Experimentation project
Functional Programming in Spreadsheets

Supervisor: Atze Dijkstra

Rick Klomp
Student number: 5540232

December 27, 2015

1 Introduction

Popular spreadsheet programs (e.g. Excel, LibreOffice) provide a domain specific language to manipulate data. These DSLs are Turing complete (Hermans, 2013). Thus, they're in principal as powerful as any other programming language. However, many features known from languages such as Haskell are not provided by these languages. Examples of these features include (but are not limited to):

- Higher order functions
- Lazy evaluation
- List comprehension

This has raised the question if the experience of spreadsheet programming can be improved by providing a cleanly designed language that provides powerful mechanisms known from functional programming languages.

To initiate research in the area, this project has been performed to define an API that provides an interface for spreadsheet computations as well as to experiment with a basic application of the API. Results and findings of these topics are discussed in sections 3 and 4 respectively. Section 2 presents some related prior research that has been performed.

2 Prior work

There has been some research performed on the subject before.

Haxcel has resulted from a MSc thesis project:

> Haxcel is a spreadsheet-like interface to Haskell, where declerations can be edited and values of different variables monitord as declarations are updated. (Lisper, 2005)

Unfortunately, this project doesn't seem to have been succeeded.

Furthermore, Wakeling has performed research on embedding functional languages such as Haskell inside a standard spreadsheet such as Excel (Wakeling, 2007).

3 API

The API abstracts over four layers of spreadsheet computation functionality. Each layer is covered by one of the following type classes:

- Spreadsheet
- Cell
- Expr
- Var

These abstraction layers are further discussed in sections 3.1 through 3.4 respectively. Furthermore, section 3.5 discusses a feature considering visually highlighting references that is present in popular spreadsheet programs, but is probably not implementable in a sensible way using the current API.

3.1 Spreadsheet abstraction

```
class (MonadState s m, Var v, Expr e v rm, Cell c e v) =>
          Spreadsheet s c e v m rm | s -> c, s -> v, s -> m where
  updateEvals :: m ()
  getCell     :: Pos -> m (Maybe c)
  setCell     :: Pos -> c -> m ()
```

Figure 1: The Spreadsheet type class.

The definition of the Spreadsheet type class is given in figure 1. This layer functions as an encapsulation over the grid of Cells. The updateEvals function defines the interface for initiating a complete spreadsheet evaluation. With getCell and setCell it provides a way to access the next layer of abstraction (i.e. the Cell abstraction, see section 3.2). All functions depend on the state of the spreadsheet, thus they're run in a MonadState environment.

3.2 Cell abstraction

```
class (MonadReader (Map v e) m, Var v, Expr e v) =>
          Cell c e v m | c -> e, c -> v, v -> m, e -> m where
  evalCell      :: c -> m c
  parseCell     :: c -> c
  getEval       :: c -> Maybe e
  getText       :: c -> String
```

Figure 2: The Cell type class.

Figure 2 gives the definition of the Cell type class. The Cell abstraction enforces a state machine-like usage. Prior to calling evalCell, the Cell's content should succesfully have been parsed by the parseCell function. Any definitions of global variables that the cell's expression requires during evaluation should be supplied through the MonadReader environment. Similarly, prior to calling getEval, the Cell's expression should succesfully have been evaluated by the evalCell function. These functions don't return information showing if something failed, thus the state should somehow be stored and manipulated inside the Cell datatype (:: c). It might be useful to refine the types of these functions to reflect the state behaviour as well.

3.3 Expr abstraction

```
class (MonadReader (Map v e) m, Var v) =>
          Expr e v m | e -> v, v -> m, e -> m where
  evalExpr        :: e -> m e
```

Figure 3: The Expr type class.

In figure 3 the definition of the Expr type class is given. This layer of abstraction defines an interface to the spreadsheet language. All global variables that are required for evaluating an expression should be supplied through the MonadReader instance. It might be useful to refine the types of this function to reflect the state behaviour as well.

3.4 Var abstraction

```
class Var v where
```

Figure 4: The Var type class.

Figure 4 gives the definition of the Var type class. This layer is currently purely used to allow for different kind of variable encodings within languages. Perhaps this part of the API should be extended with functions once some kind of annotated text mechanism has been added (see section 3.5).

3.5 Annotated text

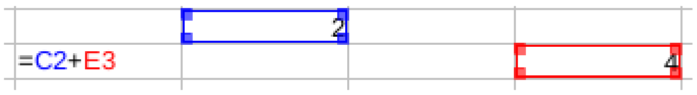


Figure 5: Visual highlighting of references in LibreOffice.

Currently, the API probably doesn't provide enough flexibility at some places to allow for adding features in a sensible way such as visually highlighting references (see figure 5). Information stored at the higher layers of the API (e.g. the state of the spreadsheet) isn't by default available at lower layers of the API (e.g. in the evalExpr function). It would most likely be wise to change the API in such a way that all information is provided through a single MonadState instance, and that all functions of the API run in this monadic environment.

4 API application

A basic spreadsheet program has been developed to experiment with the API. Section 4.1 discusses implementation specifics of this spreadsheet program.

The API abstracts over the backend matters that stay constant regardless of the implementation. It seems that some of the frontend matters can be abstracted over as well. In section 4.2 a mechanism is discussed that can probably be abstracted over.

There are arguably issues with popular spreadsheet programs considering other areas than the expresssions language. These are briefly discussed in section 4.3.

Section 4.4 quickly explains how the program can be used.

4.1 Experimentation API application

The expression language that is provided is Lambda calculus extended with three small features:

- Back and forth computation between Church numerals and Haskell numeral notation. The predefined function toInt translates a Church numeral to a Haskell numeral, and Haskell numerals are translated to Church numerals during parsetime
- Back and forth computation between Church lists and Haskell list notation. The predefined function toInt translates a Church list to a Haskell list, and Haskell lists are translated to Church lists during parsetime
- Cell reference variables (e.g. 1A, 5B)

All technical challenges have been solved in naive ways. For example, when a cell expression has been modified, the entire sheet is recomputed. If during this recomputation an evaluation differs from a prior evaluation, the change is pushed and the entire process is repeated (i.e. the entire sheet is again recomputed).

4.2 Frontend abstraction

All popular spreadsheet programs seem to have atleast two modes a user can be in, where one mode is for moving items (e.g. the cursor, cells) through the spreadsheet and the other is for editing items (e.g. cell content, figures). If it is indeed the case that for any possible spreadsheet program atleast two user modes are provided it is probably useful to abstract (some of) the mode changing behaviour to higher level mechanisms.

4.3 Spreadsheet miscellaneous issues

Dan Halbert seems to have a valid remark to current popular spreadsheet programs:

> I think hidden formulas is one of the big issues with spreadsheets. [...] The other big problem is the awkardness of assigning meaningful names to cells and blocks of cells. (Halbert, 2012)

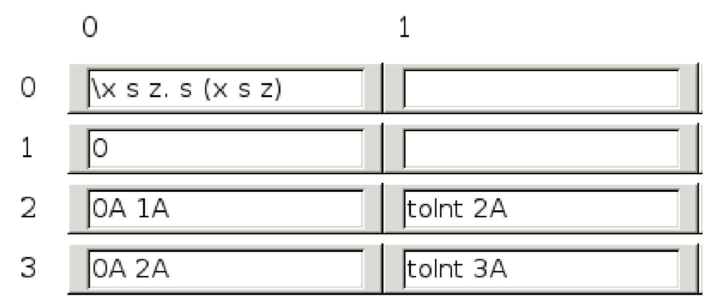
A solution for both of these issues might be to add a secondary environment to the spreadsheet program wherein regular code can be written. For example, toplevel definitions could behave like global variables, and can subsequently be referenced to in cell expressions.

4.4 User guide

The executable has two different modi that can be entered during startup: UI webserver hosting mode and lambda evaluation testing mode. To enter the latter, pass lambda code to the program through the stdin. The UI webserver hosting mode is entered if the stdin is closed during startup.

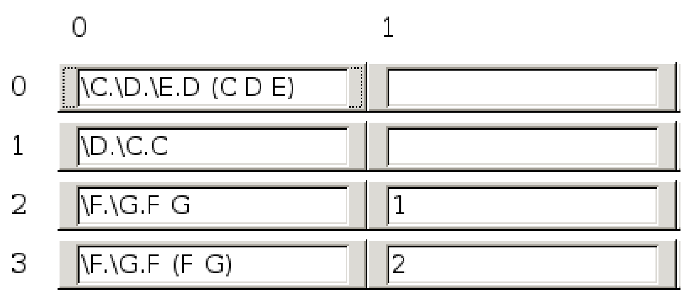
The webserver can be accessed by browsing to the link that is echoed by the webserver during startup (by default this is http://127.0.0.1:8023/). A grid of textfields is shown, the focus on textfields can be moved by using the arrow keys. Using the enter key, the contents of the focused textfield can be edited. For now only the extended lambda calculus code (see section 4.1) is allowed as content (no normal string content is allowed). The content will be parsed and evaluated when the editing mode is exited. To exit editing mode of a textfield, either press the enter key to accept the made changes or press the exit key to undo the made changes.

A small demonstration is given in figures 6 and 7. The first shows the code that has been entered in each of the cells, and the latter shows to what each cell's content evaluates to. A minor defect is that the column names are numbered, while they have to be referenced in lambda code by using capital letters (the A's in the references refer to the first column, thus 0A would be the (0, 0) cell).



	0	1
0	\x s z. s (x s z)	
1	0	
2	0A 1A	toInt 2A
3	0A 2A	toInt 3A

Figure 6: A spreadsheet showing unevaluated cell content.



	0	1
0	\C.\D.\E.D (C D E)	
1	\D.\C.C	
2	\F.\G.F G	1
3	\F.\G.F (F G)	2

Figure 7: A spreadsheet showing evaluated cell content.

5 Conclusion

An API has been defined that provides an abstraction over spreadsheet computation functionality. It most likely needs to be refined to allow some spreadsheet features to be defined in sensible ways.

Furthermore, the problem area in general has been researched. There has been some prior research performed. However, no extensive research has been performed, which in this case is probably needed in order to get a clear picture of the benefits and cost of functional programming spreadsheets.

Bibliography

Halbert, D. (2012, November). *Let's fix spreadsheets.*
Retrieved from http://www.lambda-the-ultimate.org/node/4626

Hermans, F. (2013, September). Excel Turing Machine [Web log post].
Retrieved from http://www.felienne.com/archives/2974

Wakeling, D. (2007). Spreadsheet functional programming.
Journal of Functional Programming, 17, pp 131-143