Program Design

Stuart Paton

February 21, 2013

1 Introduction

In this document I will discuss my design and modelling considerations for my project's implementation by thorough discussion of each task chosen.

2 Internals

To construct accurate models of each Catalan structure I have, in the file "Internal.hs", created a type class called Catalan which is defined as follows:

```
class Catalan a where
        empty :: a
        cons :: a -> a -> a
        decons :: a -> (a,a)
```

In this type class, I have thee operators: *empty*, *cons* and *decons* which are the operators for an empty structure; to construct, or compose a strucutre and deconstruct (or decompose) a structure.
This module also holds type synonyms for the type **Permutation** which is:

```
type Permutation = [Int]
```

As we can see, a permutation is just a list of integers.
The functions for composing a bijection for two catalan structures is also stored in this module. It is a recursive function which runs a bijection by decomposing a structure and checking it does not have a value Nothing. It looks as follows:

```
bijection :: (Catalan a, Catalan b) => a -> b
bijection w = case decons w of
                Nothing -> empty
                Just (u,v) -> cons (bijection u) (bijection v)
```

3 Catalan Structures

The module **Catalan Structures** is the module which contains all of the bijections I have implemented within the program. These bijections will be analysed by finding statistics for each structure and investigating which statistics are preserved for each bijection.
Currently the standard bijection from [2] has been implemented and is as follows:

```
ssp2dp :: StackSortablePermutation -> DyckPath
ssp2dp [] = []
ssp2dp ssp = [U] ++ ssp2dp (red (alpha, beta)) ++ [D] ++ ssp2dp beta
        where
        (alpha, beta) = stripMaybe $ decons ssp
```

This bijection converts a stack sortable permutation to a Dyck path. It does this by using the recursive formula: $f(\pi) = uf(\pi'_L)df(\pi_R)$ and $f(\epsilon) = \epsilon$. π'_L is defined as "the permutation of $1, 2, ..., |\pi_L|$ obtained from $|\pi_L|$ by subtracting $|\pi_R|$ from each of its letters."[2]
To perform the subtraction, I have used a function which I have named **red** for reduction. It is as follows:

```
red :: (StackSortablePermutation, StackSortablePermutation)
        -> StackSortablePermutation
red ([], beta) = []
red (x:xs, beta) = x - pi_r : red (xs, beta)
        where
        pi_r = length beta
```

It takes a pair of StackSortablePermutation's as input and returns a single StackSortablePermutation. It does this via the method for obtaining $|\pi_L|$ above.

4 Dyck Paths

To model Dyck paths, I created a module which I named **DyckPath**. To represent a Dyck Path I have a list of up-steps and down-steps. Each step is represented by the algebraic data type Step which uses the encoding U for an up-step and D for a down-step. A full Dyck path is represented as a list of steps shown below in the type synonym DyckPath.

```
data Step = U | D deriving (Eq, Show)
type DyckPath = [Step]
```

Next I created an instance of Catalan for the type DyckPath. It is constructed as follows:

```
instance Catalan DyckPath where
        empty = []
        cons alpha beta = mkIndec alpha ++ beta
        decons gamma = stripMaybe $ decompose gamma
```

The function *mkIndec* takes a Dyck path, alpha, and makes an indecomposable DyckPath by prepending a U to the start of alpha then a D to the end of alpha. Then to fully compose our Dyck path with a given alpha and beta we just append beta to mkIndec alpha as is shown above.
Decomposing a Dyck path is the hardest task faced in the design and implementation of the model of a Dyck path. To do this we make a function called *decompose* with takes in a parameter gamma, where gamma is a full Dyck path. Our function decompose looks like the following:

```
decompose :: DyckPath -> Maybe (DyckPath, DyckPath)
decompose [] = Nothing
decompose xs@(U:xt) = Just (map fst (init ys), map fst zs)
                  where
                    0:ht = height xs
                    (ys, zs) = span(\(_, h) -> h > 0) $ zip xt ht
```

This function starts off by taking a Dyck path and mapping each element of the Dyck path to the height of each element in the half, except from the first element which is disregarded. This is shown by 0:ht. In order to obtain (ys, zs) we use the span function which splits the list into our alpha and beta lists disgregarding the down-step which is appended to alpha. To finish off we apply the following to *Just*. We map the first element of the pair to all the elements of ys except the first, and we then map the first element of the pair to zs.
For the height function, it is defined as follows:

```
height :: DyckPath -> [Int]
height = scanl (+) 0 . map dy
         where
         dy U = 1
         dy D = -1
```

As this is in $O(n)$ time instead of the next example which is in $O(n^2)$ time it is more efficient as it repeatedly adds the partal sums starting from 0 to each element which was mapped to their dy values. The next example is the $O(n^2)$ version.

```
height :: DyckPath -> [Int]
height = map sum . inits .map dy
          where
            dy U = 1
            dy D = -1
```

Here we start by mapping our encodings of U and D to create a list of 1's and -1's. By applying this to the function inits, it creates a list of partial sums which we then fully add together using "map sum" and we have the height of each element.

5 Stack Sortable Permutations

To model Stack Sortable permutations, or 132-avoiding permutations I have used the standard model for constructing them within my Haskell module named **StackSortPerm**.
A stack sortable permutation is a permutaiton which avoids the permutation 132. As such, it is in the form $\alpha n \beta$. That is, to say $\alpha \prec \beta$ or, all the elements of α are less then all the elements of β and n is the largest element of the permutation. To model this we make an instance of the Catalan type class:

```
instance Catalan StackSortablePermutation where
        cons = mkIndec
        decons = decompose
```

Where,

```
type StackSortablePermutation = Permutation
```

and as defined in Internals:

```
type Permutation = [Integer]
```

To create our cons operator we must ensure that we take as parameters our alpha and beta and then generate n. This is created as follows:

```
mkIndec :: StackSortablePermutation -> StackSortablePermutation
           -> StackSortablePermutation
mkIndec alpha beta = alpha ++ [n] ++ beta
        where
        n = toInteger $ length (alpha ++ beta) + 1
```

To generate n, we take the length of alpha and the length of beta and then add one to the result, and finally convert it from *Int* to *Integer*.
Finally to decompose our permutation σ we use the following function:

```
decompose :: StackSortablePermutation
             -> (StackSortablePermutation, StackSortablePermutation)
decompose sigma = removeHeadSnd $ break (l ==) sigma
        where
        l = toInteger $ length sigma
```

Here, to decompose σ into a pair of stack sortable permutations, (α, β) I firstly use the *break* function from the Haskell prelude, which splits a list into a pair of lists over a given condition. So here we are splitting the permutation sigma where at the position n where $\sigma_n = l$. In this case l is $|\sigma|$. Secondly, I use the function *removeHeadSnd* which is a function to remove the head of the second list in a pair since when the function break is applied, the element it splits the list over is the head of the second list. The function is the following:

```
removeHeadSnd :: (t, [a]) -> (t, [a])
removeHeadSnd (alpha, beta) = (alpha, tail beta)
```

As we can see, it just returns the original first list of the pair, along with the tail of the second list.
Once this is created, the function will decompose to its original α and β.

5.1 Graphics

To create permutation matrices on screen, the GTK bindings for Haskell were used in from the package "Graphics.UI.Gtk". To build the window, three functions were created: *drawPerm*; *renderFigure*; and *figure2render*. Starting off *figure2render* looks as follows:

```
figure2Render :: StackSortablePermutation -> DC
figure2Render perm = P.plotPerm $ permToString perm
```

Here, in order to render the permutation matrix we use the function *plotPerm* from Anders Claesson's sym-plot package [1]. This creates a permutation of type DC when given a string as input.
Next we have to render the permutation matrix in order to visualise it. This is acheived using the *renderFigure* function which is as follows:

```
renderFigure :: DrawingArea -> StackSortablePermutation
                -> EventM EExpose Bool
renderFigure canvas perm = do
        liftIO $ defaultRender canvas $ figure2Render perm
        return True
```

This function starts off by taking in the drawing area and permutation and parameters then sequentially carries out the *liftIO* operation and then returns *true* as a *Bool* must be returned by the function. *liftIO* is a function which simply takes the regular function *defaultRender* and converts it to the type *m a*. To render the figure, the defaultRender function is used. It renders a diagram for the drawing area given and rescales it to use the full area available.
Finally we define the *drawPerm* function which is as follows:

```
drawPerm :: StackSortablePermutation -> IO ()
drawPerm perm = do
        initGUI
        window <- windowNew
        canvas <- drawingAreaNew
        canvas `on` sizeRequest $ return (Requisition 256 256)
        set window [windowTitle := "Permutation Matrix",
                containerBorderWidth := 10, containerChild := canvas ]
        canvas `on` exposeEvent $ renderFigure canvas perm
        onDestroy window mainQuit
        widgetShowAll window
        mainGUI
```

This function draws a window for the permutation matrix and shows the rendered image of the permutation matrix within the window. Lines 3 to 7 and

lines 9 to 11 are standard kit for building a window with GTK. In lines 3 to 7 we create a new window, and a new canvas, then set the size of the canvas and set the window properties. In lines 9 to 11 we state that the application will terminate when the window is terminated, then that the window will be displayed and the main loop for the graphical user interface will run.
In line 8, which is the line we are most interested in, states that we will redraw the figure and place it on the canvas.

6 Young Tableaux

To model Young Tableaux I have used the standard model for constructing them within my Haskell module named **Young Tableaux**.
A Young Tableaux is a table consisting of a finite collection of cells arranged in left justified rows with the row lengths weakly decreasing.

7 Statistics

In order to investigate the statistics related to each Catalan structure, I have a section of each module which defined the statistics which relate to individual structures.

7.1 Dyck Path Statistics

For Dyck paths, I have the following statistics:

```
--Number of up steps
uCnt :: DyckPath -> Int
uCnt = count U

--Number of down steps
dCnt :: DyckPath -> Int
dCnt = count D

--Number of returns to the x axis
returnsXAxis :: DyckPath -> Int
returnsXAxis dp = count 0 $ height dp

--Number of peaks
{- algorithm:
1) split into lists at each 0
2) find number of highest element of each list
3) sum of counts from step 2
-}
peaks :: DyckPath -> Int
```

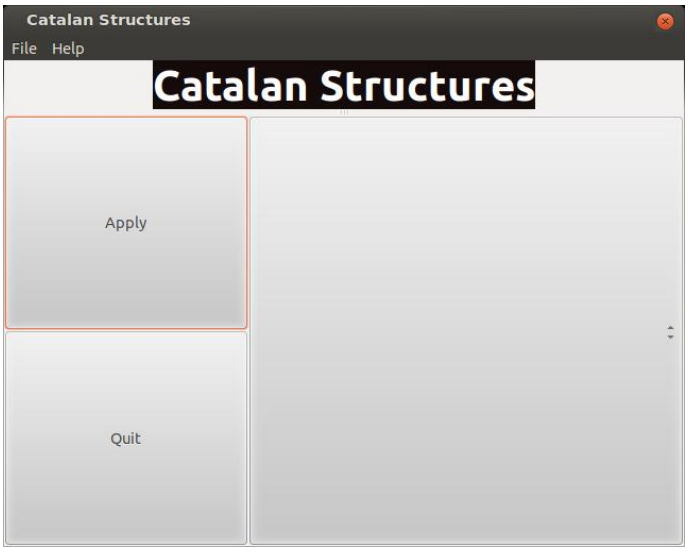
```
peaks dp = sum $ largestElemCnt $ split
        where
        split = splitWhen (== 0) $ height dp

--Height of the Dyck Path
heightStat :: DyckPath -> Int
heightStat dp = maximum $ height dp
```

Although the comments describe what each statistic is for, I will systematically describe how each statistic is obtained and in relation to the base set of permutation statistics in [2] what each statistic will relate to.
The statistics above count the number of up-steps, down-steps, peaks and returns to the x-axis. Respectively these will relate to the asc, dsc, peak and valley statistics for permutations.
The statistic *uCnt* is obtained by counting the number of up-steps in the path, *dCnt* is defined similarly. The *returnsXAxis* statistic counts the number of returns to the x-axis by calculating the height of each step in the list, then counting the number of heights that are 0 and subtracting 1 since we always start on the x-axis so the height of the first step is always 1. The *peaks* statistic counts the number of peaks by calculating what the largest height of each section is and keeping a count. Finally, the *heightStat* statistic finds the maximum peak of the path and returns it.

8 Graphical User Interface

The main interface for the program looks as follows:



Modelled using Glade [3]

To build this main interface, the user interface design software, Glade, was used for modelling and the Gtk bindings for Haskell were used to implement each action from the interface. To model the interface I adopted the standard method for modelling any user inface with Glade and Haskell, it is as follows:

```
main :: IO ()
main = do
        initGUI

        builder <- initBuilder

        <Omitted content>

        widgetShowAll main_window
        mainGUI
```

In the above code, we define the function *main* which is of type *IO ()*. Within the main body of the function, using the IO monad we sequentially carry out a number of actions. Firstly we initalise the GUI toolkit for Haskell's Gtk bindings, and create a builder. To create the builder used for our user interface we define the function *initBuilder*.

```
initBuilder :: IO Builder
initBuilder = do
        builder <- builderNew
        builderAddFromFile builder "mainGui.glade"
        return builder
```

In the *initBuilder* function, we create a new builder using the function *builderNew* and then add the user interfaces *.glade* file to the builder and finally return the builder.
Continuing with the function main, to end the function we explicitly tell the program to show the main window, denoted here as $main_windowandthencallmainGUItorunthemainGUIloopandc$

References

[1] Anders Claesson, sym-plot. last accessed 21 february 2013.*, https://www.github.com/akc/sym-plot, 2013.*

[2] Anders Claesson and Sergey Kitaev, Classification of bijections between 321- and 132-avoiding permutations*, 2008.*

[3] The Glade Project, Glade*, http://glade.gnome.org/.*