Functional Programming

Parallel Prefix Circuits

Stefan Cross

FPR Coursework Submission, Hilary Term 2014

TODO:

Introduction and definitions

Q8 - Show that this kissing combination operation is associative, when it is defined. Define a datatype of segments, and make it an instance of Semigroup.

General:

Reconsider title headings and vocab

Introduction

The functional programming paradigm is becoming increasingly prevalent with in recent years. Great strides have been taking to bring the approach and concepts into mainstream languages such as Microsoft's F#, Java's with Scala, and other Java Virtual Machine (JVM) alternatives. Industry is reaping the rewards of the academics astute application of mathematical concepts into high-level programming languages that offer improved handling of multi-core processing due to immutability and the concept of referential transparency…

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 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 compu- tation of size *n* takes as input a list $[x_1,\ldots,x_n]$ of length *n* and returns as output the list $[x_1,\ x_1\ x_2,\ \ldots,\ x_1\ x_2 \cdots x_n\ ]$—the 'sums' (with respect to binary oper- ator) 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 GPUs.

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 log- arithmic time on linearly many processors. Moreover, the computation follows a fixed sequence, oblivious to the actual data, so it is amenable to implementa- tion in hardware. For example, here is a so-called Brent–Kung circuit of size 16, computing the prefix sums in parallel in only 7 steps on 16 processors:

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)
```

As this example is often utilised to as input or alternatively to prove the correctness of a function implementation we shall refer to this as the standard circuit representation.

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.

```
> 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 take 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 placed 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:

```
> data Circuit
>     = Id Size
>     | Fan Size
>     | Beside Circuit Circuit
>     | Stretch [Size] Circuit
>     | Above Circuit Circuit
>     deriving(Show)

> 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
```

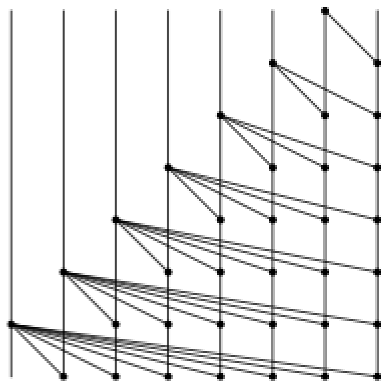
The functional correctness can be confirmed with the following examples:

```
*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 other following ways that would depend on the comprehension of the Stretch combinator, so assuming 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)
```

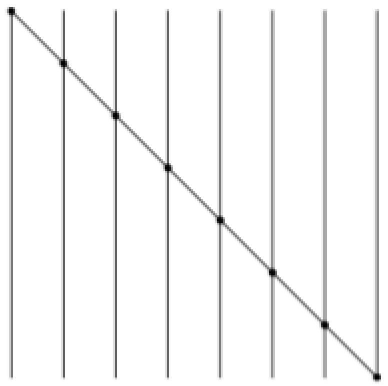
The functional correctness can be confirmed with the following examples:

```
*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 prefiously given definition as per below the 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 function serial

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. Define 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] ""
["","",""]
```

Conclusions

Functional Programming leads to increased modularisation. 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 complexed ones. This philosophy is prevalent at a higher level also, when taking library functions and combining them to produces 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.

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 lists operations and recursion over iteration and accumulating parameters to preserve referential transparency would have been more favoured and every effort was made to continually refactor the solutions to adhere to functional standards.