>         point' xs i = (replace '$' (intToDigit(fst(head xs))) (replace '£'
(intToDigit i) point))
>         point'' xs i = (replace '$' (intToDigit(snd(head xs))) (replace '£'
(intToDigit (i + 1)) point))
>         createFan (xs:xss) i = if length xs > 1
>             then fan' xs i : createFan ((tail xs):xss) i
>                 else fan' xs i : createFan xss (i + 1)
>         createFan [] _ = []
>         -- Takes list and int and subs out greek letters for values, these
are nested, working through tuple vals and layer i val
```

```
>         fan' xs i = (replace 'α' (intToDigit(fst(head xs))) (replace 'β'
(intToDigit i) (replace 'ɣ' (intToDigit(snd(head xs))) (replace 'ϱ'
(intToDigit (i + 1)) fan))))

> output :: FilePath -> Circuit -> IO()
> output file c = writeFile file (unlines([svg(layout c, width c)]))
```

The functional is executed with the following commands:

```
*Main> output "serial3.svg" (serial 3)
*Main> output "serial9.svg" (serial 9)
*Main> output "scan3.svg" (scan 3)
*Main> output "scan9.svg" (scan 9)
```

Whilst the serial functions execute as expected, time escaped and the Scan functional output was not able to be fully completed which was systemic from the efforts on point 13, the layout function. It is expected that the following would iterate through the Scan types to produce the necessary deviations of the list contained, however the matter of nested lists with in the connection type would need to be resolved:

```
package x = case x of
      (i, (Fan f)) -> (i, (i + f - 1))
      (i, (Stretch ys f)) -> ((head ys) -1, (sum ys)-1) : package((i, (Stretch
(tail ys) f)))
```

Conclusions

Functional programming leads to increased modularisation and component oriented programming. Indeed this is evident from the low level when building functions, catering for each parameter pattern match, breaking down problems to solve base cases and glueing together smaller functions to produce more complex ones. This philosophy is prevalent at a higher level also, when taking library functions and combining them to produce more sophisticated results. Indeed whilst the solutions in this paper may not be the most efficient, they are separated in such a way that they could be utilised for other purposes, decreasing coupling and increasing cohesion. Where possible the type declarations of functions has been omitted to keep code clean, concise and less verbose as Haskell is able to infer type declarations.

A concerted effort has been made to apply a variety of functional concepts such as; algebraic definitions, set list comprehensions, higher order functions with currying, lambdas and functional composition.

The steep learning curve of the functional paradigm and the syntax of Haskell can hinder the adoption of pure functional programming languages. However with familiarity comes increased insight into efficiency and optimisation. Indeed by the end of the paper it became obvious that there was still room for many improvements. Utilising list operations and recursion over iteration and accumulating parameters to preserve referential transparency would have been more favoured and more time to refactor the solutions to adhere to concise functional standards.

The solutions are somewhat verbose for the functional paradigm and with more time further refinement could have lead to better optimisations, the code could benefit from refining the amount of parentheses and utilise function composition with dot syntax for more elegant code. More importantly with the final two sections, 13 and 14, it would have been beneficial to explore alternative data structure to lists, in hindsight tree structures may have vastly improved the function computation time.

Given more time it would be interesting to investigate the representation of other circuits such as point 7, the Brent Kung circuit as well as other types. This foray into the Functional Paradigm with Haskell has undoubtedly changed my personal perspective on software development for the better and there was a certain satisfaction, almost ‘joy’ to improving programmatic conciseness by the end.