# Haxcel
# A spreadsheet interface to Haskell written in Java.

Johan Malmström
Master thesis
Department of Computer Science
Mälardalen University
jmm98002@idt.mdh.se

Supervisor: Björn Lisper

March 30, 2004

**Abstract**

The spreadsheet paradigm offers a fast interactive loop, where the effects of updates to definitions and data are immediately visible. This makes the paradigm attractive for program development, where redefinitions can be immediately tested and the results displayed. We have designed a simple, compiler-independent spreadsheet interface to Haskell, where cells host Haskell definitions. Spreadsheets are also used for high-level array calculations. In order to meet that demand we present an extended array library for Haskell, which provides a number of typical array-language facilities. Together, the interface and the array library provide an interactive environment that can be used both for development of general Haskell code and for array-oriented spreadsheet calculations.

# Contents

**1 Introduction** **1**
1.1 Definition . . . . . 1
1.2 Motivation . . . . . 1
1.3 Overview . . . . . 2
1.3.1 Haxcel . . . . . 2
1.3.2 Xarray . . . . . 2
1.3.3 Haskell . . . . . 3
1.4 Spreadsheets . . . . . 3
1.5 Road map . . . . . 3

**2 Spreadsheets** **4**
2.1 Traditional spreadsheet applications . . . . . 5
2.1.1 Excel . . . . . 5
2.1.2 Mesa . . . . . 7
2.1.3 Ragtime . . . . . 8
2.2 Problems with common spreadsheet applications? . . . . . 9
2.2.1 Range update problem. . . . . . 9
2.2.2 Hard to do simple data manipulations. . . . . . 10
2.2.3 Code reuse and function definitions. . . . . . 10
2.2.4 No local variables . . . . . 10
2.2.5 Limited number of data types . . . . . 11
2.2.6 No recursion in languages . . . . . 11

**3 Haskell** **12**
3.1 The functional programming language Haskell . . . . . 12
3.1.1 Compilers and interpreters . . . . . 12
3.1.2 Ordinary script and literate script . . . . . 13
3.2 hmake and hi . . . . . 14
3.3 Hugs and runhugs . . . . . 14
3.3.1 runhugs . . . . . 14

**4 XArray** **16**

**5 Haxcel** 19
- 5.1 The application Haxcel . . . 19
  - 5.1.1 User interface and editing . . . 20
  - 5.1.2 Description of some tasks . . . 21
  - 5.1.3 The Edit window . . . 22
  - 5.1.4 The Value window . . . 22
- 5.2 Design . . . 23
  - 5.2.1 Design goals . . . 24
  - 5.2.2 A more flexible indexing scheme . . . 25
  - 5.2.3 HAX Format . . . 26
  - 5.2.4 Program Generation . . . 26
  - 5.2.5 Execution of Main . . . 30
- 5.3 Implementation . . . 32
  - 5.3.1 Problems to solve . . . 32
  - 5.3.2 Compiling . . . 32
  - 5.3.3 Parsing structured data . . . 32
  - 5.3.4 Parsing numbers . . . 32
  - 5.3.5 Execution . . . 33
- 5.4 Implementation details . . . 33
- 5.5 Java . . . 34
- 5.6 XML . . . 34
- 5.7 Working environment . . . 34

**6 Related works** 36
- 6.1 Haskell related . . . 36
  - 6.1.1 Haskell editors . . . 36
  - 6.1.2 Haskell compilers an interpreters . . . 36
- 6.2 Spreadsheets . . . 37
- 6.3 Other approaches to spreadsheets . . . 37
  - 6.3.1 Logic and constrain based spreadsheets . . . 37
  - 6.3.2 Functional Spreadsheets . . . 38

**7 Examples** 39
- 7.1 An simple example . . . 39
- 7.2 An application for environmental science: Toxicity Ratings . . . 40
- 7.3 Budget for a small research department . . . 40

**8 Experimental results** 45

**9 Conclusions** 47
- 9.1 Future work . . . 47
  - 9.1.1 Better error handling . . . 48
  - 9.1.2 Editing in the spreadsheet-view . . . 48
  - 9.1.3 More GUI-functionality . . . 48

9.1.4 Literate scripts . . . . . . . . . . . . . . . . . . . . . . . . . . 48
9.1.5 Import of a existing Haskell module . . . . . . . . . . . . . . . . . . 49
9.1.6 Higher dimensions arrays and other structured data types . . . . . . 49

**A Defintions** **53**
A.1 Words . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

# List of Figures


2.1 An example of a spreadsheet window, . . . . . . . . . . . . . . . . . . . . . 4
2.2 An example of a common spreadsheet window . . . . . . . . . . . . . . . . 6
2.3 Circular reference warning. Microsoft Excel. . . . . . . . . . . . . . . . . . . 7
2.4 Interference between tables when inserting a row for “Charlie” in the first table. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

4.1 `meet` and `join` on array (matrix) bounds. . . . . . . . . . . . . . . . . . 16

5.1 The main window of Haxcel. . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.2 Example of specified options saved in the preference file . . . . . . . . . . . 22
5.3 The edit window. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.4 This is an example of the Array view of a Array type. . . . . . . . . . . . . 23
5.5 This is an example of the normal view of a Array type. . . . . . . . . . . . 23
5.6 The settings view of a value window. . . . . . . . . . . . . . . . . . . . . . 23
5.7 An example of the common index scheme. . . . . . . . . . . . . . . . . . . 25
5.8 Cell addressing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.9 Grammar of the HAX format. Described in Section 5.2.3 . . . . . . . . . . . 28
5.10 Program construction, case 1 . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.11 Program construction, case 2 . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.12 Grammar for analyzing the output from Main . . . . . . . . . . . . . . . . . 31
5.13 Example of an array. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.14 The FSM for parsing output from the execution . . . . . . . . . . . . . . . . 33

7.1 The windows associated with the first example Simpleex. . . . . . . . . . . 40
7.2 The reading illustrated in a graf. The calculated LD50 is marked. . . . . . . 42
7.3 Snapshot of the Haxcel screen for the budget spreadsheet. . . . . . . . . . . 43

# List of Tables


5.1 Version table of used software . . . . . . . . . . . . . . . . . . . . . . . . . 35

7.1 Test result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

8.1 Measurements for `Simpleex` and `Budget`, in seconds. . . . . . . . . . . . 46

# Listings

3 .1 Max2.hs . . . . . 13
3 .2 Max2.hs . . . . . 14
3 .3 Hello World . . . . . 15
4 .1 XArray declarations . . . . . 18
5 .1 Simpleex.hax . . . . . 27
7 .1 Simpleex.hs . . . . . 39
7 .2 Budget.hs . . . . . 41

# Acknowledgments

Master year in computer science at IDt, Mälardalen University.
Friends in the research group at IDt and fellow students through my studies [1].
Master year students especially Linus, Conny, Ezra, Daniel
Apple Computer for making reliable and stable hardware and software.
Björn Lisper for supervising, inspiration and encouragements.
Friends and family who has supported my education.

[1] the pizza group for mention some...

# Chapter 1

# Introduction

This is the report for my Magisterexamen[1] project. The project was supervised by professor Björn Lisper who also is the head of the research group that I have had the pleasure of being a part of during my master year in computer science[2], at the Department of Computer Science(IDt) at Mälardalen University.

The project had no external funding but was supported with office space, equipment and tools by IDt.

## 1.1 Definition

The project was to design and implement a spreadsheet interface to the functional programming (FP) language Haskell [PH99]. The project had two main purposes:

1. Provide a environment with a strong and immediate feedback to Haskell programmers.

2. Provide a sound and potentially very powerful spreadsheet language to spreadsheet programmers.

The purpose of this interface was to investigate if Haskell in a spreadsheet environment is a good idea. In discussions we made the following demands for the software: the software should be working on multiple platforms, and the software should be independent of Haskell implementations.

## 1.2 Motivation

Spreadsheets are widely used in day to day business and often provide a efficient way to do calculations. A spreadsheet is a set of cells which are referenced by it's position in a row-column matrix. Spreadsheets were used before the computer era with extensive manual

[1] The Swedish equivalent of a master degree.
[2] see http://www.idt.mdh.se/magister/ for more information on the master year

recalculations but has now become one of the most used computer applications in both science and administration.

The spreadsheet has proven to be a very powerful environment to visualize computed data and the immediate feedback makes the spreadsheet paradigm attractive for program development. However, the current spreadsheet implementations available on the market has many drawbacks when it comes to computing. We want to provide a powerful visualization tool for the spreadsheet- and Haskell programmer. Common spreadsheets solve many problems and it's well-known that modern spreadsheets are well suited for ad-hoc business decisions making and financial modeling. Nevertheless has the programming languages community, historically, been very cold opposite the spreadsheet metaphor[YC94]. This was probably the result of the flaws in early implementations of spreadsheet applications. More recent work has shown that the paradigm is becoming a interesting field of research, and we want to show another approach to how to work with spreadsheets as a visual aid. See section 2.2 for our encountered disadvantages in common spreadsheet applications.

A spreadsheet is a good example of simple functional programming. In a spreadsheet the focus is on *what* is to be computed, not *how* it should be computed. A cell's value is specified or computed without the need of specifying for example in which order the cells should be updated. But why are the programming languages and environments in common spreadsheet applications so clumsy? The question has been raised by others before us. We have an idea that taking the advantages of a lazy, poly morphic language could improve the spreadsheet paradigm.

## 1.3 Overview

The project consists of three parts, namely the java application *Haxcel*, the Haskell library `Xarray` and a Haskell compiler.

### 1.3.1 Haxcel

*Haxcel* is the name of the prototype that we have designed and implemented in this project. The interface makes it possible to create or update Haskell declarations and immediately view the result.

### 1.3.2 Xarray

We present a extension to the Array library, Xarray, that have features otherwise found in languages like Fortran 90. The library can be used independently of the *Haxcel* interface.

### 1.3.3 Haskell

Haskell [PH99] is a general purpose, purely functional programming language. A functionall programming language provides a simple programming model in that there is one value computed, the result, dependant of other values, the inputs. We think Haskell can provide an powerful programming language to spreadsheet programmers that needs more functionality than what is provided by common spreadsheet software. We chose Haskell since that it is a strong language in general, and since it is relatively easy to define domain-specific sublanguages within it.

## 1.4 Spreadsheets

Spreadsheet systems are very popular. They are used for small calculations, but also for serious application programming in areas like science and administration, when there is a need to perform calculations where fast and visually comprehensible feedback is more important than computational speed. A big reason for the popularity is the very tight definition-eval-display loop, where the effects of a redefinition of an entity are immediately seen. This makes the spreadsheet paradigm attractive, compared with traditional compiled languages where the compile-eval-print loop is considerably more tedious and where results are harder to present in a way that a human can grasp quickly.

Spreadsheets support an array-oriented style of computing. Sections of spreadsheets can be selected, and collective operations like summation can be applied to these. This makes it convenient to express many calculations. Without doubt, part of the popularity of spreadsheets can be attributed to this. The spreadsheet languages are also typically without side-effects. This provides a simple and intuitive evaluation model.

However, the common spreadsheet systems have weaknesses. The languages are typically very restricted: for instance, recursion is invariably not allowed, and only a few basic data types are usually supported. This and other common weaknesses are discussed in Section 2.2.

## 1.5 Road map

This report is outlined as follows: In Chapter 2 we give a short introduction to the spreadsheet paradigm.In Chapter 3 we introduce the FP language Haskell and give references to further reading on this subject. The library Xarray is presented in Chapter 4. In Chapter 5 we describe the design and implementation of the project. The chapters that follows is (Chapters 6, 7, 8) related work, examples and experimental results The report is finished by a conclusion and a section of future work in Chapter 9.

This report is typeset in LaTeX using TeXShop[Alg01]. Graphs and diagrams is created in OmniGraffle [Omn02].

# Chapter 2

# Spreadsheets

A spreadsheet is typically a 2-D array of cells used to visualize a set of interrelated values. A typical example of a spreadsheet is illustrated in Figure 2.1. Each cell is addressed by a 2-D coordinate and its value is represented either by a formula or a constant. If a formula is used it's defined by referring to other cells or built-in functions for getting values from external sources like data bases or other spreadsheets. In Figure 2.1 cell E4 is marked and the value is visible in the edit field in the top. In this case the value of the cell is a formula defined with references to the values in cell D4 and D2.

By doing these definitions and declarations in a spreadsheet the user has constructed a small application although the common user of spreadsheets seldom refer to the spreadsheets as applications. This may be one of the factors behind the success of the spreadsheet paradigm, the user don't need traditional programming skills to start building applications in a spreadsheet.

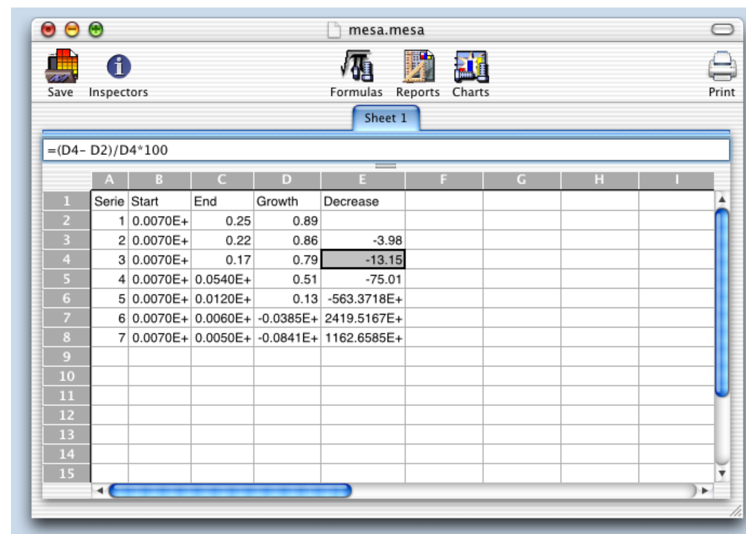


Figure 2.1: An example of a spreadsheet window,

The spreadsheet era for computers dates back to May 1979 when Dan Bricklin and Bob Frankston introduced the application *VisiCalc* [BF99] for Apple II and changed the way to do calculations on a desktop computer. Before spreadsheet applications were introduced, manual spreadsheets were common in calculation but required massive manual recalculations when some data were changed. Today the *de facto* standard in spreadsheet applications is Microsoft's Excel, hereafter referred to as Excel. Excel is of course much more competent and has more functionality than its ancestor VisiCalc, but is basically in the same paradigm where the spreadsheet represents a 2D area were you can store your data and have calculations made based on formulas. The paradigm has a rigid indexing scheme, where rows are indexed with letters and columns with numbers. See Section 5.2.2 for a discussion on index schemes.

Within this paradigm there have been and are many different spreadsheet applications that together with Excel define a standard functionality and how to use spreadsheets. To supplement Excel we have specifically looked some other spreadsheets program, they are Mesa from P & L Software [P&L01] a spreadsheet program, RagTime Solo from RagTime GmbH [AL01], a program that combines word processing and spreadsheet work.

## 2.1 Traditional spreadsheet applications

The common knowledge of spreadsheets over the years can be tracked as what has been the generic "name" for this paradigm. Starting with Visicalc, this domain of applications has been called "visicalc", "lotus" after Lotus spreadsheet application Lotus 1-2-3 and now "excel" after Microsoft's application.

In an article in the January/February 2002 issue of Scientific Computing World F. Grant [Gra02] looks beyond Excel to test what alternatives there are. Grant dissect Corel's Quattro Pro 10 [QUA] and Insightsful's S-Plus 6 [SPL] among others. We have compared Excel with Mesa from P & L Software [P&L01] a spreadsheet program, RagTime Solo from RagTime GmbH [AL01], a program that combines word processing and spreadsheet work. All this different software share some common set backs, for instance they do a good job as data containers, simple organizers, visualizers but if you want to do more heavy analytics of data they are not strong enough. Grant writes in [Gra02] "Excels statistical functions, for example, work well in their natural office habitat but are not the most robust or reliable when pushed to the limit".

### 2.1.1 Excel

Excel is the dragon of the market. Microsoft has managed to make this to a standard application, and every other application on the spreadsheet market has to be somewhat compatible with Excel, and is judged by Excel. Excel was first released in 1985 for Apple Macintosh and was one of the first spreadsheet applications to offer a graphical user interface.

In Excel you do your work in spreadsheets or worksheets, as Microsoft prefers to

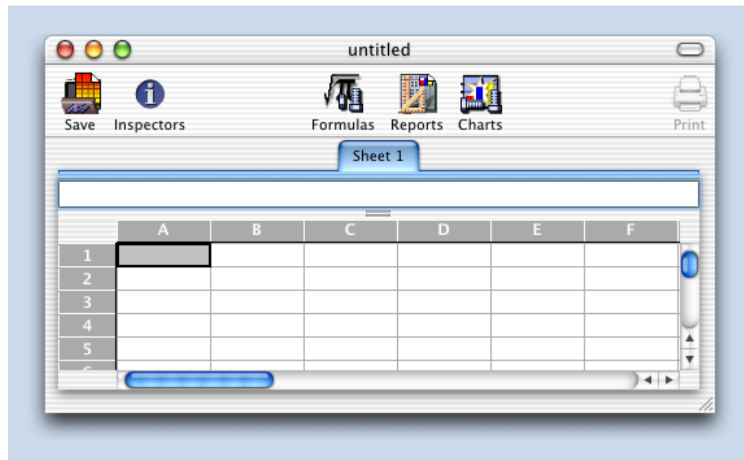


Figure 2.2: An example of a common spreadsheet window

say, organized in workbooks. Worksheets consists of a table of cells, where the rows of cells is enumerated from 1 and the columns is indexed by letters, starting with A. This style of referencing cells, *A1*-system, was first introduced by Lotus 1-2-3 and is now the standard way. Earlier spreadsheets often addressed cells by a *R1C1*-system. As described above a cell can either hold a formula or a constant value. In either cases it's the value that is shown in the cell. A cells value can be displayed in different ways but there is no type checking. For example the integer *123456* is display as *2241-01-04* if viewed as date-format and *00.00* if the cell is formated to show time-format. If a value of a cell begins with $=$ then Excel regards the value as a formula and tries to evaluate the formula. If the formula is correct the evaluated value is shown, otherwise a error text is written in the cell. Excel distinguish between the following cell contents: numbers, text, logical value, formula, error value and matrix.

A workbook contains of a collection of worksheets and graphs saved in the same file.

### Formulas in Excel

Formulas in Excel can solve mathematical calculations, compare values or concatenate strings. Formulas consists of a text-string where there result is an assignment. A formula begins with a assign character, $=$, followed by operands that forms an equation. The operands are separated by calculation operands. There a four different types of calculation operands,

1. Mathematical operands, for instance $+, -, /.*$.

2. Logical operands.

3. Text operand, the &-sign concatenates text.

4. Reference operator, can be used for referencing cells or cell ranges. Besides having references to cells in the same worksheet it's also possible to have references to cells in other worksheets and to cells in other workbooks.

For instance the `=E6` assigns the value of cell `E6` in the current worksheet to the current cell. The value is calculated either immediately or when a recalculate command is executed. Excel provides a large set of predefined functions that helps building formulas but one can not define own functions to be included in formulas. The predefined formulas provides for example statistical functions and logical functions.

**Macros in Excel**

Excel provides a macro language where one can define functionality to be executed when triggered by user controlled events for example pressing a keyboard combination. The language is *Visual Basic for Applications*. Macros helps the user to do sequences of commands that are repeated often in a more easy way. The macros are stored in Visual Basic modules. Macros can not be included in formulas, and should be regarded as procedures more than functions.

**Circular references**

A circular reference is when one create a relationship between two cells with formulas that depend on the other cell's value. For example if cell $C6$ has the formula $= E6 * 2$ as value and $E6$ has the value $= C6 + 5$ one has created a circular reference. Excel can not evaluate cells that have circular references and warns the user by showing a dialog, see Figure 2.3 and also have a "Circular References toolbar" that helps one to remove circular references.

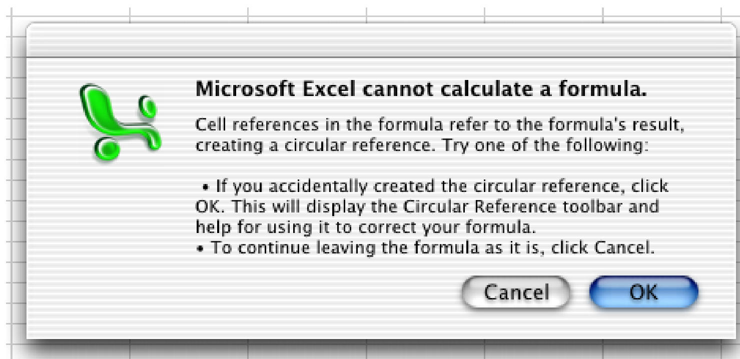


Figure 2.3: Circular reference warning. Microsoft Excel.

### 2.1.2 Mesa

P&L Software has a spreadsheet application called *Mesa* [**?**, P&L01]. Although Mesa is a traditional spreadsheet program, available for Apple's Mac OS X, it makes some things

different from Excel. I particular Mesa uses a Mac OS X convention called Inspectors to achieve many results. An Inspector let you view and change properties like size, fonts and formats of the current element, for example a cell, a chart or a report.

Mesa can work with spreadsheets with up to 99999 rows (indexes from 1 to 99999) and columns labeled from A to XFDAA. Each cell has a value and formatting information. Mesa has three data types: numbers, strings and error. The value of a cell is either a numerical- or a string constant, or a formula. A formula can change some formatting information on a cell depending of the value of the formula. A formula can refer to a value in a cell during calculation. When writing formulas Mesa let one chose between Excel style or Lotus style formula.

A functionality that Mesa does not provide, but Excel does, is for example 3D pie charts and pivot tables.

#### Code reuse

Mesa has provided two functions to enhance code reuse, `NEXT()` and `SAME()`. By using them you only need to change a formula used in many cells in the original cell instead of using the "copy+paste"-method. For example if you want the same formula as in the cell $B2$ one write $= SAME(\mathrm{B2})$. The problem with code reuse is more generally discussed in Section 2.2.3

#### Cell Addresses

Mesa uses the *de facto* standard $A1$-system for referencing cells.

Some Mesa functions take a cell address as their argument. In most Mesa functions, when one enters the address of a cell as an argument, the function uses the contents of that cell as the argument. In a few functions, however, the function is looking for information about the cell itself. This is the case for functions that ask specifically for a cell address as an argument. An example of a function that takes a cell address as its argument is the function `ISEMPTY`, which checks to see if the cell at a given address is empty. For example, $= ISEMPTY(U2)$ returns 0 if cell $U2$ has any contents or 1 otherwise.

#### Naming ranges

In Mesa it's possible to name a range of cells, for instance if one names the cells in the range from cell $A1$ to cell $F10$ `Range1` then the two formulas $= SUM(A1 : F10)$ and $= SUM(Range1)$ would give the same result. This gives the user an option of more suitable names on data ranges when building formulas. The naming of ranges is done with the Label Inspector.

### 2.1.3 Ragtime

Ragtime [**?**, AL01], from RagTime GmbH, is odd child of in the spreadsheet family, combining a word processor and spreadsheets. You could say that this application focuses on

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | | Salary | Other $ | | | | |
| 2 | Alan | 17000 | 9 | | | speed | strength |
| 3 | Bob | 44000 | 99 | | THX99 | 47 | 95 |
| 4 | Dave | 23000 | 999 | | AGP | 512 | 173 |
| 5 | | | | | MPH | 70 | 0 |

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | | Salary | Other $ | | | | |
| 2 | Alan | 17000 | 9 | | | speed | strength |
| 3 | Bob | 44000 | 99 | | THX99 | 47 | 95 |
| 4 | Charlie | 19000 | 9999 | | | | |
| 5 | Dave | 23000 | 999 | | AGP | 512 | 173 |
| 6 | | | | | MPH | 70 | 0 |

Figure 2.4: Interference between tables when inserting a row for “Charlie” in the first table.

the printed result. The functionality of the spreadsheet part is of the common type, but there is functionality we wish to mention as good examples.

**Unions** are single large cells created from ranges of cells. It’s useful when creating larger containers for example pictures.

**3D planes** In Ragtime a spreadsheet is defined as a plane with rows and columns, so what they have done is enabled multiple planes in a spreadsheet. It’s possible to do selections in a 3D range of stacked spreadsheets.

## 2.2 Problems with common spreadsheet applications?

In our work with spreadsheets we have encountered problems with common spreadsheets program that makes them more or less clumsy as application developing environments. We here present encountered problems.

### 2.2.1 Range update problem.

A problem is that the “tabular” and “definitional” uses of coordinates sometimes clash. “Tabular” use of spreadsheets can, for instance, require that rows or column are inserted in order to update a table that is embedded somewhere in the spreadsheet area. This will offset a number of definitions, so they now are associated with new coordinates. Even though most spreadsheets can update references to coordinates in definitions automatically, to compensate for this, it provides a burden for the programmer to keep track of the changing names of defined entities. Some entities like ranges are typically not updated and operation on a range will be performed over wrong range after the update. Worse still is that insertion or deletion of rows or columns, in order to update one embedded table, can interfere with other tables or definitions that just happen to share the same rows or columns, see Fig. 2.4. In such a situation, a tedious re-editing of the spreadsheet can be necessary.

### 2.2.2 Hard to do simple data manipulations.

It's hard to implement manipulations over sequences or data ranges. Formulas for such manipulations often require formulas which involve relative addressing, or writing spreadsheet macros.

### 2.2.3 Code reuse and function definitions.

Formulas can seldom be reused besides by copying the formula into different cells. It reduces integrity because it's difficult to tell when a formula has been overwritten with a value. Another problem is that coping formulas into cells hinders the development of spreadsheet applications because it is difficult to extend existing applications.

Another, related problem, observed also in [CB97, dHRvE95], occurs when spreadsheet definitions are copied between cells. Very often, the copying is done to implement a similar computation on, say, a different row of a table than the original one. A convenient treatment of coordinates in the definition is then to let them "move", so they have the same positions relative to the new coordinate of the copied definition. Therefore, spreadsheets tend to have this behavior as default. But sometimes one would like to keep coordinates "absolute", so they don't move but refer to the same entity before and after copying. Spreadsheet systems like Excel provide a syntax that lets the programmer override the default behavior, but the ambiguity of "copying semantics" is still a source of many bugs in spreadsheet programming.

A solution to the problems mentioned above is to create a clear separation between the use of spreadsheets to define entities and promptly visualize the results, and the use of spreadsheets for tabular calculations. Cells can be given general names rather than just coordinates, and one can have a separate cell for each name binding. Arrays can be used for tabular data. A change to the index range for one array, for instance to add or delete a row or column, will then not affect other arrays. If the formula language is sufficiently rich in array computing primitives, for instance to specify a similar computation being repeated over a range of array sections, then there will be less need to copy and paste definitions. Haxcel was designed to provide a solution of this kind.

### 2.2.4 No local variables

The lack of local variables in formulas can lead to unwieldy constructs. In normal spreadsheet work the work-around for this is to put temporary values in the global cell space and apply "not-visible" or hide formats to this cell. This leads to spreadsheets with bad layout and unsafe behavior. One could argue that this could lead to uncontrolled memory use but spreadsheet has static formulas with a static number of variables so this is not true. On the other hand a language without local variables can be very limiting when constructing more complex algorithms and spreadsheet languages with local variables could lead to safer spreadsheets.

### 2.2.5 Limited number of data types

Common spreadsheets has a very limited set of data types. For instance RagTime, presented in Section 2.1.3, has seven different data types: empty, number, text, multi-line text, date, time-span and error code. This basic data types restrains the ability to implement more complex structures. By using a language with possibilities of a more complex data type one could use spreadsheets for visualization of for instance trees or lists. Spreadsheets could also benefit from a more type safe language.

### 2.2.6 No recursion in languages

A language where it's impossible to create an algorithm witch includes any loop-construct, for instance recursion is classified as not Turing-complete. This is a restraint that excludes many algorithms. For instance after collecting values from an experiment it could be convenient to use iterative algorithm for estimating results.

# Chapter 3

# Haskell

Haskell is a polymorphic typed, lazy, purely functional programming language. The work on Haskell started in the late 80s and the current version is described in the *Haskell Report* [PH99] commonly known as *Haskell 98*. The language is named after Haskell B. Curry who was one of the pioneers of $\lambda$ -calculus.

## 3.1 The functional programming language Haskell

In a functional programming language the programmer concentrate on the relationships between values and functions are the central component for accomplish values. Values are grouped together in types, such as numbers. The base types in Haskell is `Int` and `Float` for numbers, `Bool` for booleans and characters, `Char`. From this base of types, types can be built as *tuple-*, *list-* and *function*-types. Haskell also provides the ability to define new types using *algebraic* and *abstract* type mechanism. type classes, and type class instances. Haskell is a polymorphic typed language. Polymorphism means many forms and combined with Haskell as a strongly typed language this enhances the re-usability of functions. A type is polymorphic if it contains type variables. One important feature in Haskell is *higher order functions*. This means that functions can be a parameter to another function, returned by functions or stored in data structures. Haskell evaluates expressions with *lazy evaluation*, which imply that only parts of structures which are needed will be examined. A functional program, a *script*, is made up of a series of definitions of functions. A Haskell definition is a name associated with a value of a given type. Definitions is grouped into *modules*.

### 3.1.1 Compilers and interpreters

There are four different Haskell compilers/interpreters (ghc, nhc, hbc and Hugs) that all are fruits of research of their creators, and they all have flavors that is suited for different things. We didn't want to restrict our software to just one of all these flavors but wanted to provide a tool that can be used with different compilers. This is a trade-off to what we

```
module Max2 where

-- max2 returns the largest of two integers
max2 :: Int -> Int -> Int
max2 x y
        = if x >= y then x else y
```

Listing 3.1: Example of a simple Haskell module.

could have done if we had been working tightly to only one existing compiler or doing our own compiler.

As mentioned above there are four different Haskell compilers:

**ghc** This compiler, Glasgow Haskell Compiler, written in Haskell is a full implementation of Haskell 98. There is an interactive environment called GHCi and also a number of experimental languages extensions for example concurrency and exceptions. The team behind GHC has two main targets for GHC, firstly applications writers where GHC provides an platform for software development. Secondly implementors of Haskell related tools where GHC can provide an environment for different computer science research projects. GHC is developed at the University of Glasgow.

**nhc** or nhc98 aims to produce small executables. nhc98 provides some debugging abilities that is not provided by any other Haskell compiler. The nhc98 compiler is distributed by York Functional Programming Group and they have also implemented *hi* an interactive development environment for Haskell and *hmake* a batch compilation tool. This tools are described in depth in section 3.2.

**hbc** and hbi is based on a LML compiler. This compiler and interpreter supports Haskell 98, but the system seems to languish and it's web pages has not been updated for some time. The hbc/hbi system is developed at Chalmers Technical University.

**Hugs** is a Haskell interpreter written in C. It has fast compilation and combined with the convenient interpreter provides a good development system. Hugs 98 is available for all Unix platforms and Windows. Hugs is described more in section 3.3.

### 3.1.2 Ordinary script and literate script

In Haskell one define scripts with *declarations* in *modules*. A simple module `Max2` with one declaration of a function `max2` is shown in Listing 3.1.

In Haskell its possible to write two different styles of scripts. The first type is the "ordinary" type, which Listing 7 is sample of 3.2. The other kind is called *literate script*. In a ordinary script everything in the file is interpreted as program text, except where it is explicitly indicated that something is a comment. In a literate script it's the other way around, everything in the file is commentary text if not specified explicitly as program text.

```
This is an example of a literate script.

> module Max2 where

max2 returns the largest of two integers

> max2 :: Int -> Int -> Int
> max2 x y
>        = if x >= y then x else y
```

Listing 3.2: Example of a Literate script.

Program text is specified with a `'-'` and the script file is stored in a `'.lhs'` file. The simple module `Max2` is shown in literate script version in Listing 3.2.

We have chosen not implemented functionality for literate scripts in Haxcel, the motivation is that we wanted to focus on the main functionality of spreadsheets. We discuss literate scripts further in the future work section on page 48.

## 3.2 hmake and hi

One of the most used applications in Unix program development is the utility program make, were one defines for examples compile dependencies of source files and make takes care of recompiling when necessary. *hmake* provides this functionality to the Haskell programmer. *hmake* gives us control of different modules, compiled or not. By specifying a *import*-declaration, hmake makes sure that it's the latest version is being used, and this is the choice made in transforming a module to working program, see 5.2.4.

The implementation of *hi — hmake interactive* as described in [Wal00] by Wallace has been a source for thought on making a interactive environment to a non-interactive compiler.

## 3.3 Hugs and runhugs

The above listed environments is compilers, Hugs is instead a interpreter. Hugs is probably the most used Haskell environment, and is available for almost every computer platform. It has a very fast compiler and has features like incremental compilation. The trade off made is that Hugs run-time performance can't compete with real compilers like GHC and HBC.

### 3.3.1 runhugs

The command *runhugs* provides the possibility to make stand alone programs that uses Hugs. It is there for possible to write Unix-scripts the will be interpreted by runhugs. For instance a "Hello World" program could look like in Listing 3.3. runhugs has shown to

```
#!/usr/local/bin/runhugs +l

> module Main where
> main = putStr "Hello,␣World\n"
```

Listing 3.3: Example of a shell script using runhugs.

have very short turn-around-time, the time spent from running the command to showing the result.

# Chapter 4

# XArray

THIS TEXT IS NOT FINAL

Lisper describes in [LM02] an extension to the Array library.

`Xarray`is an array library for Haskell 98 that can be used independently from Haxcel. It provides some array operations normally found in array languages like Fortran 90. Besides being backwards compatible with Haskell's standard arrays, a number of features are supported: "elementwise intrinsics" overloading for arithmetic operations, automatic promotion of numerical constants into arrays of appropriate size, a number of array projection operations and folds, and some function for composing matrices out of lists of vectors and vice versa.

The extended arrays provided by the `Xarray`library is are essentially a simplified version of the *data fields* [Lis98] of Data Field Haskell. In Listing 28 we list the major new operations and type declarations.

The extended array type `Array a b` is defined as an instance of `Num` and `Fractional` when `b` is an instance of `Num` and `Fractional` respectively.

Basic features of Xarray:

- Ordinary Haskell array or infinite array. An infinte array is an function together with a particular bound `Universe`.

- Computing bounds of arrays with functions `join` and `meet`, illustrated in Figure 4.1.

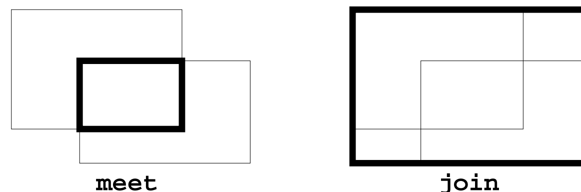


Figure 4.1: `meet` and `join` on array (matrix) bounds.

- Scalable arrays is provided through the type of the extended array.
- Operations over sparse arrays is solved by defining `Maybe a` as an instance if `Num` and `Fractional` when a is an instance of the corresponding class.
- Elementwise conditional `ifA` **FÖRLKLARA MER**
- Restriction operations `at` and `when`
- projection operations
- Dimensions operations, makes matrices out of vectors and splits matrices into vectors
- fold opratios over arrays

```
data (Ix a) => Array a b = Arr (A.Array a b) | Inf (a -> b)
                           deriving ()
data Bounds a = B (a,a) | Universe deriving Eq

meet :: Pord a => Bounds a -> Bounds a -> Bounds a
join :: Pord a => Bounds a -> Bounds a -> Bounds a
inBounds :: Ix a => Bounds a -> a -> Bool
mkarray :: Ix a => Bounds a -> (a -> b) -> Array a b
finarray :: Ix a => (a,a) -> (a -> b) -> Array a b
infarray :: Ix a => (a -> b) -> Array a b
at :: (Ix a, Pord a) => Array a b -> (a,a) -> Array a b
when :: Ix a => (a -> Bool) -> Array a b -> b -> Array a b
proj_xx :: (Ix a, Ix b) => Array (b,a) c -> b -> a -> c
proj_1x :: (Ix a, Ix b) => Array (b,a) c -> a -> Array b c
proj_21 :: (Ix a, Ix b) => Array (b,a) c -> Array (a,b) c
...(etc)...
ifA :: (Ix a, Pord a) => (a -> Bool) -> Array a b -> Array a b
                                          -> b -> Array a b
cols2matrix :: Ix a => [Array a b] -> Array (a,Int) b
rows2matrix :: Ix a => [Array a b] -> Array (Int,a) b
matrix2rows :: (Ix a, Ix b) => Array (b,a) c -> [Array a c]
matrix2cols :: (Ix a, Ix b) => Array (b,a) c -> [Array b c]
foldlA :: Ix a => (b -> c -> b) -> b -> Array a c -> b
foldlM :: Ix a => (b -> c -> b) -> b -> Array a (Maybe c) -> b
...(etc)...
```

Listing 4.1: Selected declarations and defined functions of the extended array library. (Haskell's usual `Array` module is imported qualified as `A`. `Pord` is a partial order class with operations `glb`, `lub`, and `lt` used by `join` and `meet`.)

# Chapter 5

# Haxcel

This chapter is structured as follows. First we make a walk through the software showing features and introducing some thoughts. Next we present the design of the software and finally writes about the implementation.

## 5.1 The application Haxcel

We made the choice to use to get a platform independent application. We have focused on to make the software a effective prototype for investigating if Haskell could be used in a spreadsheet environment and it should be easy to use. The software is distributed as a -jar file, which make the software easy to install. We do not provide a Haskell back-end, see Chapter 3 for references.

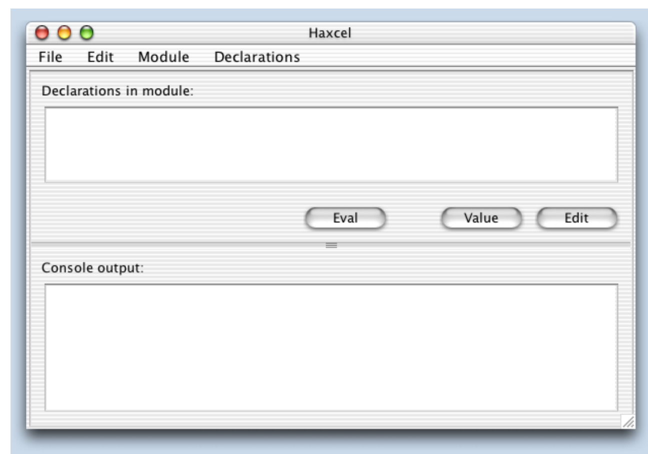


Figure 5.1: The main window of Haxcel.

### 5.1.1 User interface and editing

Haxcel has a main window that represent a Haskell module, see Figure 5.1. The main window contains a menubar, two fields and three buttons. The upper field contains a list of the declarations in the current module. This list is of course empty from start. The lower field is a console area were error messages is displayed. For instance if there is a compile error the software will notify the user by a dialog and the user will find more information on the error on the console. The three buttons are *Eval*, *Value* and *Edit*. The Eval-button starts the eval-loop described below. When the evaluation is done one can use the Value-button to show the value of the declaration. All editing of declarations is done in the Edit-window, displayed when one click the Edit-button. The Edit- and the Value-button requires a selected declaration to be highlighted, and the Eval-button requires that there is an active module.

#### The menus

The menubar has four menus: *File*, *Edit*, *Module* and *Declarations*. File and Edit is de facto standard to have in a graphical user interface and one can here find the usual menu items like New, Open, Close etc. We have not implemented any menu driven cut and paste but relies on the keyboard equivalents that is supplied with Swing. They shortcuts is platform dependent, for instance copy is `Command`-C on Mac OS X and `ctrl`-c on other platforms, so a user can take comfort in previous knowledge.

The choices of on the *New*-menu is as follows:

**New** Creates a new module. Brings up a dialog for entering the name on the module to create. One does not suffix the module name with `.hs` or `.lhs`. Menu shortcut is `ctrl`[1]-N.

**Open** Brings up open-file dialog for the user to locate and open a previously saved module. Menu shortcut is `ctrl`-O.

**Close** Closes the current module. Active only if there is an opened module. Menu shortcut is `ctrl`-W.

**Save** Used for saving the current module in HAX-format. Menu shortcut is `ctrl`-S.

**Save As...** When one wants to save the current for the first time or want to save it as another module.

**Export** For exporting the current module as an Haskell-file. See Section 5.1.2 below.

**(Import)** Not implemented. See future work in Section 9.1.5.

**Quit** For a controlled termination of the software. Menu shortcut is `ctrl`-Q.

[1]Mac OS X users uses the `Command`-key instead, but from here on we, to simplify reading, only write `ctrl`

Usually one in the *Edit*-menu find items like items Undo, Copy, Cut and Paste. We have chosen to not include them in this prototype, see the future work section (Section 9.1). The choices of on the *Edit*-menu is as follows:

**Select compiler** Here one chose to use hmake or runhugs as Haskell back-end.

**Compiler flags...** Brings up dialog where on specify options to the compiler. See Section 5.1.2 below.

The choices of on the *Module*-menu is **Compile** and **Eval**. Eval is the equivalent of the button Eval, and Compile can be used for syntax checking when *hmake* is chosen as back-end.

The choices of on the *Declarations*-menu is as follows:

**New declaration** The submenu contains all the type of declarations that can be created in Haxcel. When one of them is selected a dialog for name the new declaration is displayed. After the name has been specified the edit-window for the new declaration opens.

**Rename** To rename the declaration currently selected in the declaration list.

**Delete** Deletes the selected declaration, see Section 5.1.2.

### 5.1.2 Description of some tasks

In this section we give a more in depth explanation of some special features in *Haxcel*.

#### Changing preferences

Haxcel makes its easy for the user to change the Haskell compiler or compiling options to be used. In the Edit menu there is a choice for preferences and the only customization available is changing the compile command. The syntax for the options is the syntax for hmake (see Section 3.2). The default option is *-nhc98* which specifies that the nhc98 compiler is used.

The specified options is saved in the preference file for Haxcel. This file is saved in two different way according to which operating system being used. If the software is running on Mac OS X the file *"Haxcel Prefs"* is saved in the users preference folder ($< user >$/Library/Preferences). On any other Unix system the file *".haxcelprefs"* is saved in the users home directory. The file is a plain text file with the specified options. It's possible to edit the file in a text editor. An example of a preference file is in Figure 5.2 show how to specify the nhc98-compiler and that module dependencies should be printed. If the options added to the compile command generates output it's written in the console area in the main window of the software. See Section 5.3.2 for further information on compiling.

Further preferences would demand a more complex file format.

```
-nhc98 -M
```

Figure 5.2: Example of specified options saved in the preference file

#### Delete and rename of a declaration

It's possible remove a declaration and it's also possible to rename a declaration. When renaming a declaration it's important to rename the declaration in the Haskell code. This is a bit awkward but we have a solution for this in the future work section, see page 48.

#### Export of a module

It's possible to export a complete module from Haxcel to a common Haskell file that can be used independently from Haxcel. The user is free to choose name for the exported file and where to save the file. It's not possible to import a Haskell module to Haxcel, although this would be a nice thing, instead we have kept the syntax of a Haxcel file fairly simple and it's possible to convert a Haskell file to the hax format by hand. For further information on the hax format see Section 5.2.3.

### 5.1.3 The Edit window

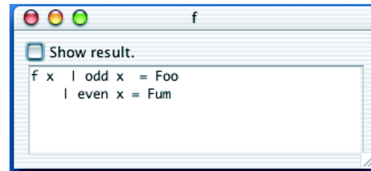


Figure 5.3: The edit window.

The edit window consists simply of a text field for the Haskell source and a checkbox. The checkbox has the important function of telling *Haxcel* whether to show the value for this declaration or not. An example is in Figure 5.3. The position and dimension of the window is saved between sessions in the module file.

### 5.1.4 The Value window

The figures 5.4 and 5.5 shows the two different ways of showing a computed value of a declaration when the result is of a array data type. One can change the display type by choosing in the settings view of the value window, as displayed in Figure 5.6. The dimmed radiobuttons for changing between horizontal or vertical view of a array is not implemented, see future work in Section 9.1.3. The contents in a cell is formated automatically by the users operating system settings. This could be somewhat confusing to experienced Haskell users

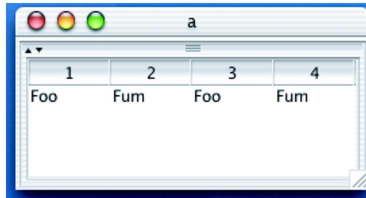


Figure 5.4: This is an example of the Array view of a Array type.

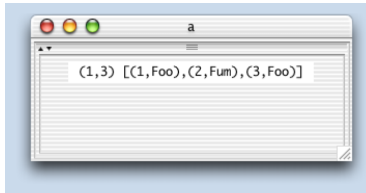


Figure 5.5: This is an example of the normal view of a Array type.

that expect to see numbers as plain digits. Today *Haxcel* treats *integers*, *floats* and *strings* specifically. Any other kind of data type that has derived the Haskell class Show is treated as a string.

When a computed value is not a array the value window is slightly different, showing just the value and there is no settings view for this kind of data.

When saving a module the specifications of the value windows is saved to.

## 5.2 Design

The task of the project had a very soft description, and as mentioned in the Definition on page 1 the main task could be summarized to investigate whether Haskell in a spreadsheet environment is a good idea. So when we started to discuss the design of the project we formed some design goals for the software. This is discussed further in Section 5.2.1.

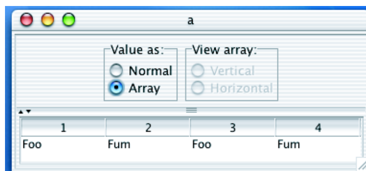


Figure 5.6: The settings view of a value window.

### 5.2.1 Design goals

When discussing the design of Haxcel we formed three main goals.

**Multi-window spreadsheets** One spreadsheet should not be a single 2-D area as in Excel but rather a collection of windows, one per defined entity in the module. This is as far we know a new way to look at spreadsheets. We should also find a different way to index and address cells in a spreadsheet.

**Componentification** by making Haxcel independent from existing Haskell compilers. We didn't want to be tightly bound to a single Haskell environment, instead we wanted to provide a tool that could be a part of any Haskell programmers environment.

**Platform independence** through and other standard components available on many platforms.

Beside this three main goals, Haxcel should provide functionality to create Haskell modules and declaration. Althoug it's possible to work with multiple modules, Haxcel is restrikted to have only one active module. Each declaration should have it's own window and we decided to give the user the reposnibilty of deciding wether a declations value should be shown or not. The user is there for responsible of not showing a infinte data structure, as Haxel does not handel infite reulsts. On the other hand Haxcel does not restrict the use of infinite structures.

#### Platform independence

The third goal was to provide platform independence with main target at Solaris and Mac OS X. We decided to use for implementing the front end of Haxcel. Java was touted to be a platform independent language but has shown to be the different. We therefor decided to only use Standard Java 2 components, e.g Swing [Sun02a]. Another standard component we decided to use was XML [XML02, HM01], and by choosing Suns standard XML package [Sun02c] that now is a part of Java 2 v.1.4 [Sun02b] we have sticked to the idea of standard components.

For the back end of Haxcel we are using hmake (see further discussion in Section 3.2) witch let the user choose Haskell compiler. The major benefits of this are the configurability and that it's a part of the nhc98 [NHC02] distribution of Haskell. This approach is very flexible but have a major drawback too, the use of a compiler makes the turn-around time, see Chapter 8, rather long and makes Haxcel rather useless for interactivity use. We wanted to speed up this and added a choice for another back-end. The distribution of the interpreter Hugs 98(see Section 3.3) has a command line version of the interpreter called *runhugs*. By using runhugs we reduce the the turn-around time allot and get almost instant feedback, see Chapter 8 for benchmarks from our experiments.

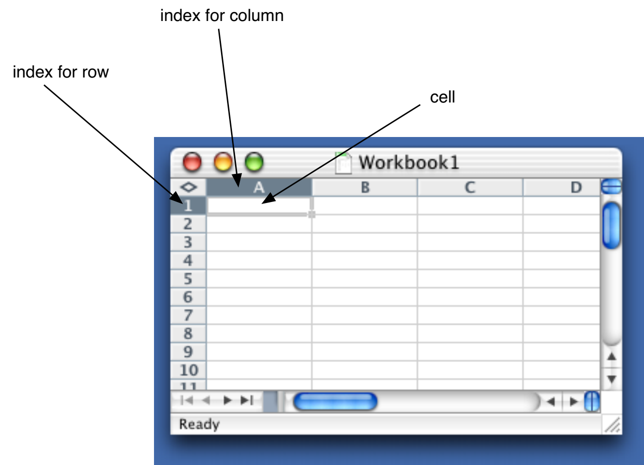


Figure 5.7: An example of the common index scheme.

### 5.2.2 A more flexible indexing scheme

Common spreadsheets have a rigid index scheme for indexing, often a sheet's rows are enumerated by integers $(1, 2, 3, \ldots)$ and columns by letters $(A, B, \ldots, AA, AB, \ldots)$. This way of indexing is effective in relatively small spreadsheets but it does not say anything of the contents of a cell, for instance the cell denoted *A4* can contain an integer, a string or a sound. An example of the common index scheme is shown in Figure 5.7, here represented by Microsoft's Excel (see Section 2.1.1).

Rather than using Excels and other's rigid indexing scheme, where rows are indexed with letters and columns with numbers, any subrange of an `Ix` data type can be used to index any dimension. This gives some interesting possibilities: for instance, one can create own enumerated data types where the names of the elements are descriptive: this can sometimes eliminate the need to put rows with text cells into the spreadsheet.

Another advantage with self defined `Ix` data type is that it makes the spreadsheets more type safe, for instance it makes it impossible to add two arrays of different `Ix`-types.

#### Cell addressing

In common spreadsheets there can be ambiguities in how a cell is addressed depending on if it's the cells value that is addressed or its reference. An example of this is given in Section 2.1.2 were in Mesa the function `ISEMPTY()` takes the address as argument and not its value.

Mesa also implements a function `SAME` that also takes a address as argument and uses that cells formula. If the formula has references to other cells they are changed according to the position between the two original cells. For example, illustrated in Figure 5.8, if cell $B2$ contains the formula $= B3 + 1$ and cell $C2$ contains the formula $= SAME(B2)$ the effective formula in $C2$ is $= C3+1$. If one instead define the formula in $B2$ as $= \$B\$3+1$

the value of $C2$ would be the same as $B2$. If one now define $D2$ as $= SAME(C2)$ there will be an error.

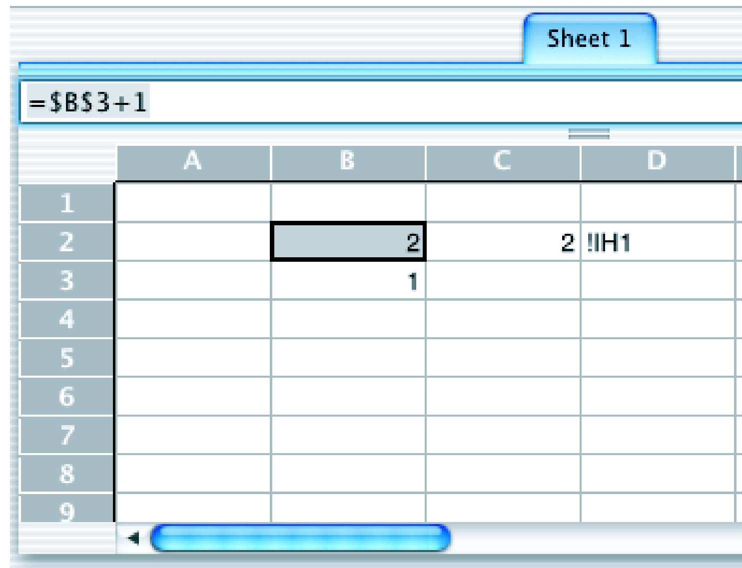


Figure 5.8: Cell addressing.

By using Haskell we avoid pitfalls like this and combined with `Xarray`and in particular `Ix` data types one get a more type safe spreadsheet.

### 5.2.3 HAX Format

When we discussed how to save data our choice fell on XML as an natural choice for specifying the internal format of Haxcel files. With XML it's possible to use standard Java classes for parsing of XML so there was no need for implementing a XML parser. The hax-format is specified in very simple XML syntax, and could probably be more effective by using the DTD standard, see Section 5.6 for further discussion on XML. We decided to keep it simple due to that it's not the main target of the project.

When the hax-file is parsed it has to be analyzed and we defined the grammar for the hax-format as described in Figure 5.9. A simple example of a hax-file is shown in Listing 5.1.

### 5.2.4 Program Generation

In general a Haskell program is semantically complete according to the Haskell Report [PH99] when it is compiled, but the report has not defined a precise meaning to individual statements as Wallace mentions in [Wal00]. We face the same problems as Wallace in making our work semantically correct, but we do not need to evaluate individual expressions in a

```
<?xml version='1.0' encoding='utf-8'?>
<module>
        <modid name="Simpleex"/>
        <declarations>
                <declaration deftype="import">
                        <windowdata ewrect="225,199,98,69"
                                        ewvisible="1"
                                        />
                        <haskell haskellname="Array" valuetype="">
                                <![CDATA[import Array]]>
                        </haskell>
                </declaration>
                <declaration deftype="eq">
                        <windowdata ewrect="119,109,165,94"
                                    ewvisible="1"
                                    vwrect="0,0,0,0"
                                    vwvisible="1"
                                    />
                        <haskell haskellname="n" valuetype="">
                                <![CDATA[n = 3]]>
                        </haskell>
                </declaration>
                <declaration deftype="data">
(... etc ...)
                </declaration>
        </declarations>
</module>
```

Listing 5.1: Haskell source code for Simpleex.hs

```
<!-- enkel grammatik KORREKT????
module : modid definitions

modid : name windowdata

definitions : definition definitions

defintion : deftype haskel

deftype : eq
: class
: instance
: data
: type
: newtype

haskel : haskelname haskelcode

-->
```

Figure 5.9: Grammar of the HAX format. Described in Section 5.2.3

arbitrary order. Instead the order is bound by which declarations the user marks as expressions to be evaluated at execution. This means that all bindings are checked at compile time and the scope of declarations must be correct, otherwise no execution is done and the user is notified through compile errors.

When generating the program from the declarations made we have taken into consideration two different standard cases. The first is when the user has created on module `ModuleM.hs` as described in Figure 5.10. In this case each declaration that is marked to show its calculated value is considered a expression and included in the declaration of main in `Main.hs`. The other standard case is when the defined module is named `Main.hs` as described in Figure 5.11. If this is the case the declared module will be renamed in the program generation phase to avoid a conflict with the `Main.hs` that will be created by Haxcel. The different steps of the program generation phase is described below. First described with *hmake* as the back-end.

1. Build haskell file from module. This step generates a haskell source file based on the declarations made in the main window. All declarations that the user has defined in this module are included. The module is always renamed by adding a underscore in front of the name. This is because we don't want to remove any exported modules in the cleaning up step.

2. Build Main module. All declarations are searched to see if the user has marked

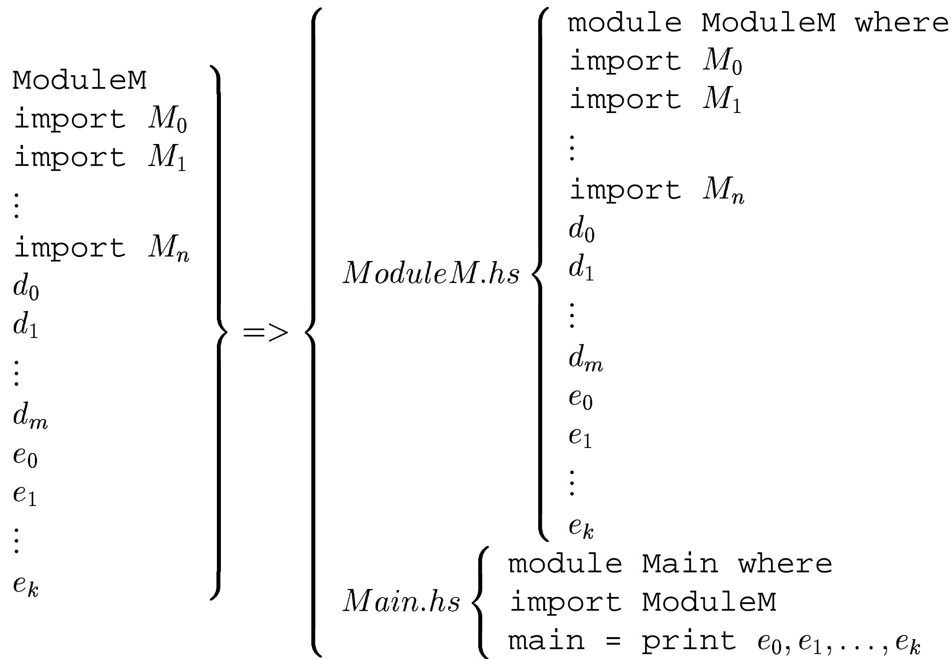


Figure 5.10: Program construction, case 1

one or more declarations to be shown. This declarations are treated as expressions to be computed and are included in the Main declaration that is to executed by the execution step later on.

3. Compile with *hmake*. See also Section 5.3.2.

4. Execute the Main file that is the result of the previous step, this step and the following is described in more details in Section 5.2.5.

5. Parse the output from previous step.

6. Cleanup, that is remove any file used or created in the above mentioned steps.

When *runhugs* is chosen the steps are as follows.

1. Build Haskell file from module. This step generates a Haskell source file based on the declarations made in the main window. All declarations that the user has defined in this module are included. The module is always renamed by adding a underscore in front of the name. This is because we don't want to remove any exported modules in the cleaning up step.

2. Build Main module. All declarations are searched to see if the user has marked one or more declarations to be shown. This declarations are treated as expressions to be computed and are included in the Main declaration that is to executed by the execution step later on.

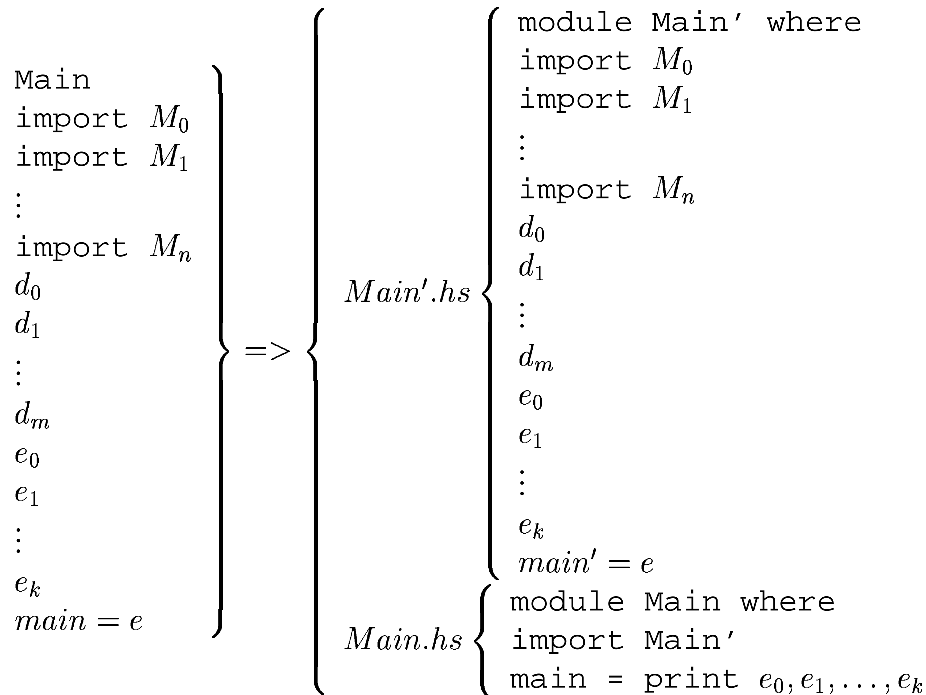


Figure 5.11: Program construction, case 2

3. Start runhugs with Main.hs as input file. See also Section 5.3.2.

4. Parse the output from previous step.

5. Cleanup, that is remove any file used or created in the above mentioned steps.

### 5.2.5 Execution of Main

The result of the compiling is a executable file named Main. This file is executed and the output from the file is parsed by Haxcel. We choose to rely on the Haskell class *Show* for the output data. This means that one can implement data structure that derives this class and *Haxcel*will be able to show them.

#### Format of output

The output from the execution is read by Haxcel and parsed according to the grammar in Figure 5.12. The output is created by a print statement in the *main* declaration that is created in the program generation phase. In discussions we talked about using a more complex format, for instance using XML for tagging the output stream but we decided to keep the syntax very simple and straight forward.

$$
\begin{array}{lcl}
stream & \rightarrow & OUTPUT\ data \\
data & \rightarrow & :id: result\ NEWLINE\ data \\
 & | & \epsilon \\
result & \rightarrow & ARRAY\,dimvalue \\
 & | & NUM \\
 & | & other \\
dim & \rightarrow & (\ dimstring\ ) \\
dimstring & \rightarrow & (\ x1, y1\ )\ ,(\ x2, y2\ ) \\
 & | & x, y \\
value & \rightarrow & [\ celldata,\ nextcell\ ] \\
nextcell & \rightarrow & celldata,\ nextcell \\
 & | & \epsilon \\
celldata & \rightarrow & (\ cell,\ data\ ) \\
other & \rightarrow & <<\ STRING\ >> \\
STRING & \rightarrow & chardataOUTPUT \\
ARRAY & & \\
NUM & &
\end{array}
$$

```
%\[
%
%
%
%
%
% FIXA SNYGGARE GRAMMATIK
%
%  stream -> OUTPUT data
%
%     data -> : id : result NEWLINE data
%             | e
%
%     result -> ARRAY dim value
%               | NUM
%               | other
%
%     dim ->( dimstring )
%     dimstring ->( x1 , y1 ) , ( x2 , y2 )
%      | x , y
%     value ->[ celldata , nextcell ]
%
%     nextcell -> celldata , nextcell
%      | e
%     celldata -> ( cell , data )
%     other ->< < STRING > >
%     STRING -> chardata
%\]
```



Figure 5.12: Grammar for analyzing the output from Main

## 5.3 Implementation

### 5.3.1 Problems to solve

Besides common design issues we have defined the following main problems.

**User interface** general window things like adding, modifying and removing declarations.

**Interface** between Java application and compiler, especially how errors are handled.

**Structured data** like list(possible infinite), trees and arrays needs to be handled and displayed.

### 5.3.2 Compiling

The final build command is a concatination of the three strings *compilecommand="hmake "*, *options="-nhc98"* and *main=" Main"*, all specified in the *HXPrefs*-class. The *options* string can be changed, see 5.1.2 above, but the other two is not userchangeble at this stadge.

### 5.3.3 Parsing structured data

```
array (1,4) [(1,False),(2,True),(3,False),(4,True)]
```

Figure 5.13: Example of an array.

Due to lack of time we had to restrict the prototype to only handle arrays. The Haskell class *Array* derives the *Show*-class and when printed the result can be described in a grammar. The grammar in Figure 5.12 is used to formally describe how we parse the data stream from the execution. In Figure 5.13 we show an example of an array that has four cells. The cells is enumerated from 1 to 4. The dimension string is either a tuple when the array has one dimension or a pair of tuples when the array has two dimension, in other words a matrix or table. We have not implemented higher dimesion arrays due to lack of time(see Section 9.1.6).

### 5.3.4 Parsing numbers

For interpretation of numbers we use Java's constructors for different numerical data types. Basically **NUM** , in Figure 5.12 is first assumed to be of class *float*, although using the *NumberFormatException* we catch other formats, for example number in the form of a rational[2]. If the *NumberFormatException* is thrown by the constructor for *Float* we treat

[2]The haskell library *Ratio* generats output numbers in the form of rationales, for example $0.5$ is shown as $1 \% 2$.

**NUM** as a instance of *String* class. If the *Float* constructor is successful we compare the integer value of NUM and the float value. If the values is equal we instead use the *Integer* class to hold the numerical value of **NUM** . When displaying instances of the java classes Float and Integer the numbers is displayed according to the settings of the users international preferences.

### 5.3.5 Execution

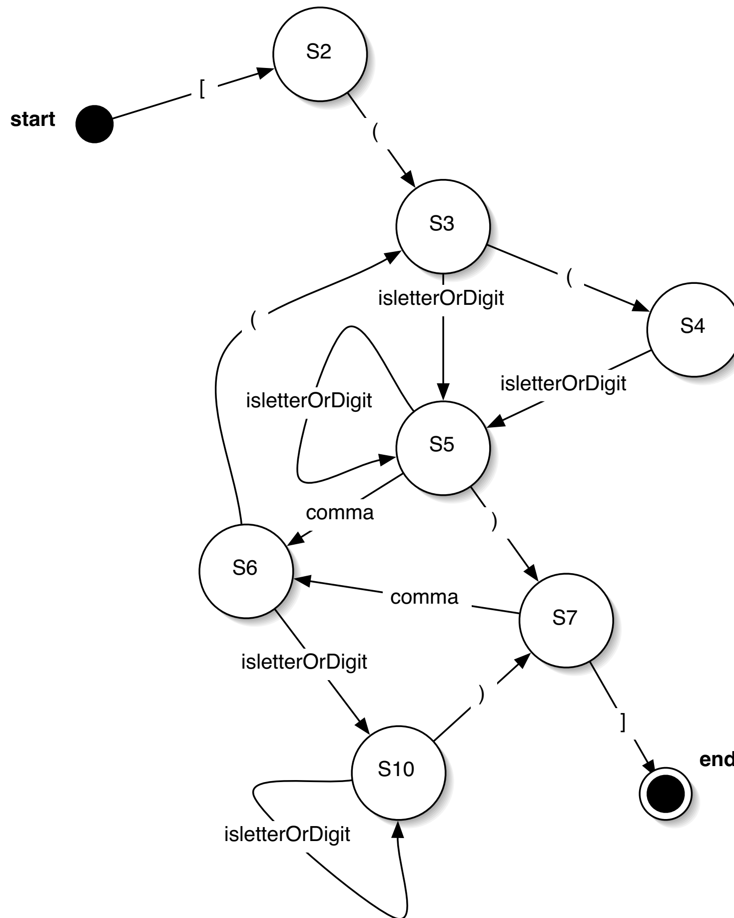


Figure 5.14: The FSM for parsing output from the execution

## 5.4 Implementation details

The software is developed on a Apple Power Macintosh G4 running Mac OS X using Apples Project Builder[Com01] and Bare Bones BBEdit[BBS02] as development environment. The software has been tested successfully on Linux and Solaris but does not work

on Microsoft Windows. The software requires Java, hmake and a Haskell compiler or interpreter and should be working on a lot of different operating systems. See table 5.1 for a complete table of requirements and were to get the required software.

## 5.5 Java

The Object-oriented programming language *Java* from Sun was touted to be a platform independent language. This was appealing to us as we wanted to provide a platform independent tool and Java was chosen as the implementation platform.

## 5.6 XML

XML is an markup language that is derived from SGML. The acronym stands for Extensible Markup Language and was designed for large-scale electronic publishing. XML has become a de facto standard in text-data formats.

We choose XML on the basis of that it's easy to encapsulate other languages inside a markup language which meant that we didn't need to invent our own format for storing information. The XML-format is relatively easy to read by humans and easy to implement parsers for.

## 5.7 Working environment

The table 5.1 shows the software used and needed to make Haxcel work. As far as we now, Haxcel works with current versions of the listed software.

Table 5.1: Version table of used software

| *Product* | *Provider* | *Description* |
|---|---|---|
| hmake | | Version: 3.05 (2002-06-13) |
| ghc | | Glasgow Haskell Compiler, Version 5.00.2, for Haskell 98, compiled by GHC version 5.00 |
| nhc98 | | v1.14 (2002-06-13) |
| hugs & runhugs | | December 2001 |
| Project Builder | Apple | Version 1.1.1 (December 2001 Developer Tools) |
| Java | Sun & Apple | java version "1.3.1", Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-root-020219-20:07), Java HotSpot(TM) Client VM (build 1.3.1, mixed mode) |
| Java XML | Sun | Java XML Pack Spring 02 dev Bundle |

# Chapter 6

# Related works

When studying related works to our projects they can be divided into two major groups, Haskell related and spreadsheet related.

## 6.1 Haskell related

### 6.1.1 Haskell editors

#### VITAL

Related work, du bör kolla in "VITAL" av Keith Hanna (http://www.cs.ukc.ac.uk/people/staff/fkh/Vital/inde
Detta är verkligen "related" i högsta grad!

#### hIDE

*hIDE* [Sve02] is a GUI-based Haskell IDE written using gtk+hs written by Jonas Svensson. It does not include an editor but instead interfaces with NEdit, vim or GNU emacs. Svenson list some of its features; Module tree, Basic CVS support, Extraction of TODO style comments from code, Creation of callgraphs (if graphviz is installed), Find where a function is used, Find functions and modules by name and type.

Our project has as main target the result of the executed program and how the computed data is presented while *hIDEs* main focus is the program that is to be computed.

### 6.1.2 Haskell compilers an interpreters

One may discuss if *Haxcel* is some what a interpreter and not only a visualization tool and editor. Although we use the runhugs or hmake as back-end interpreter/compiler *Haxcel* has to behave like an interpreter to some degree.

*hi - hmake interactive* - Compiler or Interpreter by Malcom Wallace [Wal00]. This article describes the design of *hi*, a interactive front-end to hmake. *hi* is written in Haskell. The report has a good discussion on differences between compilers an interpreters. hmake was our first chose for back-end when we started to design *Haxcel*.

*hugs* is perhaps the most used interpreter for Haskell, and we have chosen runhugs to be one of our back-ends to *Haxcel*.

*The Interactive Lazy ML* by Lennart Augustsson [Aug93]. This article describes the design and implementation of *interactive LML*, which has a Haskell front-end. It's also the foundation for *hbi* the interactive system to *hbc*.

## 6.2 Spreadsheets

In the past there have been some projects like ours that tries to improve the spreadsheet paradigm with a more powerful language. In Section 6.3.2 we presented some of them. Among others *FunSheet* presented in [dHRvE95] has taken slightly different approach than we but have the same idea of a lazy, purely functional programming language witch is not a special dedicated spreadsheet language.

Another approach in this domain is the work presented in [CB97]. Although we have not found any newer work in this project, Clack and Braine work has convinced us that we are on the right track in our approach. They propose spreadsheets that incorporate functional programming features like higher-order functions, a strong type system, curried partial applications, referential transparency and lazy evaluation.

A third proposal in the spreadsheet paradigm is the *Mini-SP*-language presented in [YC94]. Yoder and Cohn here present a dedicated spreadsheet language, and here by differs from us and the above presented work. Yoder and Cohn summarizes their research in [YC97] with the definition of a Generalise Spreadsheet Model.

## 6.3 Other approaches to spreadsheets

There have been many approaches toward the spreadsheet programming paradigm. Spreadsheets has been based on logic, constrains, functional and object-oriented thinking.

### 6.3.1 Logic and constrain based spreadsheets

W. Du and W. Wags writes in [DW90] about their approach to have a 3D spreadsheet based on *Intensional logic*[1]. The benefits of this approach, according to the writers, should be:

> . . . that it considers the whole spreadsheet to be a single entity that varies in three dimensions — two spatial and one temporal. The spreadsheet is denoted by a single variable, that you can use in expressions just like any other variable.

Another logic paradigm approach was done by M. Spenke and C. Beilken in *PERPLEX*. They show in [SB89] the possibilities in combining logic programming with spreadsheets, using programming-by-examples to define user defined predicates. Formulas in

[1]Intensional logic is concerned with assertions and other expressions where meaning depends on implicit context.

PERPLEX are not on form of functionals but are given as constraints. The emphasis in PERPLEX was to build a general programming tool and differs in that way, among others, to another constraint based approach given by M. Stadelmann in [Sta93] were the emphasis is on a powerful calculation tool. One of the main benefits of using constrains instead of formulas is that a constrain acts like a true equation that can be solved for any unknown cell.

### 6.3.2 Functional Spreadsheets

The functional programming approach to spreadsheets has been a used before us, among others is is *FunSheet* described in [dHRvE95] and the paper [CB97] where C. Clack and L. Brainde describes the design of a object-oriented functional spreadsheet. The latter is written within the CLOVER project see [BC96] for details. The functional part of the presented design includes high-order functions and lazy evaluation. Although they have a object-oriented approach the FP part is essential and they present the grammar for a functional spreadsheet very detailed.

de Hoon describes the design and implementation of *FunSheet*, a functional spreadsheet, in the article [dHRvE95]. FunSheet uses a purely functional higher order language to allow the user to describe spreadsheets computations. The language is a Clean-like, untyped functional language. In FunSheet every column is treated as functions, witch is a good design model, but we think that this gives columns a higher importance than columns and we want to use Haskell's way of define data types as indexes, both for columns and rows.

Yoder and Cohn present in [YC94] a spreadsheet language called Mini-SP. They define this functional programming language to be structured, concurrent and scalable. The language is to be a response to the bad languages in common spreadsheets and is a interactive and declarative approach that the spreadsheets bring to the array programming arena. The Mini-SP language uses ideas from More's array theory and APL. In common spreadsheets it's possible to have absolute and relative addressing to cell references. Mini-SP preserves relative addressing but absolute is default, relative addressing is done through prefixing with a "+" or "-". Cells can be grouped into rectangular blocks and then representing a array witch of course can be at least two-dimensional.

# Chapter 7

# Examples

We will in this chapter give three examples of how Haskell can be used and we will also present some benchmark results. We think that these three examples shows the benefit of using Haskell as a spreadsheet programming language.

The first example is a real simple example describing the Haskell syntax of array and how Haxcel visualize a array.

The second example is an example that shows how it's possible to write a function with local variables and recursion for estimating a curve data. The example is from environmental science but it should be obvious that computations with Haskell and a spreadsheet visualization of test results and calculated values can be used in any discipline with need for a advanced mathematics.

The third and last example make use of XArray and is a small budget-application.

## 7.1 An simple example

First a very simple example which assigns values to array of length specified by the variable $n$. When the index of the array is odd the value assigned is `Foo` if the index is even the value is `Fum`. The source code is listed in 8.

In Figure 7.1 all windows associated with Simpleex is displayed and particularly the view of Array $a$ is showed in spreadsheet form.

```
module Simpleex where
import Array -- standard Haskell array library
n = 3
data Fee = Foo | Fum deriving Show
f x | odd x  = Foo
    | even x = Fum
a = array (1,n) [(i,f i)| i <- [1..n]]
```

Listing 7.1: Haskell source code for Simpleex.hs

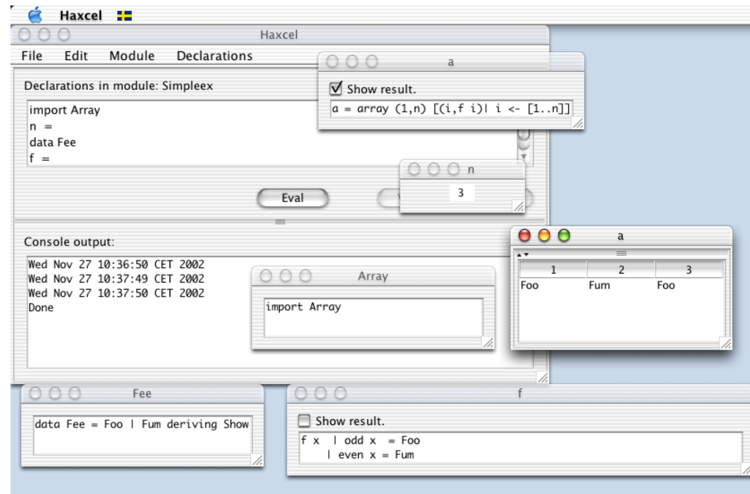


Figure 7.1: The windows associated with the first example Simpleex.

## 7.2 An application for environmental science: Toxicity Ratings

INTE FÄRDIGT TABBELLEN ÄR BARA EN DUMMY GRAF ÄR EN DUMMY

This is an example from environmental science were spreadsheets often is used to handle measured values from different tests. After the test is done several values are calculated from the given test results, using numerical methods. We here present an example of calculating $LC_{50}$ using XXX. The example has been supervised by Anna Kejler, Environmental Engineer.

One common way to measuring toxicity is to make a dose/response curve out of a test population which are exposed for a chemical. A convenient way of describe toxicity of a chemical is to determine the dose to which 50 percent of the test population is sensitive. There are a number of terms in regular use in the study of toxic effects for example **lethal concentration (LC)** meaning a concentration of a poison causing death, or sufficient to cause it, by direct action. The sensitivity of a organism vary with bodymass, metabolism etc.

The result of a test is often a S-shaped curved (see Figure 7.2) and to get a value for comparison one calculate for example $LC_{50}$, were 50 is the percentage of animals killed at a particular concentration.

## 7.3 Budget for a small research department

The Haskell source is listed in 32

As a simple example of spreadsheet programming in Haskell with the extended array library, we give a little cost budget for a fictitious research group. The costs are the salary

```
module Budget_ where
import Xarray
import Ix
data Employees = John | Sarah | Albert deriving (Eq,Ord,Ix,Show)
salaries = listArray (John,Albert) [18000.0,18300.0,24900.0]


lkp :: Fractional a => a
lkp = 0.5

dept_OH :: Fractional a => a
dept_OH = 0.1
univ_OH :: Fractional a => a
univ_OH = 0.15


office_OH :: Fractional a => a
office_OH = 0.19
vat :: Fractional a => a
vat = 0.08
table = let l = [salaries,
                        lkp*salaries,
                        dept_OH*salaries,
                        univ_OH*salaries,
                        office_OH*salaries]
        in cols2matrix (l ++ [(sum l)*vat])
person_costs = sum (matrix2cols table)
category_costs = sum  (matrix2rows table)
totcosts = sumA person_costs
instance Pord Employees
nisse = "Hello␣Working␣class"
```

Listing 7.2: Haskell source code for Budget.hs

Table 7.1: Test result

| *Day* | *Absorbing* |
|---|---|
| 1 | 0.11 |
| 3 | 0.11 |
| 4 | 0.11 |
| 5 | 0.11 |

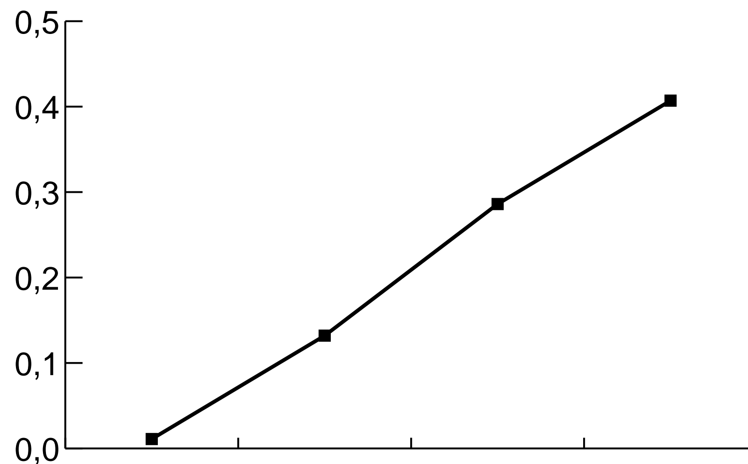


Figure 7.2: The reading illustrated in a graf. The calculated LD50 is marked.

costs plus a number of overheads, most of which are calculated as certain percentages of the salary cost, the VAT however as a fraction of the total cost (including the other overheads). We want to be able to maintain the budget as smoothly as possible: the members in the group might change, new costs may be charged the group, or the existing overhead ratios may change. We want to calculate the total cost, and the cost per person, but also the cost broken down in different categories (department overheads etc.).

Our solution is to put all costs in a matrix, whose first dimension is indexed by a type with one constructor per member. The matrix is assembled from a number of vectors, one per cost category. To keep overhead ratios easy to modify they are made symbolic through simple definitions, which are simple to change if the need arises. The definitions are given in Listing 32, and a snapshot of the Haxcel screen is shown in Fig. 7.3. (This snapshot shows a hypothetical situation where we want to play around with the overhead factors in order to see the effects on the various parts of the budget. Thus, the editing windows for the overhead factors are open, as well as the value windows for the costs in the budget.)

Note the explicit typing `Fractional a => a` on the "scalar" numerical values `lkp` etc. They are used to scale arrays. Without this typing they would not be lifted to infinite arrays, and the scaling wouldn't work. This problem is due to the monomorphism restriction in Haskell's type system.

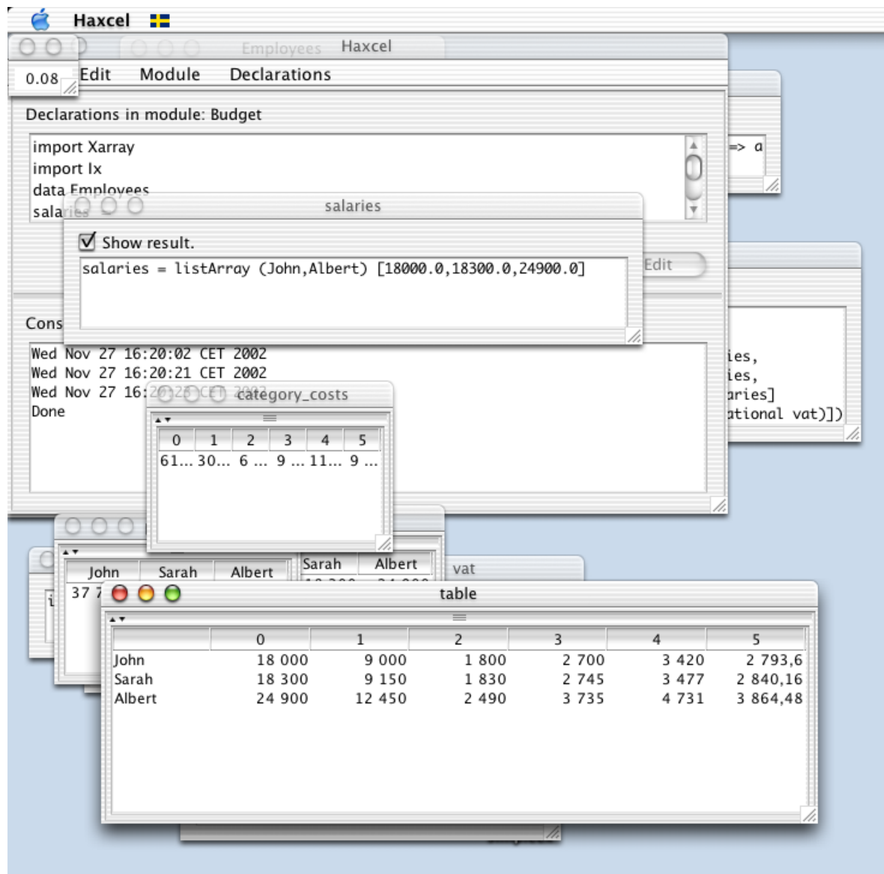


Figure 7.3: Snapshot of the Haxcel screen for the budget spreadsheet.

Our simple example demonstrates some features that might be useful when programming spreadsheet applications. Ordinary spreadsheets can be heterogenous, and strings are often put into cells to tell what numerical fields, rows, or columns mean. In Haskell, one can instead use simple algebraic data types like `Employees` to create descriptive index sets for arrays. An advantage, besides the type-safety, is that several arrays now can have their index sets given by the same data type. This is a win since one often uses different arrays to describe different properties of the same set of items (like the employees of the research group in our example). If this set changes (if, say, we hire more people), then, if the arrays are defined in the right way, it will be very easy to update the spreadsheet by simply changing the data type declaration and adding/removing input data in the proper arrays. In the usual spreadsheet model, this kind of update can be very error-prone.

It is not hard to imagine an interactive mechanism for defining arrays, where index sets can be created and copied between arrays, and the corresponding data type declarations are automatically generated.

# Chapter 8

# Experimental results

To get a grip on the turnaround time for evaluating spreadsheets, we did some simple benchmarking. We ran the two examples presented here, *Simpleex*(Section **??**) and `budget` (Section 7). We measured: the time to assemble the Haskell files for compilation, the time for hmake/nhc98 to compile, the time for the generated program to execute, and the time to parse and display the results. We ran the examples on two machines: a Power Mac G4 with a 400 MHz Power PC G4 processor, 192 MByte RAM, and 1 MB cache, under Mac OS X 10.1.5 with the nhc98 v. 1.12 compiler, and a Sun Ultra Enterprise 4000/5000 with six 167 MHz Sparc processors (whereof we used one), 256 MB RAM and 0.5 MB cache per processor, under Solaris 2.6 with the nhc98 v. 1.14 compiler. On the Power Mac we also ran both examples with runhugs. Each example was run 5 times for each configuration. The results (mean values) are presented in Table 8.1. Note that the different times were measured with different resolution, due to the kind of timing tools we had available: the java-specific parts (assemble, parse & display) were measured with resolution 1s, and the compilation and execution were both measured with resolution 0.01s on the Mac and 0.1s on the Sun, respectively. The variation in measured time between runs was low, except for the measurements of the java-specific parts where the low resolution sometimes caused jumps in the observed times.

Not surprisingly, the main time for Haxcel with hmake/nhc98 is spent compiling. This yields a turnaround time of the order 10s on our systems. This is not readily acceptable for an interactive environment. On the other hand, better compilation times might be obtained on other platforms: for instance, Wallace [Wal00] claim compilation times for *nhc98* in the order 1-2 seconds for medium-sized modules, on a PC with Linux and a 500 MHz pentium processor.

For Haxcel with runhugs on the Power Mac, we obtain much faster turnaround times, of the order 1s for both examples. Although not instantaneous, this is a much more acceptable response time.

Most of the display time for *Budget* can be attributed to the first run for each experiment, where extra time is spent opening the value windows. Subsequent runs mostly have lower display times.

| Simpleex | | | | | |
|---|---|---|---|---|---|
| | **Assemble** | **Compile** | **Execute** | **Display** | **Total** |
| Mac/`runhugs` | 0.0 | N/A | 0.60 | 0.0 | 0.60 |
| Mac/`hmake` | 0.0 | 5.44 | 0.06 | 0.0 | 5.50 |
| Sun | 0.2 | 7.4 | 0.0 | 0.0 | 8.2 |

| Budget | | | | | |
|---|---|---|---|---|---|
| | **Assemble** | **Compile** | **Execute** | **Display** | **Total** |
| Mac/`runhugs` | 0.0 | N/A | 0.79 | 0.4 | 1.19 |
| Mac/`hmake` | 0.0 | 7.37 | 0.29 | 0.4 | 8.06 |
| Sun | 0.2 | 10.3 | 0.7 | 0.4 | 11.6 |

Table 8.1: Measurements for `Simpleex` and `Budget`, in seconds.

# Chapter 9

# Conclusions

We have designed *Haxcel*, a compiler-independent spreadsheet interface for Haskell programming, and presented the accompanying extended array library `Xarray`. Together, they demonstrate what a high-level programming environment can look like, which can be used both for conventional Haskell programming and for spreadsheet calculations. We also think Haxcel overcomes some of the problems with the conventional spreadsheet model, and it extends current spreadsheet languages to the full power of Haskell.

Do we think that Haskell is a good language for spreadsheet calculations? Yes, probably for someone who has a good grasp of modern functional programming. But we do not think Haskell in its current form is the perfect spreadsheet language. Spreadsheet calculations are best expressed in an array model, and while quite a few array language primitives can be reasonably well embedded in Haskell we have also found some annoying limitations. For instance, the ability to overload numerical literals as infinite constant arrays, but having to provide explicit type annotations for variables defined to have the values of these literals, is a safe bet for tripping spreadsheet programmers not well acquainted with Haskell's type system. Furthermore, Haskell was designed with lists rather than arrays in mind: thus, all the "good" operators and function names are already taken for list operations. This makes the language less intuitive for array-oriented expressions. Some of these problems could possibly be alleviated by providing a "simple spreadsheet mode" to Haxcel, where assumptions about data types are made and explicit type declarations are automatically generated, but this feature remains to be added and evaluated.

## 9.1 Future work

We now has a prototype that we have used for conducting some test and have found some topics that needs to be improved and further developed. Clearly, many features can be added to Haxcel.

The current turnaround times for small examples are annoying for hmake/nhc98 but acceptable when using runhugs. If still faster turnaround times are needed, then one could add a mode where hugs runs in a parallel thread rather than being invoked in batch mode. If this is still not enough, then one could make a more elaborate dependency analysis to

recalculate and update only those values that have changed since the last evaluation. We believe all these features would be fairly straightforward to implement.

### 9.1.1 Better error handling

The current lack of error handling in Haxcel is not satisfactory. Also, the lack of knowledge about types is a limitation. As mentioned earlier, a possible solution is to add modes to Haxcel where it interfaces more closely to certain compilers. However, this complication could be avoided if there were standard protocols for Haskell compilers to communicate type information and error messages. Such protocols could then also be used by other Haskell tools. Also, if parsers in Haskell compilers could be used as components to produce parsed code in some standard format, then tools like Haxcel could take advantage of them. The proposed Haskell Execution Platform [SMG+99] has addressed some of these issues, but we don't know to which extent it has materialized.

### 9.1.2 Editing in the spreadsheet-view

Better means to define arrays interactively would be nice for spreadsheet programmers: currently, they can be defined only through textual Haskell expressions.

### 9.1.3 More GUI-functionality

The *Edit*-menu lack of some item usually found here. It could be a good idea to make Haxcel more "user friendly" and implement menu driven copy, cut and paste commands. Another missing command in the *Edit*-menu is Undo/Redo.

Better graphical support to display data structures like lists is desirable, as well as the possibility to link cells to external viewers or players, and an ability to import data from the outside and convert into suitable Haskell values. The simple textual input of declarations could be enhanced into a more visual programming environment.

It may be a good idea to exclude the name of the Haskell declaration from the editable field of a declaration. A benefit from this would be that renaming would be more consistent.

We had thoughts about implementing functionality for showing a one dimensional array in horizontal or vertical view. We still thinks that this could be a good idea but have not implemented the functionality though we have implemented the controls for this in the settings view for a value window.

### 9.1.4 Literate scripts

It would not be difficult to implement support for literate scripts (see Section **??**). One thought about literate files we have discussed is making the hax-file to literate script, making it easier to convert Haskell modules to *Haxcel*.

### 9.1.5 Import of a existing Haskell module

Haxcel today support export of Haskell-modules, in other words creating a regular `.hs`-file for use independently from Haxcel. It would be nice to be able to go the other way, importing an existing Haskell-module to Haxcel, this has not been implemented due to lack of time.

### 9.1.6 Higher dimensions arrays and other structured data types

We discussed in the design of *Haxcel*different ways to display 3-D arrays an other data structures like trees and list. Due to lack of time we decided to only implement arrays of one and two dimensions.

# Bibliography

[AL01] Jens F. Adam and Stephen Lindenmayer. *RagTime 5 Reference*. RagTime GmbH, 2001.

[Alg01] Eugene Algorithms. TeXShop version 1.13d, 2001.

[Aug93] Lennart Augustsson. The interactive Lazy ML system. *Journal of Functional Programming*, 3(1):77–93, Jan 93.

[BBS02] Bare Bones Software. BBEdit 6.5.2 for Mac OS X, 2002.

[BC96] Lee Braine and Chris Clack. Introducing CLOVER: An object-oriented functional language. In *Implementation of Functional Languages*, pages 1–20, 1996. citeseer.nj.nec.com/article/braine96introducing.html.

[BF99] Dan Bricklin and Bob Frankston. Visicalc: Information from its creators, 1999. http://www.bricklin.com/visicalc.htm.

[CB97] C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming*, GlaFP'97, September 1997.

[Com01] Apple Computer. Project Builder, version 1.1.1 (december 2001 developer tools), 2001.

[dHRvE95] Walter A. C. A. J. de Hoon, Luc M. W. J. Rutten, and Marko C. J. D. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 5(3):383–414, 1995.

[DW90] W. Du and W. Wadge. A 3d spreadsheet based on intesional logic. *IEEE Software*, May 1990.

[Gra02] Felix Grant. Looking beyond excel. *Scientific Computing World*, pages 24–28, January/February 2002.

[HM01] Elliotte R. Harold and W. Scott Means. *XML in a nutshell, A desktop quick reference*. O'Reilly, 2001.

[Lis98] B. Lisper. Data fields. In *Workshop on Generic Programming, Marstrand, Sweden*, June 1998.

[LM02] B. Lisper and J. Malmström. Haxcel: A Spreadsheet Interface to Haskell. In *14th International Workshop on the Implementation of Functional Languages*, pages 206–222, September 2002.

[NHC02] nhc98 from york, 2002. http://www.cs.york.ac.uk/fp/nhc98/.

[Omn02] The OmniGroup. OmniGraffle 2.0.4 (v20)). http://www.omigroup.com, 2002.

[PH99] S. L. Peyton-Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. http://www.haskell.org/onlinereport/, 1999.

[P&L01] P&L Software. *Mesa User's Guide*, 2001.

[QUA] Corel corporation. http://www.corel.com.

[SB89] Michael Spenke and Christian Beilken. A spreadsheet interface for logic programming. In *CHI'89 Proceedings*, pages 75–80, 1989.

[SMG+99] Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, and Simon Peyton Jones. Architecture of the haskell execution platform (hep). Technical report, Microsoft Reaserach, Cambridge, UK, July 1999.

[SPL] Insightsful. http://www.insightsful.com.

[Sta93] Marc Stadelmann. A spreadsheet based on constraints. In *ACM Symposium on User Interface Software and Technology*, pages 217–224, 1993.

[Sun02a] Sun Microsystems, Inc. *Java Swing*, 2002. http://java.sun.com/j2se/1.4/docs/api/javax/swing/package-summary.html.

[Sun02b] Sun Microsystems, Inc. *Java(TM) 2 Platform, Standard Edition (J2SE(TM))*, 2002. http://java.sun.com/j2se/1.4/index.html.

[Sun02c] Sun Microsystems, Inc. *Java(TM) Technology and XML*, 2002. http://java.sun.com/xml/.

[Sve02] Jonas Svensson. hide, 2002. http://www.dtek.chalmers.se/ d99josve/hide.

[Wal00] M. Wallace. hi — hmake interactive — Compiler or Interpreter? In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, Aachener Informatik Berichte, pages 339–345. RWTH Aachen, 2000.

[XML02] Extensible markup language (xml), 2002. http://www.w3.org/XML/.

[YC94] A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. In *International Conference on Computer Languages*, pages 20–30. IEEE, 1994.

[YC97] A. Yoder and D. Cohn. Domain-specific and general-purpose aspects of spreadsheet languages, 1997.

# Appendix A

# Defintions

## A.1 Words

To relsolve any ambiguities this section deals with definitions of words.

**Spreadsheet** A spreadsheet contains *cells* which is orded in *rows* and *columns*. In a spreadsheet it's possible to edit figures and tables and perform complex calculations.

**Cell** A cell possesses three elements; *value*, *format* and *formula*.

**Row and column**

**Value**

**Format**

**Formula**

**Worksheet** same as *spreadsheet*

**Workbook** a collection of spreadsheets.