AFP Project Report - Spreadsheet

Hassan Al-Awwadi (6998704) Vlad-Auras Nohai (5373123)
Apoorva Anand (2037610)

1 Introduction

For our AFP project, we decided to build a Spreadsheet-like tool on top of a web server. In doing so, we worked with different aspects of application development - front-end development in the browser with Elm, back-end server work with Haskell, and writing algorithms for data structures such as graphs.

A spreadsheet is a tool for organizing, analyzing, and storing information in a tabular form, where each cell can contain primitive data, formulas and/or functions. To help with the propagation of information through the cells, we use two graphs - The forward graph and the backward graph. Every node in our forward graph points to nodes that depend on it for information; this graph is used for the propagation of changes through our spreadsheet. The backward graph is for convenience and is a trade-off between time and space. If a dependency is changed because of a change in a formula, instead of going through every edge of a node in the forward graph, in the backward graph we point to every node's parent. Thus if we change that node to depend on another parent, or no parent at all, we know this immediately through the backward graph. A crucial constraint here is that these graphs must be *Directed Acyclic Graphs* (DAGs).

Graphs in Haskell are (in)famously not a "solved" problem. There are 4 main approaches:

(i) `Data.Graph`
(ii) `Alga`
(iii) `FGL`
(iv) `Hash-Graph`

All of them have comparable performance (as shown *here*), each with their niggles.

(i) `Data.Graph` is poor at repeated modifications to our graph - something that is vital to a spreadsheet.
(ii) `Alga` is probably the most modern and well-developed solution with lots of graph algorithms, and yet we could only leverage this after constructing these graphs in what Alga calls "Algebraic Graphs", we didn't think this was worth it.
(iii) `FGL` had some bits and pieces that we wanted, and some that didn't exist, which meant we had to write it ourselves.
(iv) And finally `Hash-Graph`, a graph library based on constructing graphs built on top of Hash Maps. This library wasn't used by a lot of people.

Because of all the above reasons, for better or for worse, we decided to stick to our placeholder solution - A graph based on `Data.Map`.

2 Important Datatypes

2.1 Graph

Our graph uses an adjacency list representation

```
type Graph = Map (Int, Int) [(Int, Int)]
```

Where every coordinate points to other coordinates.

2.2 Array

Our array holds the formula for how the information inside a cell is built, along with its evaluated value.

```
type Arr   = Map (Int, Int) (Formula Int, Int)
```

2.3 Formula, Target and Operations

We were able to support the following features of a spreadsheet:

- Raw data input (For example, just the number "42")
- Reference-based information - Both absolute and relative. Absolute reference points to an exact location in our spreadsheet, whereas a relative reference points to a cell based on the cell that we write this formula in
- Basic arithmetic operations

```
data Formula a where
  Raw :: a -> Formula a
  Ref :: Target -> Target -> Formula a
  Op  :: (a -> a -> a) -> Formula a -> Formula a -> Formula a
  Un  :: (a -> a) -> Formula a -> Formula a
  deriving Show

data Target
  = Loc Int
  | Rel Int
  deriving Show
```

2.4 Spreadsheet

Finally, the spreadsheet is just a record that holds all of these together.

```
data Spreadsheet = S { table :: Arr, backward :: Graph, forward :: Graph }
  deriving Show
```

2.5 Other Stuff

There are also some datatypes used in Elm for the frontend part of our application, and some others for our middleware. We are not going to describe them in detail here.

2.5.1 Elm Models

```
type Msg =
    PressCell Coord
  | ReleaseMouse
  | PressTopBorder Int
  | PressBotBorder Int
  | HoverOverTopBorder Int Bool
  | HoverOverBotBorder Int Bool
  | HoverOver Coord Bool
  | EditModeCell Coord
  | EditCellUpdate Int Int String
  | AddRows Int
  | AddColumns Int
  | ConfirmEdit
  | ResponseServer
    (Result Http.Error (Result ((Int,Int), String) (List ((Int,Int), String))))
  | PressedLetter Char
  | PressedControl String

type alias Coord = {
  x : Int,
  y : Int
  }

type alias Cell =
    {
      pos_x : Int
    , pos_y : Int
    , content : String
    , value   : Result String String
    }

type alias Model =
  { max_x  : Int
  , max_y  : Int
  , values : A.Array (A.Array (Cell))
  , selectedRange : (Coord, Coord)
  , clickPressed : Bool
  , editingCell : Maybe Coord
  , clipboard : Maybe (Coord, Coord)
  }
```

We use Scotty as our web framework, and use JSON to marshall data between Elm and Haskell.

3 Important Concepts and Techniques

Several classic graph algorithms were implemented by us as part of this project. They rely on, and maintain the acyclic nature of our graphs.

3.1 Graph Algorithms

3.1.1 Depth First Search (DFS)

If the DFS of our graph hits a node that it has seen before already, then we know that we've found a cycle. This is done immediately after the user gives a formula input.

```
dfsDetect :: (Int, Int) -> Map (Int, Int) [(Int, Int)] -> Bool
dfsDetect i g = not $ evalState (dfs' i g) S.empty

dfs' :: (Int, Int) -> Map (Int, Int) [(Int, Int)] -> State (S.Set (Int, Int)) Bool
dfs' i g = do
  visited <- get
  if S.member i visited
  then return False
  else do
    let restBools = map (\x -> evalState (dfs' x g) (S.insert i visited))
                        (M.findWithDefault [] i g)
    if any not restBools then return False else return True
```

3.1.2 Topological Sort

To make sure that we propagate the changes through the right nodes, and in the right order, we must topologically sort a graph, i.e., calculate the formula of a node, only after the possible recalculation of it's parent - otherwise we end up with incorrect information.

The Topological Sort also acts as a sanity check, because it too reveals if somehow have introduced a cyclic dependency in our graph.

```
topSort :: Map (Int, Int) [(Int, Int)] -> [(Int, Int)]
topSort g = evalState (topSort' g (M.keys g)) M.empty

data Mark = Perm | Temp deriving Eq
type MarkedStatus = Map (Int, Int) Mark

topSort' :: Map (Int, Int) [(Int, Int)] -> [(Int, Int)]
         -> State MarkedStatus [(Int, Int)]
topSort' _ []     = return []
topSort' g (n:ns) = do
  ms <- get
  if M.findWithDefault Temp n ms == Perm
  then topSort' g ns
  else do
    visit g n
    put (M.insert n Perm ms)
    restRes <- topSort' g ns
    return (n : restRes)

visit :: Map (Int, Int) [(Int, Int)] -> (Int, Int) -> State MarkedStatus ()
visit g currNode = do
  ms <- get
  if M.findWithDefault Perm currNode ms == Temp
  then error "Top Sort detected a cycle"
  else do
    put (M.insert currNode Temp ms)
    let neighbours = M.findWithDefault [] currNode g
    mapM_ (\x -> unless (M.findWithDefault Temp x ms == Perm) (visit g x)) neighbours
```

One thing to notice is that, thanks to the State Monad, we get to write these algorithms "imperatively", having to make no changes from the textbook definitions of these algorithms. In fact the implementations of both these algorithms are directly based off of wikipedia!

3.2 Parsing and Evaluation

To implement the spreadsheet, we had to support a tiny language. This is for the calculation of formulas. The grammar of our language is:

```
Formula  ::= Integer
           | 'c' RTerm 'r' RTerm
           | 'r' RTerm 'c' RTerm
           | UnaryOp Formula
           | BinaryOp Formula Formula

RTerm    ::= '$' Integer
             | Integer

BinaryOp ::= '*' | '+' | '-' | 'max' | 'min'

UnaryOp  ::= '(-)'

Integer  ::= [0-9]+
```

We used **Parser Combinators** to implement the parser for the grammar. Specifically, we used the `Text.ParserCombinators.ReadP` library.

For the evaluation, the following interpreter for Formulas was implemented -

```
eval :: Arr -> (Int, Int) -> Formula Int -> Int
eval _   _     (Raw i)      = i
eval arr (x,y) (Ref x' y')  = let
  (nx,ny) = (locate x x', locate y y')
  (_, r) = arr M.! (nx, ny)
  in r
```

```
eval arr (x,y) (Op f l r) = let
  l' = eval arr (x,y) l
  r' = eval arr (x,y) r
  in f l' r'
eval arr (x,y) (Un f z) = let
  z' = eval arr (x,y) z
  in f z'
```

Here `locate` is a function used to find out the exact information about the reference in our formulas.

4 Results and Examples

1) We were able to make quite a nice GUI, we highly recommend you build the project and check it out. Elm was very pleasant to work with, even for someone not experienced in front-end work.
2) Propagation of data through our `Data.Map` based graph was implemented.
3) A tiny language's parser and evaluator were also implemented.

```
-- Enter these into the text fields of our frontend to see results

-- Binary operator example
cell00 = "+ 4 2"
cell03 = "+ 4 + 4 2 "
-- Yes we're using Prefix notation, sorry :)

-- absolute positioning
cell04 = c0-r0 -- refers to cell00

-- relative positioning
cell02 = c$+1-r$+0 -- refers to cell03
-- things after $ are integers used to calculate data
-- based on the current cell's positioning
```

To test out the application, build the frontend using elm (or simply use elm reactor within the client subdirectory, and then navigate to the index page), and do a cabal run for the backend.

5 Reflection and Improvements

The project showed Haskell's power in being a "boring" language. The fanciest thing we ended up using were Monads (State Monads, most of the time) and although we were initially worried because a lot of literature about implementing Spreadsheets and its algorithms were focused on imperative algorithms, we had no trouble translating these into Haskell.

We actively wanted to avoid unnecessary complexity, so our choices in terms of libraries and data structures also worked out well. Scotty was straightforward to use. We were able to get our hands dirty with the "Parse, don't validate" style of programming, which lent itself very well to the problem we were trying to solve.

5.1 Some Struggles

5.1.1 Propagation

This was hard. Graphs are not a solved feature in Haskell, so we had to do a lot of graph work ourselves, and we are no experts. Having to reinvent the wheel (badly) because no user-friendly wheel is present was annoying.

5.1.2 Front-end

Our frontend looks good, we think, but it took a lot of time to get it looking good and working good. Largely we can write this off as a lack of personal experience. None of us were real frontend devs.

5.1.3 Web Dev

We are no webdevs, and because of this, we struggled with annoying bugs for way longer than we needed to. One example is that for a while our server wasn't receiving any requests from our client thanks to an incorrect CORS policy. Unfortunately, we didn't even know what a CORS policy was, to be able to fix it :(

5.2 Future Improvements

There remain a lot of features that we wanted to add but ended up leaving out because of time.

5.2.1 Infix operators

Right now we use a prefix notation, which is nice because we don't have to bother with associativity, but is not the nicest user experience.

5.2.2 Domain Specific Language

We have a few prebuilt functions for operating on data, but lack any way for users to create their own lambda functions. This is sorely missed, especially since we only have a very select number of functions to start with.

5.2.3 Spreadsheet features

Talking about a limited number of functions, one thing to note is that we only have one unary function, a few binary functions, but 0 more than binary functions. A particular feature that we are missing out on is functions that work on *ranges* of data, like `SUM A1:A10` or the like. Nor do we have any nonerary functions, like a CLOCK function that just spits out unit time wherever it is called. `if then else` would also be a useful addition.

5.2.4 Website

Right now when the program is executed it makes an IORef holding an empty spreadsheet, and this is its single source of truth for spreadsheets. There is no way for our program to function as a *website*, since it can only interact with a single frontend page. We need to have a single spreadsheet per session user, so that multiple clients could communicate with the server at the same time without tripping over each other's feet.

5.2.5 Ourselves

In many ways the project is immaterial, more importantly, we felt that if we had managed our time better, we could have gotten way more features in, and way more personal development in haskell, elm, graphs, and project work. Though this project will be shelved for now, we hope that our personal development will continue on, to new ventures :).