ADDING NESTED HEADERS AND A PROPER GTK-BASED GUI TO THE HASKELL TABLE TOOLS

ADDING NESTED HEADERS AND A PROPER GTK-BASED GUI TO THE HASKELL TABLE TOOLS

By
SEPANDAR SEPEHR B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2010) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Adding Nested Headers and a Proper Gtk-Based GUI
to The Haskell Table Tools

AUTHOR: Sepandar Sepehr B.Sc. (University of Tehran)

SUPERVISOR: Dr. Wolfram Kahl

NUMBER OF PAGES: viii, 59

Abstract

Specification of Large Scale Systems like Safety Critical Softwares is an important yet frequently tedious process. The requirement analysts often face a large number of conditions and cases during the process of requirement elicitation. One of the tools used to deal with this is employing multi-dimensional mathematical expressions, called *Tabular Expressions* by Parnas, to show and to study the specifications and to also check the properties of the system.

The compositional syntax that was introduced by Kahl is a simple and comprehensible basis for implementation of *tabular expressions*. Kahl's implementation of his formalization in the functional programming language *Haskell* started a new approach to syntax and semantics of tabular expressions.

In this thesis, we expand the project that Kahl initiated using his formalization to establish tool support of regular tables. A tool like this one has always been missing for people who use tabular expressions in the requirement analysis. We have designed a platform-independent, graphically interactive, and expansible tool using GTK in Haskell. In this thesis, we have also added the support of nested headers and implemented some features needed to work with nested headers for a better practical tool.

Acknowledgments

I would like to firstly thank my family who have always supported and encouraged me in the most important ways throughout my life.

Also, I am very thankful to Dr. Wolfram Kahl for enabling me to pursue graduate studies and giving me this opportunity to study at McMaster University. His comments and ideas are always informative and they greatly helped shape my work.

Lastly I would like to thank my friends who helped me stay balanced during the stressful times.

Contents

List of Figures

Chapter 1

Introduction and Motivation

Since 1977[1] when *tables* were introduced by Parnas *et al.*, plenty of projects have used tabular expressions extensively in the documentation. Throughout the time, the importance of tables and tabular expressions has been proven. However, the need for a comprehensive tool is still not satisfied. The main issues that a practical tool should therefore consider are the following:

- The tool should be general and adaptable to different projects.
- Diverse types of expressions should be supported and it should be able to accept new expression types.
- It should be standalone and not dependent on other tools.
- Platform independency is a very important feature.
- The tool should support nested headers. Tables have become more complicated and users demand a tool that can facilitate the creation and editing of tables with nested headers.

Previous Tools

For more than 20 years, tables have been used in many projects. However, there has not been a functional tool for tables. In 2005, Constance Heitmeyer *et al.* introduced a toolset for SCR (Software Cost Reduction) tabular notation. In paper [HABJ05], Heitmeyer states: "the tools include a specification editor, a consistency checker, a simulator, and tools for verifying application properties — including a model checker, a verifier, a property checker based

[1] As described in [Hen80], and later explained in [BKS97] and [JPZ97]

on decision procedures, and an invariant generator".

In 2007, Dennis K. Peters *et al.*, in [PLW07] explained their IDE for tabular expressions. This was the first time a tool was developed for tabular expressions. However, it was not a standalone tool, it is an IDE inside the Eclipse Framework based on OMDoc Document Model. The IDE does not seem to be complete; thus, it requires further work.

Wolfram Kahl, in 2003 defined the Compositional Syntax and Semantics of Tables. He also developed the Haskell implementation of tables which has been the infrastructure of the current project.[2] Kahl *et al.* have been working on the project and developed a tool for tabular expressions using Gtk2Hs in Haskell having a Mozilla embedded widget. [Kah04] constitutes a draft of the last project, which has not been published yet.

Kahl has led a detailed work for designing an elegant tool for tabular expressions. Since the project has been developed using *Haskell*, it has been possible for different students to become involved in the project. A clear elaborate design using Haskell makes the codes scalable, expandable and easy to understand. Using the same project, it was possible for us to expand previous undertakings and to add new features. We also tried to make it practical, easy to use in order to satisfy the desired criteria.

However, using *mozembed* [3] in the program hinders it from begin a viable solution for our need. Mozembed is not a completely supported extension of Gtk toolkit and it has not been precisely developed in Gtk2Hs either. Besides, installation of mozembed in Gtk2Hs needs a lot of effort and expertise and it is usually a very tedious process that will difficult the use of the tool for most users. This approach will also result in the tool not being platform independent. These shortcomings prompted us to use Kahl's project which was developed with an outstanding pattern that allowed us to rewrite the parts we needed to adapt a better graphical interface. In addition, considering the need for nested headers in table tools, we decided to implement nested headers in our tool to create a new state-of-the-art project.

[2]For more information, the codes and paper refer to his website at `http://www.cas.mcmaster.ca/~kahl/Publications/TR/Kahl-2003a_CSST/`

[3]A widget that allows the users to embed Mozilla browser in Gtk application

Organization

In chapter 2, we discuss the origins of *tabular expressions*, survey previous tools available for tables and the algebras that are used for development of our project.

Then, chapter 3 introduces the designed types for holding tabular expressions. It also includes the design and the general outline of the tool that we worked on following the new design and related changes in order to adapt some new features.

In chapter 4 we show what the new tool looks like and how it works. We illustrate the implemented features and we also indicate how the user should interact with them. This chapter can be approached as an introduction to nested headers for readers who are not familiar with the notion of *nested headers*.

Afterwards, in chapter 5 we overview the new types that we have introduced to cover nested headers and the features they compose. The modules that have been changed or added to modify the old GUI are introduced in chapter 6 where we explain the types and methods we have written for this purpose.

Finally, in chapter 7 we describe the process of installation and the basic information needed to work with the tool. It is followed by the last chapter 8 that is the conclusion and it also addresses the future direction and possible future work of the thesis. Lastly, some source codes are placed in the appendix.

Chapter 2

Background and Related Work

Safety Critical Systems, as well as other complicated systems, require a precise, correct and complete specification. In section 2.1, we explain how tabular expressions became a reliable candidate and the evolution of this idea.

Furthermore, the algebras behind the design of our tool are briefly summarized in section 2.2. A short explanation of the compositional table syntax and semantics has also been added in this section. Finally, section 2.3 briefly introduces Haskell programming and some basic concepts used in Haskell.

2.1 Tabular Expressions

David L. Parnas is known as the inventor of Tabular Expressions. In late 1970s, Parnas *et al.* used *tables*, which were two-dimensional mathematical expressions used in the presentation of complex relations in software requirements specification documents. Later, Parnas proposed the first formal definition of 10 basic classes of tables in [Par92]. In this paper, Parnas defined multi-dimensional notations and called them tabular expressions, which he claims "have proven useful for describing the specified mathematical functions in practical applications" as cited in his paper.

Later on, in [Jan95], Janicki combined the definitions of 10 classes of tables into one general definition. Having a homogeneous approach, Janicki extracted the semantics from different tables and suggested a new technique for storing the relational meaning (the semantics) of many tables. He used *cell connection graph* approach, which holds the *information flow*, to show the se-

mantic of a particular table dynamically. Using Janicki's tabular expressions, they could show Parnas's classes of tables in one class of table and the semantic of it accompanying, which means that each of Parnas's tables could be a special case of Janicki's. This concept was later further developed in [JK01].

Our objective is not to explain in detail the basic ideas of tables, but we would like to clarify the concept of nested headers with a simple example. In Figure 2.1 we see a tabular expression shown in a table [1] with two dimension. Figure 2.2 shows the same expression in a one-dimensional table with nesting. We can see that the result of the function according to both of the tables for values of $x = 0$ and $y < 0$ is $x-2$, and for $x = 0$ and $y \geqslant 0$ is $x-1$.

	x < 0	x == 0	x > 0
y < 0	2 · x	x - 2	3 - x
y ≥ 0	2 · x	x - 1	3 - x

Figure 2.1: Table T_X, an example of a two-dimensional table

x < 0	x == 0		x > 0
	y < 0	y ≥ 0	
2 · x	x - 2	x - 1	3 - x

Figure 2.2: Table T_Y, a one-dimensional table representing the same expressions of Table T_X with nested headers.

However, the table without nested header (T_x) has 6 grid cells and the table with nesting (T_Y)holds just 4. This makes it more clear and easier to understand. The third table T_Z in Figure 2.3 also holds the same expression, in one dimension and without nesting. The advantage of T_Y over T_Z is that $x = 0$ is written just once and is more comprehensible.

All of the three tables represent one mathematical function that is:

[1] The figure is generated using our own tool that we are talking about in this thesis

x < 0	x == 0 ∧ y < 0	x == 0 ∧ y >= 0	x > 0
2 · x	x - 2	x - 1	3 - x

Figure 2.3: Table T_Z, a one-dimensional table representing the same expressions of Table T_X and T_Y without nested headers.

$$f(x,y) = \begin{cases} 2 \cdot x & \text{if } x < 0 \\ x \cdot 2 & \text{if } x = 0 \ \wedge \ y < 0 \\ x - 1 & \text{if } x = 0 \ \wedge \ y \geqslant 0 \\ 3 - x & \text{if } x > 0 \end{cases}$$

2.2 Table Algebras

In 2003, Wolfram Kahl in [Kah03] presented a new framework to define *"semantic rules"* for tables that were not only general, but also flexible. Kahl claims that "this new table semantics framework is explicitly motivated by the desire to have an understanding of tables that can be used for reasoning about tables and table transformation, and also as a basis for machine-support of table manipulation and transformation." He also affirmed that "it may therefore appear less "direct" to the table user, but the compositional approach we take has advantages for reasoning and mechanization."

In [Kah03], Kahl defines the compositional view of table syntax. Then he introduces formal definition, typing of tables, table constructors and consequently the first *table algebra* for tabular expressions. This work is then continued with cooperation of Furusawa in [FK04] by presenting an algebraic structure that includes *nested headers* in detail.

A detailed specification of *Basic Table Algebra* and the algebra of tables with *Nested Headers* and *General Table Algebra* is included in [FK04]. In this section, we briefly cover the *Compositional Table Syntax and Semantics* that is the basis of our tool's tables.

Compositional Table Syntax and Semantics

In [FK04] and [Kah04], Kahl has explained the reasons for developing a new syntax and semantics for tabular expressions. Parnas, as said in [Par92], has come to the conclusion that having their syntax of tables and not capturing the meaning (semantics) of tables, they face different "interpretations". Consequently, they decide to categorize different interpretations into ten categories of tables and not a general case. Parnas concluded "the various interpretations do not have enough in common to justify attempting a single, more general, definition of all interpretations" [Par92].

As Kahl has explained, the first unified framework was proposed by Janicki and the term *tabular expression* came into existence. "All the table semantics definitions mentioned up to here rely heavily on indices in their formulations", Kahl has stated in [Kah04]. Later on, a more elegant formalism was proposed by Desharnais, Khedri and Mili in [DKM01].

Kahl has proposed the latest framework that we used in this project in [Kah03]. For more information on the details of compositional table syntax and semantics and how we compose tables, refer to [Kah04].

2.3 Programming in Haskell

Haskell [PJ+03] is a computer programming language that is strongly and polymorphically statically typed, lazy, purely functional. It contains many features according to [Hut07]; including:

- Concise programs
- Powerful type system
- List comprehensions
- Recursive functions
- Higher-order functions
- Monadic affects
- Lazy evaluation
- Reasoning about programs

From the features above, the main ones that are relevant for this project are the type system, list comprehension and higher-order functions. Moreover, a concise program that allows us to put in place shorter and more clear code, will also provide us with considerable support for the maintainability of the project.

One of the main ways to structure and manipulate data is by using lists. Therefore, being able to work with lists easily and having many useful functions to work on lists is a great advantage. Haskell has a powerful built-in list comprehension notation that makes it very easy for us to define functions associated with new data types and to work on them. Other features are discussed in the following sections.

2.3.1 Higher-order functions

A higher order function is a function that takes other functions as arguments or returns a function as result. For example the function twice in the example below is a higher order function.

```
twice    :: (a → a) → a → a
twice f a = f (f x)
```

The function twice as seen in the signature of it, takes a function from a to a and then gets another value of type a and returns a value of type a. Then, a can be replaced by a type like Int, String, Bool or any user-defined type. We use this feature in many places like defining the semantics of the tables that are functions and apply them to expressions of the tables.

2.3.2 Types and Classes

Two of the most important concepts of Haskell are Types and Classes. Types are usually categorized into Basic (Bool, Char and etc.), List, Tuple, Function, Polymorphic and Overloaded types. Besides, the other notion is *Type Classes*. "A *class* is a collection of types that support certain overloaded operations called *methods*" [Hut07]. Haskell's type system provides many features to avoid run-time errors, by using sophisticated *type inference*.

Type Declaration

In Haskell, there are some mechanisms for declaring new types and classes. A typical example of type declaration is type declaration of String. Strings are defined as:

```
type String = [Char]
```

Using **type** declarations, we are simply introducing synonyms for convenience. There are other types as we explained that can be declared using the same mechanism but we are not going into the details. Complete explanation of type declarations can be found in Haskell textbooks such as [Hut07] and in the Haskell 98 Report [2].

Data Declaration

If we do not want to declare type synonym, but create a completely new type, we use the **data** mechanism. One of the simplest data types is Bool which is defined as:

```
data Bool = False | True
```

A data declaration can also be parametrized. For example data type Maybe is declared:

```
data Maybe a = Nothing | Just a
```

The **data** mechanism can also be used to declare recursive types as the simple example is for natural numbers.

```
data Nat = Zero | Succ Nat
```

This means a natural number is either Zero or successor of another natural number, constructing natural numbers recursively.

[2]The Haskell 98 Report is available online at `http://www.haskell.org/onlinereport`

Newtype

A concept between type synonym and datatype is **newtype**. It has a constructor like a datatype, but it can have only one constructor with only one argument. According to Haskell 98 Report, newtype declarations are called renamed datatypes. For example we can have:

```
newtype NewInt = NICons Int
```

This introduces a new type that has the same implementation as Int but may have different type class instances associated with it. For example, one could change the ordering of integer numbers.

In addition, **newtype** declarations may also be used to define recursive types.

Class and Instance Declarations

According to Haskell 98 Report [PJ+03]:

> A **class** declaration introduces a new type class and the overloaded operations that must be supported by any type that is an instance of that class. An instance declaration declares that a type is an instance of a class and includes the definitions of the overloaded operations — called class methods — instantiated on the named type.

The first class that most of the programmers use in Haskell is Eq as the class of types that support equality and inequality operators. Here comes the declaration of Eq:

```
class Eq a where
  (≡), (≢) :: a → a → Bool
  x ≢ y    = ¬ (x ≡ y)
```

Now every type that implements Eq should define the equal and not equal operations. For instance, Bool that is the first case that comes to mind has the Eq instance of:

```
instance Eq Bool where
  False ≡ False = True
  True ≡ True  = True
  _ ≡ _        = False
```

This means that when we compare False with False or True with True, the result of equal operation is True, otherwise, it is False. Considering the definition of inequality in the class type, we do not need to implement that operation anymore because it is simply the negation of the result of equality.

Chapter 3

Tables Types and Table Editor Design

In this chapter, the formalization of a type for *Arbitrary-Dimensional Table* is introduced. Then, the design of the original project is briefly explained in section 3.2. In the end, in section 3.3, the new design of current project is introduced. The nested headers that are explained in the following chapters are added to our work that can be inspected in the last section.

3.1 Existential Types for Arbitrary-Dimensional Tables

The types that are developed to hold regular tables are described in the first part. In the next subsection, the skeleton of regular tables and how the folding on them is defined.

3.1.1 Regular Tables

A Regular Table, as explained by Kahl in [Kah04], has two parts, a *frame* and a *grid*. The grid is an n-dimension array of *grid cells* for an n-dimension table. The frame consists of two parts, *cell information* and *headers*. Cell information is "typing of cells, typing of cell semantics and a representation of the cell semantics function" [Kah04]. In the following chapter in section 4.1.1, we

will show an example of tables with information cells and how we use them to read tables. For an n-dimension table, there will be n headers each containing *header cells* in an array plus its typing and semantics information.

These letters are then introduced:

- g: grid element type
- h: header type
- ci: cell information type

to define *skeleton tables*. Kahl then uses the technique proposed by Okasaki in [Oka99] to make sure that the grid cells are conforming to the dimensionality of the table. Kahl introduces these data types to hold regular tables that follow.

```
data RT0 h ci g = RegTable {frame :: ci, grid :: g}

newtype NextRT rt h f g = NextRT (rt h (NextFrame h f) [g])

data NextFrame h frame = NextFrame {header :: h, subframe :: frame}
```

For more information regarding to the data types mentioned above refer to [Kah04].

3.1.2 Folding

Kahl introduces a folding class that should be used by tables to support folding over them:

```
class SkelFold t where
  foldSkel ::
    (ci → e → r) →
    (h → [r] → r) →
    t h ci e → r
```

Using the folding in foldSkel function, one can access the desired data in the regular table. Then Kahl defines a module RegTable that introduces a new data type to hold tables of arbitrary dimensionality:

```
data RT h ci g = forall rt o REGTABLE rt ⇒ RT (rt h ci g)
```

REGTABLE is a class defined to collect the classes that the three arguments of rt should implement.

3.2 A Modular and Generic Table Editor Design

The design of Kahl's tool has a module dependency graph as shown in Figure 3.1.

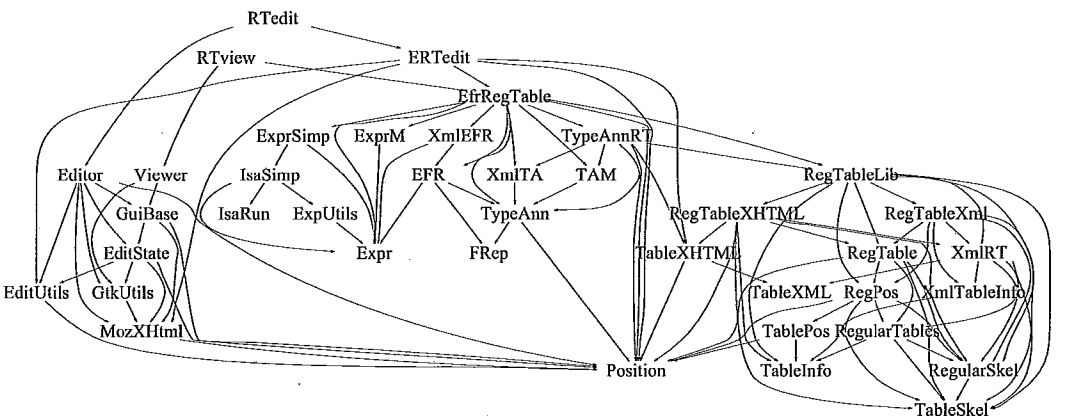

Figure 3.1: Module dependency graph of the old project

For more information about the modules refer to [Kah04]. The main parts that we need to focus on for the current project are GUI part and positions for editing the table.

Kahl explains:

> The leftmost cluster is a generic GUI-based editor (and viewer), currently built using Gtk2Hs, extended with a binding for the embedded Mozilla widget available in Gtk. Only those modules from which GtkUtils is reachable actually depend on Gtk2Hs. [Kah04]

This is the main part that we had to change in the current project and is going to be explained in chapter 6. Kahl also explains the use of XHTML for display which we have replaced completely by pure Gtk-based GUI.

According to [Kah04]:

> The use of positions for updating displayed parts — with type variable a instantiated to String, posGet is used by the editor to retrieve a string representation of an expression for initializing the entry field for editing, and posSet for parsing the edited string into a new expression to be inserted into the currently selected position:
>
> ```
> class PosEdit c a where
>   posGet :: c → ReadPos a
>   posSet :: (a → ReadPos a) → c → ReadPos c
> ```
>
> This is built on a simple special-purpose parser monad ReadPos:
>
> ```
> newtype ReadPos a = ReadPos (Pos → Maybe (a, Pos))
> ```
>
> ```
> instance (PosEdit c b, PosEdit b a) ⇒ PosEdit c a where
>   posGet c = (posGet c :: ReadPos b) ≫= (posGet :: b → ReadPos a)
>   posSet f = posSet (posSet f :: b → ReadPos b)
> ```
>
> Instead of using this multi-parameter class, all the generic parts of our code actually use single-parameter classes for specific type-constructor positions, e.g.:
>
> ```
> class PosEdit3_1 f where
>   posGet3_1 :: f a b c → ReadPos a
>   posSet3_1 :: (a → ReadPos a) → f a b c → ReadPos (f a b c)
> ```

However, we then replaced the position that is defined in Kahl's project with a new data type and omitted ReadPos monadic type. This is going to be explained in 6.1.

3.3 The new design including Nested Headers

Based on the code of [Kah04], we modified and added some parts to reach our goals. The new design of the tool as a module dependency graph is shown in figure 3.2 The new graph has been arranged to emphasize more clearly the devision into three main components that are also visible in Figure 3.1

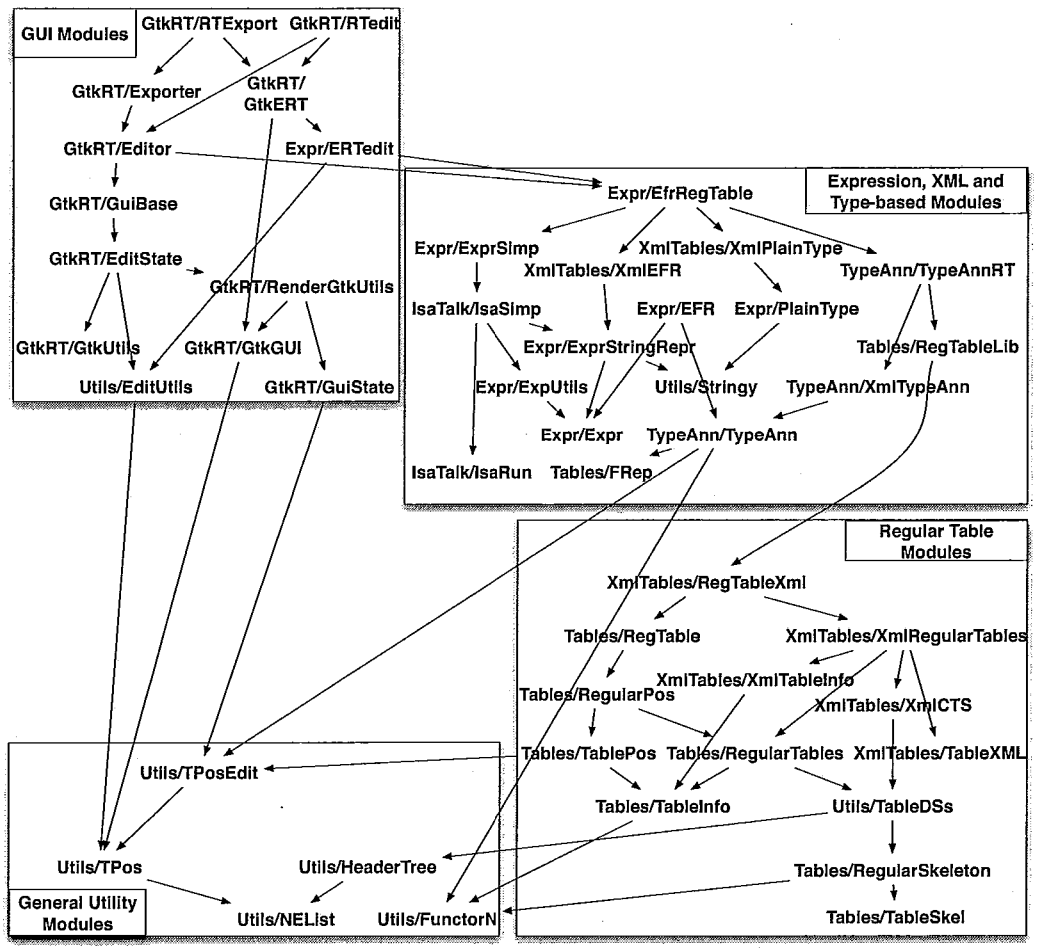

Figure 3.2: Module dependency graph of the new project

We made major changes to the project. First, we added a generic type constructor for the header type. This is required for replacing list of cell elements with nested headers, going to be explained in chapter 5. There are also two new data types introduced that are in modules NEList and HeaderTree. They

are all implementing the interface that comes in TableDSs module, that is explained in chapter 5 in section 5.5.

Second, ReadPos and the related module were replaced with a new parsing/editing module called TPosEdit that comes in a module of the same name. ReadPos was a simple parser monad for parsing the positions. It was based on the old position data type and implemented Haskell's MonadPlus that is an instance of Reader monad. Reader monad represents a computation for reading values from a shared environment. We are not going to provide a detailed explanation of them since they are not needed anymore in the project.

ReadPos accompanying PosEdit and Position modules provide the complete parsing and editing of positions. However, after changing the positions presentation, for simplicity, all of them are replaced with TPosEdit. This replacement is going to be explained in details in chapter 6 in section 6.1. It is based on a new data type for accessing positions instead of Pos that is called TPos.

Third, for the sake of portability and because of mozembed installation issues that are going to be discussed in detail in section 6.2, all the mozembed-dependent parts are replaced with pure Gtk methods. The names of the methods (GuiBase, EditState, GtkUtils and GuiState) have not been altered for the most part, only two new modules named GtkERT and GtkGUI are added which will be explained in section 6.2.

Chapter 4

TableTools in Action

After running the application, in the menu bar there are two menu items called *Table Functions* and *Header Functions*. The former includes the functions that manipulate all of the table and the latter are the functions which effect primarily the header structure. In this chapter we explain how the user should work with these features and interact with the application. This will also clarify the structure of our tables and nested headers and how we deal with them.

4.1 Table Functions

This menu contains functions for editing a table as a whole. In the explanation of the first menu item, we are going to describe how we use the information cells to interpret the table and extract the full tabular expression.

4.1.1 Toggle Info and Reading Table

Each table has some elements that gives us the information of how to interpret the table. By selecting this menu item it shows/hides the info buttons. They are also editable and by clicking on them the user can change their value.

A simple one-dimensional table looks like the table T_1

After toggling the info, we see the table T_1 with information in Figure 4.2

x > 0	x ≤ 0
2 · x	x ÷ 2

Figure 4.1: T_1, a simple one-dimensional table

Header 1	Cell Info
bool	nat
bool	bool
XXX ∧ YYY	f == XXX
XXX ∨ YYY	

x > 0	x ≤ 0
2 · x	x ÷ 2

Figure 4.2: Table T_1 with information

This table shows the semantics of the simple table and tells us how to read the table.

The first column on the right side, shows the information of grid cells. The first two elements show the type of the cell expressions and the type of table's function expression. In the illustrated example, *nat* shows that the cell expressions are of type natural numbers and *bool* says that the function is of type Boolean. The last element tells us what the final value of expression looks like: $f == XXX$ means that the function that is shown by the table is f and its value XXX is going to be replaced by grid cells. In the example we have two equations, $f = 2 \cdot x$ and $f = x/2$.

The left column contains the information for header cells, which are $x > 0$ and $x \leqslant 0$ in the T_1. The first two cells show the type of header cells themselves

and the type of their final expression when they are combined with the other expressions using the following information. The following two elements show the function that should be used to derive the final value. The first function that is $\wedge$ in the example should be applied to the expression and the lower level header cell value to get the corresponding grid cell. The last element shows the function to be applied to derived equations to form the final equation. From the information we can read the table to get the following equation for function $F(x)$:

$$F(x) = \begin{cases} 2 \cdot x & \text{if } x > 0 \\ x/2 & \text{if } x \leqslant 0 \end{cases}$$

by expanding the tabular expression as follows:

$$F := \{(x) \mapsto f \mid \begin{pmatrix} (x > 0 \wedge f = 2 \cdot x) \\ \vee\ (x \leqslant 0 \wedge f = x/2) \end{pmatrix}\}$$

This equation is developed using the equation 4.1 that we are going to explain.

Another example we have here shows a simple two-dimensional T_2 in Figure 4.3.

	x > 0	x ≤ 0
y == 0	x · y	y - x
y ≠ 0	x ÷ y	x + y

Figure 4.3: T_2, a simple dwo-dimensional table

After toggle info, we see the table T_2 in Figure 4.4.

We see that for each header there is a column of information. The first header is the one on the top of the table and second one is on the left side. Using the information and referring to the explanation of interpreting the tables that came for T_1, we get the following function for T_2:

$$F(x, y) = \begin{cases} x \cdot y & \text{if } x > 0 \wedge y = 0 \\ x/y & \text{if } x > 0 \wedge y \neq 0 \\ y - x & \text{if } x \leqslant 0 \wedge y = 0 \\ x + y & \text{if } x \leqslant 0 \wedge y \neq 0 \end{cases}$$

Header 2	Header 1	Cell Info
bool	bool	int
bool	bool	bool
XXX ∧ YYY	XXX ∧ YYY	f == XXX
XXX ∨ YYY	XXX ∨ YYY	

	x > 0	x ≤ 0
y == 0	x · y	y - x
y ≠ 0	x ÷ y	x + y

Figure 4.4: Table T_2 after showing the information

In order to get the complete expressions, we have to use the methods we have explained to use the information and read the table. Therefore, if we use the table T_F to define a binary function (considered as a set of mappings from pairs to results) F by writing

$$F := \{(x, y) \mapsto f \mid T_f\} \ , \tag{4.1}$$

then all the semantic functions that really are part of the full *tabular expression* T_F produce the following expansion of the above definition of F:

$$F := \{(x, y) \mapsto f \mid \begin{pmatrix} (y = 0 \wedge (& (x > 0 \wedge f = x \cdot y &) \\ & \vee \ (x \leqslant 0 \wedge f = y - x &))) \\ \vee \ (y \neq 0 \wedge (& (x > 0 \wedge f = x/y &) \\ & \vee \ (x \leqslant 0 \wedge f = x + y &))) \end{pmatrix} \}$$

4.1.2 Simplify Table

Simplify Table uses Isabelle theorem prover by supplying it with the expression that presents the table and the result would be simplified version of the complete expression in a new table. This feature requires an old version of *Isabelle* and will be updated in the future.

4.1.3 Simplify Cell

This function works the same way that Simplify Table does but just on one cell instead of the whole table. The result is the same table but just with the simplified expression in the selected cell.

4.1.4 Add Dimension

This function adds a single-cell header with empty value (shown by an underline). The current project supports up to two levels of headers, therefore, this operation does not have any affect on a two level table.

For example by adding dimension to T_1 of Figure 4.2 it creates the table shown in Figure 4.5.

Header 2	Header 1	Cell Info
·	bool	nat
·	bool	bool
_	XXX ∧ YYY	f == XXX
_	XXX ∨ YYY	

	x > 0	x ≤ 0
_	2 · x	x ÷ 2

Figure 4.5: Table T_1 after adding a dimension

The figure shows the table with information values. The new header that is created has a single empty header cell and the information values are all empty as shown in the figure.

4.1.5 Transpose

This function transposes the table, which means that header 1 would become header 2 and vice-versa. If the table has less than two levels, this operation fails.

If we click on transpose button when table T_2 is open, we get the table of Figure 4.6.

	$y == 0$	$y \neq 0$
$x > 0$	$x \cdot y$	$x \div y$
$x \leq 0$	$y - x$	$x + y$

Figure 4.6: Table T_2 after transposing

After applying the transpose operation, the header information is also switched since header 1 is now 2 and header 2 is 1.

4.2 Header Functions

The table editing operations triggered via this menu all require a header cell to be selected first. They are induced by operations on the header structure.

4.2.1 Move Out

This feature is simply the operation of swapping two columns or rows. However, "swap rows" does not explain with which neighbor the cell is going to be swapped with. If you apply "move out" on a header cell, then you know that it is going to move to the outer side of the header and be swapped with the next one. If the cell that is selected is nested, it will be moved out with all of its children. This feature should be applied just on a header cell and not on a grid cell. It is not clear for a grid cell with which cell it is going to be swapped

since it can have four neighbors.

Considering the table T_1 on page 19 (in section 4.1), after selecting $x > 0$ and choosing *move out* function, the table of Figure 4.7 will be generated.

$x \leq 0$	$x > 0$
$x \div 2$	$2 \cdot x$

Figure 4.7: Table T_1 after *move out* on $x > 0$

4.2.2 Duplicate

For the same reasons as for move out, duplicate should be done on a header cell. As the name says, this item duplicates a header and the corresponding grid elements. If the cell is nested, like move out, this operation will be done on it and its children. This means that the duplicated cell would have the same nesting as the original one.

Having the table T_1 again, after selecting $x > 0$ and this time choosing *duplicate* function, the table of Figure 4.8 will be generated.

$x > 0$	$x > 0$	$x \leq 0$
$2 \cdot x$	$2 \cdot x$	$x \div 2$

Figure 4.8: Table T_1 after *duplicate* on $x > 0$

4.2.3 Create Sibling

By selecting a header cell and then choosing this item, it creates a cell beside the selected header cell and a grid column or row under it, all with empty

values.

4.2.4 Nest

After choosing a header cell, Nest creates a child (nest) for that cell. It copies the cell element to the new cell which means it will have the same value as the parent and each of them can be changed as the user desires.

Going back to the same example, table T_1, we can apply nest on the cell $x > 0$. This is done by clicking on the cell and then choosing *nest*, the result is the table in Figure 4.9.

$x > 0$	$x \leq 0$
$x > 0$	
$2 \cdot x$	$x \div 2$

Figure 4.9: Table T_1 after *nest* on $x > 0$

The new cell that is created is under the old one and the expression inside is the same. The user can change it later and apply all the functions on it like duplicate (like the table in Figure 4.10) and so on.

$x > 0$				$x \leq 0$
$y > 0$	$y == 0$	$y < 0$		
		$z == 0$	$z \neq 0$	
$2 \cdot y$	$y \cdot x$	$z \cdot x$	$z \cdot x \div y$	$x \div 2$

Figure 4.10: Table T_1 after *nest* on $x > 0$ and *Duplicate* and more *Nest*

4.2.5 Nest with Expression

After choosing a header cell, the user should insert the expression they want to nest with in the entry field. Then, by pressing *Nest with Expression*, a nest of the selected cell is created with two children. One has the value given in the entry and the other one the negation of it.

On table T_1, choosing $x > 0$ cell and then inserting z in the entry field, by selecting *Nest with Expression* we get the table shown in Figure 4.11.

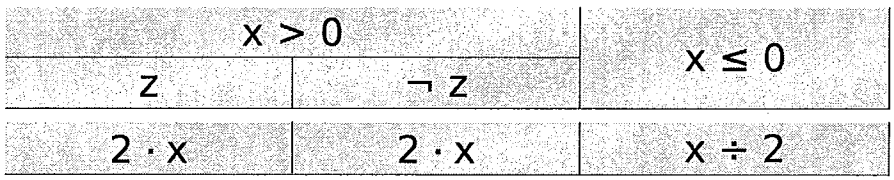

Figure 4.11: Table T_1 after *nest* on $x > 0$ with expression z

4.2.6 UnNest

As explained in the features of nested headers, by choosing the nested cell that we want to remove, it removes the cell and combines the expression of it by those ones of the children. Consequently, for a nest of n children, n cells would be created by using the semantics of the table that is inside table info for merging the expressions. If this feature is applied to a non-nested cell, then nothing would happen, it should be applied to the parent of a cell that we want to remove. If the user runs it on a non-nested cell, non-header or a cell with more than one child, the application will do nothing and show an error message.

If we have just one cell in the nest like the table we made after *nest* in Figure 4.9, by clicking on the parent and doing the *UnNest* we get one cell. Considering the table we got in 4.11, the nest has two elements. The result of choosing $x > 0$ and doing *UnNest* is shown in Figure 4.12.

We can see that the expressions are combined using the semantic that we

$x > 0 \wedge z$	$x > 0 \wedge (\neg z)$	$x \leq 0$
$2 \cdot x$	$2 \cdot x$	$x \div 2$

Figure 4.12: Table T_1 after nesting on $x > 0$ with with expression z and then applying *UnNest* on the same cell

get from information expressions. According to Figure 4.2, the first rule for combining level one's expressions is $XXX \wedge YYY$, which is the rule that is used here to merge the parent expression with those of the children in the nest.

This preserves the table semantics since $\wedge$ is associative. The expansion of tabular expression T_1 in the Figure 4.11 is:

$$(x > 0 \wedge ((z \wedge f = 2 \cdot x) \vee (\neg z \wedge 2 \cdot x))) \vee (x \leqslant 0 \wedge x/2) \qquad (4.2)$$

and expansion of the tabular expression in Figure 4.12 is:

$$((x > 0 \wedge z) \wedge f = 2 \cdot x) \vee ((x > 0 \wedge (\neg z)) \wedge f = 2 \cdot x) \vee (x \leqslant 0 \wedge x/2) \quad (4.3)$$

As we can see, the expression 4.2 is equivalent to 4.3 based on the associativity and distributivity of $\wedge$. This means that if the function we apply was another function that is not associative like $\wedge$, UnNest operation would not preserve the table semantics. This means that the correctness (semantics preservation) of UnNest depends on algebraic properties of the functions in header info.

4.2.7 Merge

As explained in the features, by selecting a header cell, it merges that cell with the next cell. However, those cells should have the same nesting and values as their children, and the grid cells corresponding to those cells should be the same. This operation checks the semantics of the table from table info and combines the expressions based on the semantics.

Working on T_1 after we nested using expression z previously, now we have a nest with two cells that have the same grid values; Therefore, we can merge them. We do this by clicking on the cell z of the table in Figure 4.11 and choose *Merge* from the menu. As explained in UnNest function, it checks the semantics written in the information cells that says we should use $XXX \vee YYY$ to merge the header cell expressions. The result would be the table shown in Figure 4.13.

$x > 0$	$x \leq 0$
$\neg z \vee z$	
$2 \cdot x$	$x \div 2$

Figure 4.13: Table T_1 after nesting on $x > 0$ with with expression z and then applying *Merge* on the cell with expression z

The expanded expression of T_1 in Figure 4.13 is:

$$(x > 0 \wedge ((\neg z \vee z) \wedge f = 2 \cdot x)) \vee (x \leqslant 0 \wedge x/2) \tag{4.4}$$

Expression 4.4 is equal to the expressions 4.2 and 4.3. In can be derived from any of the expressions by calculating using the rule of associativity and distributivity for $\wedge$.

4.2.8 Delete

Delete is designed for removing a header cell with the corresponding row or column. The header cell cannot be the last cell in the nest. If this is the case, it should be removed (without the related grid cells) by un-nesting the parent header cell.

A simple case of delete is when we select a single cell like $x > 0$ in T_1 and then choose *Delete*. The result is shown in Figure 4.14. A more complicated case is when the cell we choose is nested. For example, after we had *Nest with Expression* on $x > 0$, the cell was nested with two children. By choosing that

cell and then *Delete*, we get the same table in Figure 4.14 since it removes the cell and all its nested elements.

$x \leq 0$
$x \div 2$

Figure 4.14: Table T_1 after *Delete* on $x > 0$

Chapter 5

Types for Nested Headers

In this chapter we are going to discuss the reasons why we needed to have nested headers and how we made our decisions on the structure of nesting. Then we have an abstract discussion of the data structure for nested headers and the exchange format we have employed for storing tables. We also explain the new data types we have introduced and used for implementing nested headers. Furthermore, we explain the features we can achieve with nested headers and how we install new data structures in the old project.

5.1 Nested Headers Design and Motives

By studying the projects previously documented using tables, it shows the need for having nested headers in table tools. Therefore, we decided to add the potential of having nested headers in our tool.

We had to modify the data types that we had in our tool before adapting nested headers. First, we wanted to ensure via the type system that:

- Tables are non-empty
- Tables can have arbitrary dimensions
- Table grids have exactly the dimensionality dictated by the headers
- Nested headers are possible

- We can keep the semantics of nests separated from table's general information

To achieve all of the desired properties, we could engage a variant of HList, introduced in [KLS04]. HList is a *Strongly typed heterogeneous collections*; as Kiselyov explains, "A heterogeneous collection is a datatype that is capable of storing data of different types, while providing operations for look-up, update, iteration, and others."

However, using a variant of HList would make the data types too complicated for our purpose. Besides, we would have had to alter not only the header types but also the grid structure for having all the above-mentioned requirements. Therefore, we decided to use the previous data types and just change the structure of headers to hold nested headers. We also decided that we would not add more information to the table for interpreting the semantic of the nested headers. Thus, we have to use the semantics of the same header for its nested headers.

Using the old structure and adjusting the header parts to adapt nested headers makes it easier for us to use the pre-existing modules with small modifications. Not having separate semantic information for nests is not a serious issue since the practical examples we have studied do not need more complicated structure. Some of the practical examples we studied were the tables with nested headers that were used in [Pan06, LFM01, TyWP08, FPT07] and a confidential project kindly provided by Dr. Alan Wassyng. Still, we can have non-empty nested headers in arbitrary-dimensional tables.

5.2 Exchange Format

In order to save and to load tables to and from files, the XML format is chosen in this project for storing tables in a hierarchical and organized order. Having XML files, there is a need for a unified system to check the validity of them. For this purpose, XML supplies DTD (Document Type Definitions) to define, parametrize and regulate our files. DTD allows us to define legal building blocks of an XML document by defining type, expression and function representations and how they can be connected to one another.

Consequently, in [Kah04], Kahl has created a DTD file named RegularTable.dtd and used the HaXml package [WR99], which is a collection of utilities for working with XML documents, for parsing and generating. But, "since parametrized DTDs are not supported by HaXml, and no DTD can express the regularity constraints for *arbitrary*-dimensional tables, we had to add some robust parser combinators for the HaXml document contents datatype." [Kah04].

For replacing headers with nested headers, we had to change the data structure of headers that we had employed. Instead of using lists, we needed a more advanced data structure to hold nesting. This could be simply done by using a Decision Tree-like structure. Before, the structure was just a list that was defined in DTD as follows:

```
<!ELEMENT header (%types;, addH, combine, cell+)>
```

where we have the following definitions:

```
<!ENTITY % expr "#PCDATA">
<!ENTITY % function1 "#PCDATA">
<!ENTITY % function2 "#PCDATA">
<!ENTITY % types "elemType, semType">

<!ELEMENT cell (%expr;)>
<!ELEMENT addH (%function2;)>
<!ELEMENT combine (%function2;)>
```

It holds a header including the typing and its functions for interpretation of the header elements. As we see, the cells are defined as cell+ which means it is just a simple non-empty list. Then we changed the DTD file by changing the header element to:

```
<!ELEMENT headerChunk ((cell, headerChunk+) | cell)>
<!ELEMENT header (%types;, addH, combine,  headerChunk+ )>
```

We have changed the list of cells in a header to a list of headerChunks. Each header has a non-empty list of chunks, which can be just a cell or a cell together with another list of chunks. An example of a nested header tree can be

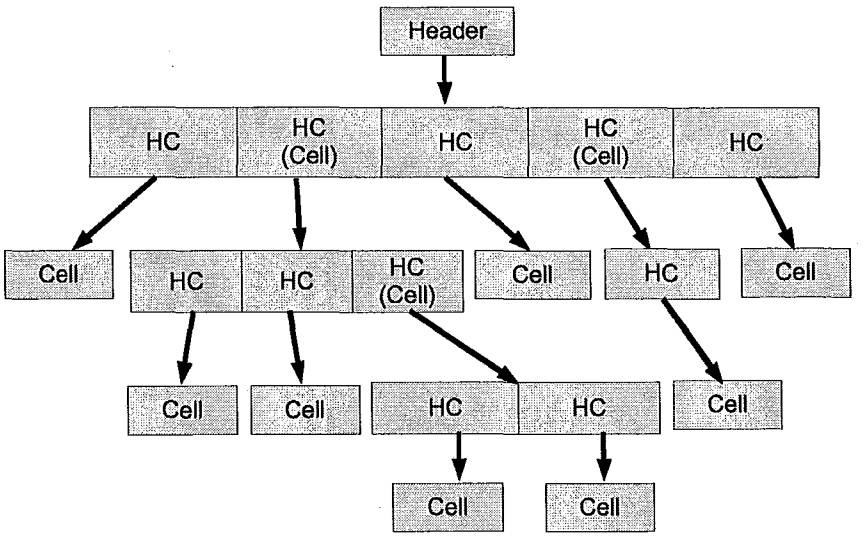

Figure 5.1: An exmaple of Nested Headers, used in DTD

seen in Figure 5.1

We could also use other implementations in the DTD, but we decided to choose the one mentioned above for the direct compatibility with the data type that is defined in the project. The data type is going to be explained in section 5.4.

5.3 NEList (Non-Empty List)

This is a datatype for storing non-empty lists. These kinds of lists are used later in the implementation of header trees. In some parts, we have used the online discussion at:
`http://www.haskell.org/haskellwiki/Non-empty_list`

The data type definition of Non-Empty List:

```
data NEList a = NELCons a [a]
```

NEList was then used to replace simple lists for the headers. Not having a header would change the dimension of the table but if we consider the empty header we are counting that header as one level. Therefore, if the header exists for a level of tabular expression, it cannot be empty.

Guaranteeing non-emptyness of these lists was previously achieved by datatype abstraction, i.e., by limiting the functions available to construct tables from outside the library. Using a type of non-empty lists instead guarantees non-emptyness of these lists via the type system, which has the advantage of adding internal consistency guarantees.

For this purpose, during the first step of changing the header structure, we replaced the list in header type with NEList, which needed a complete propagation of the change to all the modules that work on the header structure (RegularTables, RegularPos, RegTable and more).

For being able to move to nested headers, we abstract the list type constructor away into an additional parameter hs by changing the Header type as follows:

```
data GenHeader hs ty e f2 = Header
  {headerInfo :: HeaderInfo ty f2
  ,headerCells :: hs e
  }
  deriving Show
```

We could re-introduce the original type name as a synonym:

```
type Header = GenHeader []
```

in which [] would be replaced with NEList to move to replacing the structure of headers (list of elements) with non-empty list as follows:

```
type Header = GenHeader NEList
```

5.4 Header Tree

In a nested header of type a, we will have one or more elements. Those elements might consist of a single element of type a, or another list of elements

that would be defined recursively. In order to keep nested headers, we need to introduce a decision-tree-like data structure. Header Tree structure is introduced here to satisfy this purpose that is a Haskell **newtype**. A HTree (*Header Tree*) consists of a non-empty list of *Header Chunk*s (HC is acronym for Header Chunk). Each header chunk then holds just one single element defined by Leaf, or one element and a subtree that is defined by Nest.

Our Header Tree's implementation in Haskell is:

```
newtype HTree a = HTCons {unHTree :: NEList (HC a)}
data HC a = Leaf a
  | Nest a (HTree a)
```

Here we have used the data type of NEList that we defined in the previous section.

By having HTree, we can create the nested header type using the following type that is the current Header we use in the project.

```
type Header = GenHeader HTree
```

5.5 Table Data Structure

Concrete type constructors that can serve as first parameter hs to GenHeader will need to expose a common interface to enable the functionality of the tool operating on the resulting tables. This interface is collected in the constructor class TableDS. Using TableDS class gives us the flexibility and simplicity of changing the structure. Normal lists are used for grids, and trees for header cells.

HSUpdate (Header Structure Update) that is going to be used later is defined as follows:

```
data HSUpdate h = HSUpdate h (forall b o [b] → [b])
```

This data type is used to keep the content of the changed header after imposing the operation. Besides, it keeps the operation that needs to be applied to the

rest of the table, which applies the operation on the grid cells corresponding to the changed header elements. For example, if we swap two elements of a header of a one level table, the grid elements under those two elements should be swapped.

HSUpdate has a generic type using forall definition to be independent of the type of the corresponding grid. This means that it works for any level, that is for level 1 it functions on a list, level 2 on a list of list and so on.

```
class Applicative hs ⇒ TableDS hs where
  type DSIndex hs
  dupTS, swapTS, nestTS, delTS ::
    DSIndex hs → hs a → Maybe (HSUpdate (hs a))
```

DSIndex is introduced to keep the type of the index of that particular instance of TableDS. For lists and non-empty lists we just need an Int for index, but for trees we have to keep an Int for each nest of the tree, which means DSIndex would be a list (or more specifically a non-empty list) of Int. The functions written above are for the features that were introduced in the features of header trees.

The rest of the functions that should be implemented by a new data structure are as follows:

```
atTS :: hs a → DSIndex hs → Maybe a
```

The Nothing return values of atTS, swapTS and dupTS indicate that the operation in question was not possible because of illegal or inappropriate indices.

```
updateTS :: DSIndex hs → (a → a) → hs a → hs a
leavesTS :: hs a → [a]
```

leavesTS returns only the entries closest to the grid if applied to a tree. Otherwise, if applied to a list, it returns the whole list.

```
fromNEListTS :: NEList a → hs a
```

fromNEListTS produces a flat header structure from a NEList.

```
toHTree :: hs a → HTree a
```

toHTree embeds the data structure into the most general format we support.

```
fromHTree :: HTree a → hs a
```

Chapter 6

New GTK-based GUI Implementation

6.1 Position Parsing and Editing

The old approach to positions was constrained by the fact that positions needed to be parts of URLs. Instead of that string encoding, we used an algebraic data type to have a more elaborate approach to position parsing and editing of tables.

6.1.1 TPos

Previously, positions were defined as a String that was represented in data type Pos, which holds a URL-encoded position's address. We replaced this definition by introducing a new algebraic data type called TPos.

```
data TPos = CIPos Int CellInfoPos
  | HIPos Int Int HeaderInfoPos
  | TPos Int TablePos
  | EmptyPos
data TablePos = HeaderPos Int (NEList Int) | CellPos [Int]
data CellInfoPos = CellType | SemType | CellSem
data HeaderInfoPos = HeaderType | ResultType | AddH | Comb
```

The first Int is the dimension of the table (how many levels the table has), in

order to make it compatible with the old Pos data type. In the old version we kept the dimension of the table in the URL that is representing the position.

Header and Cell information positions (HEaderInfoPos and CellInfoPos) have just constant values. For header information positions, we have to keep the level of the header it is representing as well, and that is the second stored number.

In the TablePos we have HeaderPos and CellPos and each has its own indices. For cell or grid position, we keep a list of Int, and each integer is the index of that cell in that level of the multi-dimension grid. Two arguments of header position are the level of header and the index of it. Because we have moved from a simple list to a tree, a single Int would not be enough for the index of header cell. Consequently, the index would be a non-empty list, representing the position in that particular tree (the tree of the intended header dimension).

6.1.2 TPosEdit

In the old design, three modules held the responsibility of reading and editing in a position. These three modules were ReadPos, PosEdit and Position. ReadPos used Reader monad transformer (which would be ReaderT TPos now). However, it was convenient to omit monad transformer and use TPos explicitly.

Now the functionality of all the three modules has completely moved to TPosEdit for getting (setting) a value from (to) a position in the table.

We define a new class TPosEdit:

```
class TPosEdit c a where
  tPosGet :: c → TPos → Either String a
  tPosSet :: (a → Either String a) → c → TPos → Either String c
```

"TPosEdit c a" means a c may contain a at some positions. c is the container and a is the contents at the position.

For the means of error propagation and handling, Either String a is returned instead of Maybe a. If an error occurs and we cannot get or set a value, we return the corresponding error message in a String.

6.2 GtkERT

The old graphical interface of *Table Tools* was implemented using Mozilla browser embedded in the Gtk GUI, called *mozembed*. This had been done by creating a XHTML element from a table and showing it on a Gtk *Widget*. However, this solution was not a very helpful one, since the installation of *mozembed* part of Gtk is not very easy. In order to have a portable tool that everybody can use it on a system with just the basic *gtk2hs* package installed, we had to avoid unsupported components.

GtkERT is the main module that is the Gtk GUI generator for the Expression-based Regular Table editor-State (ERT) based on TypeAnn and Expr, that is defined in ERTedit module. . The current GUI consists of buttons, each representing a cell of the table. Buttons allow us to navigate through them and interaction with them is easy.

We divide this module into three major parts that we are going to explain:

- Making buttons from tables
- Adding TPoses to the buttons
- The main function called toGtkGUI

6.2.1 Make Buttons

First, we designed this function to be separated for each dimension of tables because they have different levels of headers. However, it was later divided into two parts: the first one for creating just the grid cells' buttons and the second one for generating buttons of header cells.

The signature for the function for creating grid cells' buttons is:

```
makeGBtns :: ERTS → [[IO Button]]
```

which gets an Editor State for Regular Tables based on TypeAnn and Expr, and returns a list of list of Gtk buttons. This function has the same mechanism for every dimension of tables, except for more than two dimensional tables. We are not supporting more than two levels yet because having nested headers

with two levels we can reach our goals and show desired tables in different projects. In case more levels are needed, we only require to change the graphical interface part to fulfill this need.

makeHBtns1 and makeHBtns2 make header buttons for level one and two of the table; they take the dimension of the table as an input in addition to the table itself. The return would hold a list of values for each header cell. These values contain position of the cell, button widget of it and a list of Integers that would have the format of [i, width, j, heigth]; these numbers are needed for showing them on the screen according to their place in the graphical interface and their sizes. At the end of the output we put width and length of the header tree. These are needed again for GUI implementation.

How these functions are declared is:

```
makeHBtns1 :: ERTS → Int → ([(TPos, IO Button, [Int])], Int, Int)
makeHBtns2 :: ERTS → Int → ([(TPos, IO Button, [Int])], Int, Int)
```

6.2.2 Add TPos

addTPoses is the function that adds positions to buttons that represent grid cells. It finds out based on the dimensionality of the table how the buttons should be shaped in the list of list. It is currently functioning for up to two levels of tables, therefore, it returns just a list of list of Buttons and their TPos.

This function's declaration is:

```
addTPoses :: Int → [[IO Button]] → [[(TPos, IO Button)]]
```

Note: There is no need for having extra functions for adding positions to header cells or info cells because positions are added inside the functions that create those buttons.

6.2.3 Main Function

toGtkGUI is a replacement for toMozXHtml to exclude any moz (previous *mozembed* element used) dependencies. toMozXHtml was a function that used

to generate mozilla XHTML element.

toGtkGUI is the main method of typeclass GtkGUI. GtkGUI gives us the flexibility to be able to replace current GUI with another GUI by creating another instance of this class and implementing toGtkGUI. The current implementation is as follows with the explanation.

```
instance GtkGUI ERTS where
  toGtkGUI erts = do
    let level = skelLevel ∘ unRegT ∘ unTypeAnnRT $ ertsTable erts
    let (bss, phis_WHs) = case level of
      0 → (makeGBtns erts, [([], 0, 0), ([], 0, 0)])
      1 → (makeGBtns erts, [makeHBtns1 erts 1, ([], 0, 0)])
      2 → (makeGBtns erts, [makeHBtns1 erts 2, makeHBtns2 erts 2])
      _ → error "makeBtns_n not defined yet"
    let piss = case (ertsInfoVis erts) of
      True → makeInfoBtns erts level
      False → [[]]
    let pbss = addTPoses level bss
    return (level, (piss, phis_WHs, pbss))
```

First bss (list of list of buttons) is created using makeGBtns for the grid of the table. makeBtns is the same for dimension up to two that is implemented currently, because we have maximum of two dimensions in the table of grid. phis_WHs is a list that contains elements of type: list of position, header (button), indices plus width and height of that header level.

Then piss (list of list of positions and info buttons) is created if ertsInfoVis is set to True. There is no need for an extra function to add positions since it can be done at the same time.

Then pbss (list of list of position and buttons) is created by adding positions (TPos) to the previous buttons (the Grid elements).

Return would include the dimension of the table, info buttons with their positions, header buttons including positions and the supplementary information (that are needed for drawing the headers), and grid buttons with their positions.

6.3 RenderGtkUtils

In section 6.2 we explained how we generate Gtk buttons representing the actual table. In the EditState module[1], we use the return values of the toGtkGUI function and create the visual interface. We also make it ready for rendering on the screen. For this purpose, we have to put buttons in an elegant view that is understandable and easy to interact with. We have used table widget of Gtk to arrange buttons to satisfy our needs.

6.3.1 Make Widget Table

The first function we have here is for creating the Gtk table widget that holds normal un-nested headers. By taking the Gtk table widget, header one and header two sizes, list of list of buttons and their position values, and dimension of the table, this function returns the table to be shown on the screen. We need header one and two's sizes because we are putting grid cells in the same Gtk table widget as we are putting headers. Therefore, we need to know the location of the row and column where to insert grid cells.

It can be used for both grid cells and info cells. Currently, all functions are designed for up to two levels of trees. For more levels (in case they are needed) we have to change them to generated the relevant interface. The declaration of this function is as follows.

```
makeWgtTable :: Gtk.Table → Int → Int
  → [[(TPos, IO Button)]] → Int
  → IO (Gtk.Table)
```

6.3.2 Make Header Widget Table

For adding header cell buttons to the Gtk table widget, we added makeHWgtTable. This function gets the Gtk table widget, header one and two's sizes, two lists

[1]Will be explained in 6.4

that holds level one and two buttons and their positions and indices that explains their coordinations in the Gtk Table widget. The last element would be the dimension of the table and the function adds buttons to the Gtk table widget and return the table widget. Indices of buttons would be the form of [i, width, j, height] to show the row and column of them in their corresponding header tree and their sizes. These buttons have variable size in the Gtk table widget's grid because they might have different number of children or none. Their size varies depending on the number of children, the number of parents they have, height and length of the tree.

makeHWgtTable's signature is almost the same as makeWgtTable except the list of buttons and their information.

```
makeHWgtTable :: Gtk.Table → Int → Int
  → ([(TPos, IO Button, [Int])], [(TPos, IO Button, [Int])]) → Int
  → IO Gtk.Table
```

6.3.3 Make Button Pressed

makeButtonPressed is used to create the action related to each of the buttons when they're pressed. It manages how to interact with the entry field (update the entry field when pressed and store the button in GuiState[2] for updating according to the entry field). This method receives a list of TPoses that are related to buttons of the table. When a button is pressed, it's corresponding position is added to the GuiState. After the entry field is completed, the position is read from GuiState and the value of that position is updated.

```
makeButtonPressed :: (GtkGUI t, TPosEdit t String)
  ⇒ GuiState t → Table → [TPos] → IO ()
```

6.4 GuiState and EditState

In this section, we are going to explain how we create Gtk tables from buttons and show them on the screen and interact with them. This is done by

[2]GuiState is going to be explained in section 6.4

using toGtkGUI of GTKERT as mentioned before and apply the methods of RenderGtkUtils to create the elegant graphical interface the users need. These modules existed before this thesis. Our contribution was replacing the old mozembed containers with virtual box widgets and adding elements for implementing undo, redo and consistent mode.

6.4.1 GuiState

In order to hold the visual information in a state that we can access in different functions and manipulate, an IORef data type is used. IORef is a mutable variable in the IO monad that helps us to define our GuiState in a simple manner and to be able to work with it throughout the project.

We declare GuiState of variable t as a synonym for the IORef we need. GuiState can be used for any data type desired, and we use it to store tables, which means that t is going to be initialized with our table data type.

```
type GuiState t = IORef (GuiRecord t)
```

We then keep all the information of the GUI that we need to in GuiRecord. This information include a vertical box VBox that holds the buttons of the table that is shown right now, the entry field, the tabs that are open and the active tab's identification value.

```
data GuiRecord t = GuiRecord
  {
  cntnr :: VBox
  , ent :: Entry
  , active :: Maybe TabId
  , tabs :: Tabs t
  }
```

Tabs t is a type synonym for a mapping of TabIds (which are simply strings) to EditContainer of t.

```
type Tabs t = Map.Map TabId (EditContainer t)
```

EditContainer is a new data type that is defined as follows:

```
data EditContainer a = EditContainer
  {ecContents :: a
  ,ecCntssList :: [a]   -- for implementing undo and redo
  ,ecCntssIndx :: Int
  ,ecButPos :: Maybe (Button, TPos)
  ,ecCurFile :: Maybe FilePath
  ,ecNotify :: a → IO ()
      -- to be called when content changes
  ,ecExternalUpdated :: Maybe a
      -- contains last update by external owner
  ,ecSafeMode :: Bool
      -- whether the unsafe functions should be available or not
  }
```

Button and TPos are stored for storing the element that is being edited. After the entry value is entered, the button from container will be changed. ecCntssList is for storing the previous and/or next contents before and/or after performing actions. By using that, we can have *Undo* and *Redo*. When we undo some actions, to use undo we need to know where we are on the list. For this reason, we have added ecCntssIndx that is the index we need.

6.4.2 Update Render

The Main function here that is in EditState module is updateRender. It is a new variation of the old updateRender that generates the interface without mozilla embedded elements. It uses the previously explained methods in 6.3 to generate buttons and pack them in a Gtk table widget and then pass the table widgets to another function to render them on the screen.

updateRender has the following steps:

- Getting the content of the active tab from GuiState and pass it to toGtkGUI as the table we want to show on the screen. The data we need to render would be the return values of this function.
- Create a new Gtk Table for info buttons and grid and header buttons, the size of which is decided from the size of buttons lists.
- Then use makeWgtTable and makeHWgtTable to add the buttons to the Gtk widget tables.

- Call makeButtonPressed to add positions of buttons to perform the update action when the entry field is entered for a button.
- Call renderDirect that is defined in GtkUtils to render the widgets and show on the screen.

For the complete code of this function refer to appendix A.1.

Chapter 7

Table Tools Manual

This chapter covers the process of installation and all the information users need to know to make the program run and work with it. Some features are already explained in chapter 4 and we are not going to explain them again, therefore, they are going to be reviewed along with the other menu items. Finally, the navigation in the new graphical interface is described.

7.1 Installation and running the tool

Each application has some requirements before installation. These sections covers these requirements followed by the procedure of compiling and installing the tool. Ultimately, the short guide to execute the tool is explained.

7.1.1 Requirements

In addition to GHC (Glasgow Haskell Compiler) that we use to compile our project, the following packages are needed for this purpose:

- *HaXml*: HaXml is a collection of programs and libraries that are for parsing, filtering, transforming, and generating XML documents using Haskell. We use this package for saving and reading the tables to and from XML files. Besides, in the old structure, HaXml were used for generating XHTML elements that were shown in a mozembed widget. This package is available online at http://www.haskell.org/HaXml/

- *Gtk2Hs*: It is a library that is a wrapper for Gtk+ for Haskell. Gtk+ is a toolkit for graphical user interface creation. Gtk+ is a cross-platform toolkit that originally designed for X Window System. It makes our project available on different platforms. Gtk2Hs is available online at http://www.haskell.org/gtk2hs/overview/

- *Isabelle*: Isabelle is a generic proof assistant developed at University of Cambridge (Larry Paulson) and Technische Universitt Mnchen. We use Isabelle for simplifying table and cells and other reasonings about tables.

All of the packages that we use are Free softwares licensed under the GNU Lesser General Public Licence (LGPL). We could not use any other compiler for this project because Gtk2hs requires GHC.[1]

Gtk2Hs itself has some requirements as follow:

- GHC version 6.10.3: It is originally developed using this version but they also support some other versions of GHC. Building from source requires version 6.6 or later. For more information refer to their website.

- Gtk+ version 2.0 or later.

- Other packages: happy, alex, libglib2.0-dev, libgtk2.0-dev, libgmp3-dev, autoconf, and libtool.

For supporting Mozilla rendering engine widget (mozembed), XULRunner, FireFox, SeaMonkey or Mozilla should be installed and some configurations should be made. It is also not supported on Windows systems. These complications forced us to exclude mozembed parts and make it a pure Gtk+ project using Gtk2Hs.

7.1.2 Installing

Non-Haskell packages can be installed using *aptitude* (or apt-get) in debian-derived Linux distribution and/or *port* in Mac OS X that is from MacPorts

[1]HaXml copyright available online at: `http://www.haskell.org/HaXml/COPYRIGHT` and Haskell's Copyrite at `http://www.haskell.org/haskellwiki/HaskellWiki:Copyrights`

Project[2]. Haskell packages can be installed manually or by using *cabal* application for the ease of installation.

One should follow these steps to install the tool:

1. Installing GHC by compiling manually after downloading from their website or use the pre-compiled binary files.
2. Installing Gtk+ and other Gtk2Hs's requirements as explained before.
3. Installing Gtk2Hs by using binary files or compiling the source code. Compiling manually is highly recommended for Gtk2Hs.
4. Installing HaXml by using Cabal (or other application installers you might use in Mac Os X or other platforms) or downloading the source code and compiling it.
5. Installing Isabelle by following their own instructions (downloading the source code and compiling it or using the available binary files).[3]
6. Installing the Table Tool by typing the following command in the root folder of the code in the terminal: "make RTedit".
7. (optional) Installing the validating tool by the command: "make RTvalidate"

7.1.3 Running the tool

The application can be started by running *./RTedit* from terminal in the root folder. The user can also supply the address of a table's XML file as the input to open it directly. For validating a XML file, the optional installation step should be done. Then, by running *RTvalidate* and giving the address of XML file it tells the user whether the XML file has any problems or not.

7.2 Menus

Here we point out the menus the user sees in the program and what the function of each one is.

[2]MacPorts' website is `http://www.macports.org/`

[3]Available online at `http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html`

7.2.1 File

In the File menu, there are some items that are related to file-manipulation functions.

- **New:** By selecting New, a new empty zero-level table is created. The empty table would have also empty info table, therefore, requires the user to add info values as well as the grid element values. If the user needs to have empty grid table with regular function table template, the next menu item is designed for that purpose.

- **New Template:** In case the user does not want to change the info expressions and have the regular table semantics, they can use the pre-designed templates. This menu has some sub-menus that are representing different templates that one might require. The current templates are *New 0*, *New 1* and *New 2*, each creating a new table of n-dimension with the default (the common) semantics.

- **Open and Save:** There are two items for saving to a new or pre-existing file and opening from a pre-saved file. It needs the user to type the complete name to save the file to.

- **Export:** This option is for exporting the actual representation of the table to a PNG file. For a better quality, one might need to change the font's size of the X11 to have a bigger fonts.

7.2.2 Edit

The menu item for simple editing features. Currently it has two crucial features for editing the table, Redo and Undo.

- **Undo:** After performing actions, one might want to undo the action (any of the functions that are going to be explained in 4.1 and 4.2). If one considers the flow of actions as a list, undo does the job of going back in the list. When there is no previous action to go back to, it does not work.

- **Redo:** This is the reverse of undo. When the user performs an action and chooses undo one or many times, they go back in the list of actions. By pressing *Redo* they go forward on the list. When it reaches the last action performed, then choosing this feature does not make any changes.

7.2.3 Tabs

For supporting simultaneous table editing, tabs are designed to open more than one table in the application.

- **New:** We can create a new tab by clicking on this menu item. After this operation, the user will see the name of tabs in *Switch*. A new tab will have the name starting with "Tab" accompanied by a number that shows the number of the new tab. The first new tab will be "Tab1", the second one "Tab2" and so on.
- **Switch:** If there are more than one open tab in the tool, Switch will display a list of names that shows the name of the open tabs. By clicking on each of them, the view will switch to that particular selected tab.

7.2.4 Table Functions and Header Function

The next two menus are for editing the whole table or particular cells. The functions included under these menus are already explained in section 4.1 and 4.2.

7.3 Navigation and Editing

In this tool, the navigation for editing is very convenient. The user can use mouse to click on a button (a cell element) and then the entry field will be ready for editing the cell. Beside the computer mouse, it is possible to use the keyboard and move in the table. When the focus is on a button, by pressing "Enter" it has the same effect as clicking with the mouse does. After changing the entry field, by pressing enter, the value of the last chosen cell will be updated.

Chapter 8

Conclusion and Future Work

In this thesis, we presented a graphical tool to support tabular expressions. We are confident that our application surpasses any other tool by having a cleaner view of the syntax and semantics of tables and simplicity in user interactions.

Programming in Haskell, using the modular design that we explained in chapter 3 makes this project easily expansible based on a strong infrastructure. Consequently, many powerful extensions and practical backends can be added with a little coding. It also helps the developers to solve any possible issues with a small effort which makes the process of expansion and debugging faster than starting the project.

Working with nested headers in tabular expressions is usually very error-prone and complicated. Our simple tool using the nested header structure that came in chapter 5 using the portable GUI, explained in chapter 6, supplies a user-friendly interface that simplifies the process of requirement analyze. This tool turns out to be a very promising avenue to develop a comprehensive work that can be used in the industry and in academic circles as well.

Future Work

In terms of future work, there are several avenues to be considered:

- We have not devoted too much effort in the development of expressions

data type. The top priority issues that we currently face is to write a more sophisticated data type to replace the existing Expr module. Based on the need of the job to be undertaken, other complicated expressions could be added.

- For a better and faster interaction, shortkeys will be very decent. Shortkeys should be added to the functions that we have now and file menus for the users who do not want to regularly use the mouse and prefer to navigate and edit tables with keyboard.

- Currently, exporting the tables is done by automatically capturing the screen that the table is drawn on. A more elegant export system is needed to save the tables in other formats, in particular postscript.

- Using Isabelle, by completing the available interface, we can achieve theorem proving and verification systems, property checker and invariant generator. Two of the major properties that need to be checked are completeness (coverage) and determinism (disjointness) of tables.

- Our table makes it very easy to add backends such as VHDL, Oracle and assembly code generators. In the time of need for any backend, we can append the desired backends with a few lines of codes.

- The infrastructure of our tabular expressions is capable of supporting tables arbitrary dimensions. Even though we have not faced practical examples with more than two dimensions, one might need that feature. For this purpose, the graphical interface generation should be altered to adapt arbitrary dimensional tables.

Appendix A

Source Codes

Here are some of the source codes that we explained throughout the thesis.

A.1 Update Render Method

Here is the code that can be found in "EditState.lhs" for updating the render that is explained in 6.4.

```
updateRender :: (GtkGUI t, TPosEdit t String) ⇒ GuiState t → IO ()
updateRender r = do
  gr ← readIORef r
  mec ← guiGetActiveTab r
  case mec of
    Nothing → return ()
    Just ec → do
      let cntns = ecContents ec
      guiAddCntsToL r cntns
      res ← toGtkGUI cntns
      case res of
        Left err → do
          md ← messageDialogNew Nothing []
            MessageWarning ButtonsClose err
          onResponse md (λx → widgetDestroy md)
          widgetShowAll md
        Right (l, (pi, ph, pb)) → do
          let (level, (piss, phis_WHs, pbss)) = (l, (pi, ph, pb))
          let ((phis1, phisW1, phisH1) :
```

```
    (phis2, phisW2, phisH2) : []) = phis_WHs
  btnsTable ← tableNew (phisH1 + (length $ head pbss))
    (phisH2 + length pbss) False
  infoBtnsTable ← tableNew (length $ head piss)
                           (length piss) False
  t1 ← makeWgtTable infoBtnsTable 0 0 piss level
  t2 ← makeHWgtTable btnsTable phisH1 phisH2
    (phis1, phis2) level
  t2 ← makeWgtTable t2 phisH1 phisH2 pbss level
  let psOfHs1 = map fst3 $ reverse phis1
  let psOfHs2 = map fst3 $ reverse phis2
  let ps = map fst ∘ concat $ reverse (map reverse pbss)
  let psOfInfs = map fst ∘ concat $ reverse (map reverse piss)
    -- for sepparation of the grid from header cells
  when (phisH1 > 0) $ tableSetRowSpacing t2 (pred phisH1) 10
  when (phisH2 > 0) $ tableSetColSpacing t2 (pred phisH2) 10
  makeButtonPressed r t1 psOfInfs
  makeButtonPressed r t2 $ ps ++ psOfHs2 ++ psOfHs1
  t3 ← tableNew 1 1 False
  label ← labelNew Nothing
  labelSetMarkup label $
    "<span foreground=\"green\" size=\"x-large\">"
     ++ "Consistent Mode </span>"
  if (isSafeMode $ ecContents ec)
    then $ do
      tableAttachDefaults t3 label 0 1 0 1
      guiSetSafeMode r True
    else
      guiSetSafeMode r False
  renderDirect (cntnr gr) [t1, t2, t3]
```

A.2 RegularTable.dtd

DTD file as explained in 5.2 for read tables from xml files and write to them is as follows:

```
<!DOCTYPE RegularTable [
```

```
<!-- The following entities fix the content types used for
     representing expressions, unary and binary functions, and types.
-->

<!ENTITY % expr "#PCDATA">
<!ENTITY % function1 "#PCDATA">
<!ENTITY % function2 "#PCDATA">
<!ENTITY % type "#PCDATA">

<!-- The top-level element for regular tables.
     Currently we only provide for one-, two-,
     and three-dimensional tables.
     The attributes specify the interpretation of the content types
     for expressions, functions, and types.
-->

<!ELEMENT RegularTable  (RegTable0 | RegTable1 | RegTable2 | RegTable3)>
<!ATTLIST RegularTable
        expr CDATA #IMPLIED
        function1 CDATA #IMPLIED
        function2 CDATA #IMPLIED
        type CDATA #IMPLIED
        >

<!-- We assume that grid cells and header cells
     all contain elements of the same syntactic catregory:
-->

<!ELEMENT cell (%expr;)>

<!-- All grids must be regular, i.e.,
     all immediate sub-grids of any one multi-dimensional grid
     must have the same dimensions and be regular themselves.
-->

<!ELEMENT grid1 (cell+)>
<!ELEMENT grid2 (grid1+)>
<!ELEMENT grid3 (grid2+)>

<!-- At each dimension, there is one type for the cells of that dimension,
     and another for the semantics of subtables of that dimension.
```

```
-->

<!ELEMENT elemType (%type;)>
<!ELEMENT semType (%type;)>

<!ENTITY % types "elemType, semType">

<!-- The following three elements
     are for the three kinds of semantics functions:
-->
<!ELEMENT gridSem (%function1;)>
<!ELEMENT addH (%function2;)>
<!ELEMENT combine (%function2;)>

<!-- A header should come equipped with type information and
     semantics information, and consists of a list of cells.
-->

<!ELEMENT headerChunk ((cell, headerChunk+) | cell)>
<!ELEMENT header (%types;, addH, combine,  headerChunk+ )>

<!ENTITY % gridstart "%types;, gridSem">

<!-- An n-dimensional regular table consists of n headers
     and a typed regular n-dimensional grid,
     where the grid size in dimension i
     has to coincide with the length of the i-th header.
-->

<!ELEMENT RegTable0 (                          %gridstart;, cell )>
<!ELEMENT RegTable1 (header,                   %gridstart;, grid1)>
<!ELEMENT RegTable2 (header, header,           %gridstart;, grid2)>
<!ELEMENT RegTable3 (header, header, header, %gridstart;, grid3)>

<!-- Standard entities: -->

<!ENTITY lt '&#60;'>
<!ENTITY gt '&#62;'>
<!ENTITY amp '&#38;'>
<!ENTITY quot "&#34;">

]>
```

Bibliography

[BKS97] Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors. *Relational Methods in Computer Science.* Advances in Computing. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.

[DKM01] Jules Desharnais, Ridha Khédri, and Ali Mili. Interpretation of tabular expressions using arrays of relations. In *Relational methods for computer science applications*, pages 3–13, Vienna, Austria, 2001. Physica Verlag Rudolf Liebing KG.

[FK04] Hitoshi Furusawa and Wolfram Kahl. Table algebras: Algebraic structures for tabular notation, including nested headers. *Programming Science Technical Report, Research Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology (AIST)*, 2004.

[FPT07] Xin Feng, David Lorge Parnas, and T. H. Tse. Tabular expression-based testing strategies: A comparison. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques — MUTATION*, page 134, Washington, DC, USA, 2007. IEEE Computer Society.

[HABJ05] Constance L. Heitmeyer, Myla Archer, Ramesh Bharadwaj, and Ralph D. Jeffords. Tools for constructing requirements specifications: the SCR toolset at the age of nine. *Comput. Syst. Sci. Eng.*, 20(1), 2005.

[Hen80] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEETSE*, 6(1):2–13, January 1980.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2007.

[Jan95] Ryszard Janicki. Towards a formal semantics of Parnas tables. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 231–240, NY, USA, 1995. ACM.

[JK01] Ryszard Janicki and Ridha Khédri. On a formal semantics of tabular expressions. *Sci. Comput. Program.*, 39(2-3):189–213, 2001.

[JPZ97] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In *Relational methods in computer science*, pages 184–196, New York, NY, USA, 1997. Springer-Verlag.

[Kah03] Wolfram Kahl. Compositional syntax and semantics of tables. SQRL Report 15, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, October 2003. available from http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[Kah04] Wolfram Kahl. New horizons for tool support of tabular expressions. Unpublished Manuscript, 2004.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. Technical Report SEN-E0420, CWI, Amsterdam, August 2004.

[LFM01] M. Lawford, P. Froebel, and G. Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Accepted for publication in FMSD Oct 2004, August 2001.

[Oka99] Chris Okasaki. From fast exponentiation to square matrices: an adventure in types. *SIGPLAN Not.*, 34(9):28–35, 1999.

[Pan06] Vera Pantelic. Inspection of concurrent systems: Combining tables, theorem proving and model checking. Master's thesis, Department of Computing and Software McMaster University Department of Computing and Software, McMaster University, 2006.

[Par92] David Lorge Parnas. Tabular representation of relations. CRL Report 260, Telecommunications Research Institute of Ontario, Communications Research Laboratory, McMaster Univ., Hamilton, Ontario Canada L8S 4K1, 1992.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report.* 2003. Also on `http://haskell.org/`.

[PLW07] Dennis K. Peters, Mark Lawford, and Baltasar Trancón y Widemann. An IDE for software development using tabular expressions. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 248–251, New York, NY, USA, 2007. ACM.

[TyWP08] Baltasar Trancón y Widemann and David Lorge Parnas. Tabular expressions and total functional programming. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 219–236, Berlin, Heidelberg, 2008. Springer-Verlag.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 148–159, Sept 1999.