FunSheet: A Functional Spreadsheet

Walter de Hoon, Luc Rutten, Marko van Eekelen

Computing Science Institute, University of Nijmegen

Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

e-mail: marko@cs.kun.nl

Abstract

It has been claimed that recent developments in the research on efficiency of code generation and on graphical input/output interfacing have made it possible to use a functional language to write efficient programs that can compete with industrial applications written in a traditional language.

As one of the early steps to verify this claim, this paper describes a first go at the implementation of a spreadsheet in a pure lazy functional language. An interesting aspect of the design is that the language with which the user specifies the relations between the cells of the spreadsheet is set out to be a standard, pure lazy functional language and not a dedicated spreadsheet language. Furthermore, the design incorporates the possibility to ask the system to prove equality of symbolic cell expressions: a property which can greatly enhance the reliability of a particular user-defined spreadsheet.

The resulting application is by no means a fully mature product. It is not intended as a competitor to commercially available spreadsheets. However, with its standard functional language and its proof capabilities it may serve as an interesting candidate to fill the gap between calculators with purely functional expressions and full-featured full-blown spreadsheets with dedicated non-functional spreadsheet languages.

1 Introduction

Traditionally, the only way to create an interface between a functional language and the imperative world was to give the functional input via a single, special input parameter and to interpret the result of the program (the output) as a sequence of commands for the outside world (Turner (1990)). In principle it is possible to do window-based I/O in this way. Due to the strong separation of input and output however it becomes a very tedious task to program a realistic application. Furthermore, the required efficiency is in many cases hard to achieve.

In the lazy, functional graph rewriting language Clean (Brus *et al.* (1987)), Nöcker *et al.* (1991), Plasmeijer & van Eekelen (1994)) uniqueness typing (Barendsen & Smetsers (1993)) can be used to indicate that a function upon its evaluation will hold the only reference to a certain (sub)-argument. So, such a function can destructively use such a reference (Smetsers *et al.* (1993)). Uniqueness types can be assigned to standard programs. The use of uniqueness typing makes it possible to write efficient applications with modern graphical user interfaces.

This paper describes the experimental design and implementation of a functional spreadsheet using Clean as the implementation language. Uniqueness typing will be used in particular for the spreadsheet data structure and for the graphical I/O interface.

An interesting aspect of the resulting application (**FunSheet**) is that it has a lazy, functional language as the spreadsheet specification language with a built-in mechanism that can (symbolically) evaluate functional expressions. By applying built-in symbolic transformation rules with rewrite semantics (expressing e.g. commutativity, distributivity and associativity of standard operators) this symbolic evaluator is able to prove equality of certain expressions (or at least simplify the equations).

2 General design

A spreadsheet can be seen as a graphical representation of computations. An important property of a spreadsheet is that when some of the data changes not all of the computations have to be performed again. Furthermore, intermediate results can be shown in the sheet and used for further computations.

Lazy evaluation and the fact that implementations of functional languages are based on graph rewriting (Barendregt *et al.* (1987), Peyton Jones (1987), Plasmeijer & van Eekelen (1993)) give good opportunities to explore the advantages of the use of a functional languages for the design and the implementation of a spreadsheet program.

An important overall intention of the design was to reuse as much as possible available software written in a functional language in order to keep the scope of the design and implementation within a six-months computer science Masters thesis project (de Hoon (1993)). Candidates for reuse were a symbolic evaluator written by L. Rutten to prove the correctness of the application of transformation rules on functional programs, a high-level machine independent window based I/O library written by P. Achten to increase the level of abstractness available for functional window based software (Achten and Plasmeijer (1994)), a window-based editor written by H. Huitema as a first test of the effectivity of this I/O library and a small help tool written by H. Huitema to make it easier to add help facilities to functional software. All of these components were written in Clean.

2.1 The sheet functionality

A spreadsheet is menu driven which means that various actions from the menu can be applied to the (contents of the) sheet. The design includes *sheet manipulation* actions (to save or read multiple sheets in separate windows, to change the names of the sheets or to close a sheet), *sheet editing* actions (Cut/Copy, Paste, Undo, Delete), *formatting* actions (to change the font of a sheet and to change the row height, and the column width) and an on-line *Help* facility. It also offers the possibility to *define new functions* and to *select (user-defined) functions*. Finally, functionality is available to define, delete and select *labels* as verbose synonyms for references to a cell.

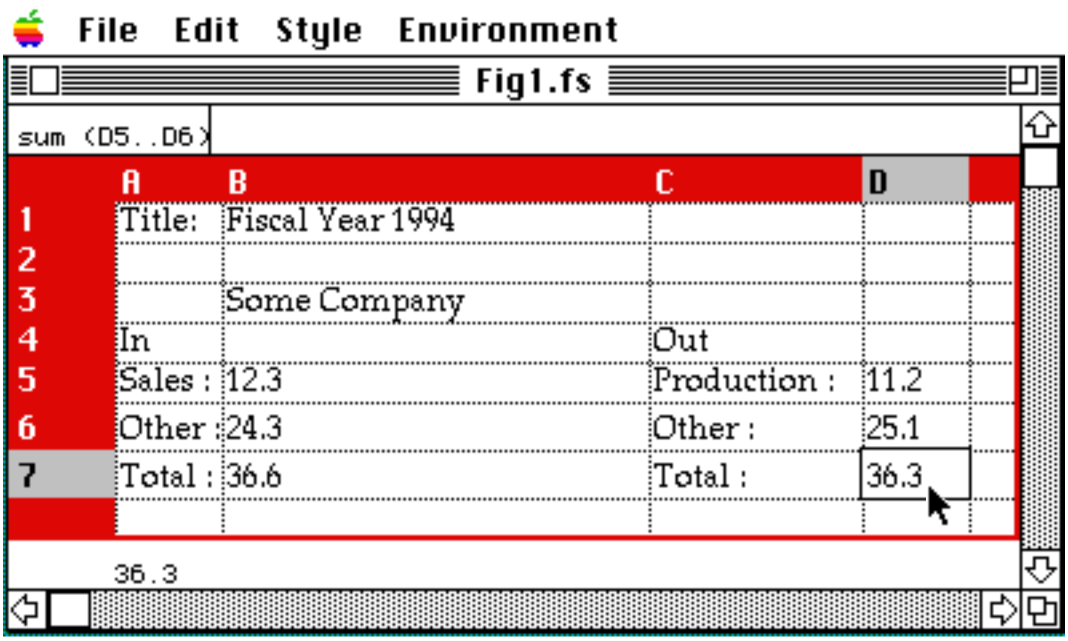

Figure 1: A user's view on FunSheet.

Most importantly, a spreadsheet has a window in which the evaluated values and the entries are displayed. The values are displayed in *cells*, squares separated by horizontal and vertical lines. Index and column information is displayed in the window constantly. The cells in the sheet can be selected by mouse clicking or by the arrows defined on the keyboard. Each sheet window contains a part displaying the cell expression to be edited by the user. After the text has been edited the computed value is shown in the selected square. The string and its evaluated value are assigned to the cell matching the selected square.

Classical spreadsheets furthermore offer lots of features among which hiding rows and columns and the possibility to make, import and export all kinds of *diagrams* in addition to the sheet. These functions are not included in the basic design. They can be added later to extend the capabilities of the application.

The most important choice of the design was to use a functional language as the spreadsheet cell expression language. An interesting aspect of the chosen functional language are its capabilities for symbolic evaluation and for applying transformation rules enabling the proof of symbolic equality for a large class of expressions.

RELATIVE ADDRESSING CONSIDERED HARMFUL

The advantages of functional languages greatly depend on the overall validness of referential transparency. With respect to cell expressions *relative addressing* referring to the neighbour cells of the current cell *would invalidate referential transparency*: the same cell expression would have a different value if it was placed on a different cell. Therefore such relative addressing is not included in the design. It is our opinion that many errors made by the user creating actual spreadsheets are caused by the complexity of the implicit context switch which is caused by copy/pasting expressions with relative addressing. It is very important to decrease the number of errors in spreadsheets and to increase the ease of reasoning about them. Many business spreadsheets are used at the core of financial administrations almost without any practical way of questioning their validity because of the complexity of the structure of the spreadsheet expressions. This complexity is greatly increased by the abundant use of relative addressing.

2.2 Having a pure functional spreadsheet language

In contrast to standard spreadsheet products such as Excel (Microsoft (1992)) we have chosen to use a pure functional language as the language with which the spreadsheet computations are described by the user. This means that all predefined functions have to be proper functions without side-effects and also each cell expression has to be *purely functional*.

The design of the spreadsheet chooses to model each column of cells as a function of indexes to values such that each cell expression in fact forms the right-hand side of one of the alternatives of this column function. These *column functions* are *first-class citizens* in the spreadsheet language. They can be used in a curried way. They can occur as arguments and as results of functions in any cell expression.

Since the type of the value of cells in the same column are not necessarily the same, it was chosen to make the spreadsheet language *not strongly typed*. In fact, only basic values carry a type on which checks can be performed dynamically.

FUNCTIONS

The functions that are predefined are not only classical spreadsheet functions like **sum** or **average** but also functions that are used often in the functional programming community such as **map** and **foldr**. Furthermore, the user is offered the possibility to develop his own functions creating a dedicated function environment per sheet. These user-defined functions can be used in the same way as the predefined functions.

CELL REFERENCES AND DEPENDENCIES

One of the most important features of a spreadsheet is the use of references. This design distinguishes two kinds of references, the references via *column functions* and references via *labels*. A *label* is an identifier referring to a cell. This label can be used anywhere in the cell expressions giving the user an extra means of introducing abstract names.

Cells are referred to via applications of *column functions*. As an abbreviation of applying a column function to an index which is formed by an integer denotation (e.g. A 1) the possibility is introduced to collapse such an application into a single identifier (A1) which is more in conformity with classical spreadsheet references.

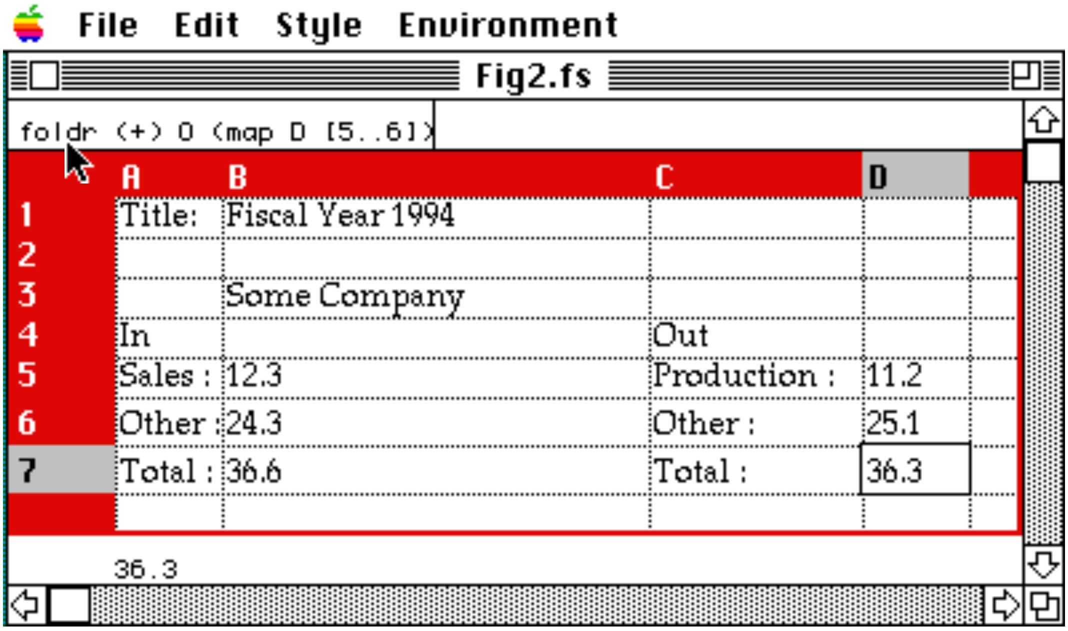

Figure 2: The use of standard and column functions.

Usually, a spreadsheet prevents having to update the whole sheet when the entry of a cell changes. This can be done by saving the addresses of the cells that depend on that particular cell (the *dependencies*). So, references create a *dependency structure* in which a cell at level 'x' depends on a cell at level 'x+1'. When a cell in layer 'x' is changed, the cells in the levels that depend on that cell are also updated. As a result of this, changing the value of one cell might have a chain reaction which recursively causes the change of a large group of cells.

For a curried application of a column function or an application of a column function with as its argument an expression which is not an integer denotation it is not possible to statically determine all dependencies. So, they have to be approximated safely. This is done by considering such expressions to be dependent on all cells in the column.

Using references to other cells creates the possibility of defining cells with a cyclic dependency structure. In many cases however such cycles correspond to erroneously non-terminating evaluation. Therefore, as in classical spreadsheets, a *cycle detector* is included which prohibits definitions that may lead to such cyclic dependencies of cells. (In fact it works with partly evaluated expressions which is why the example in figure 2 is ok, see also section 3.2).

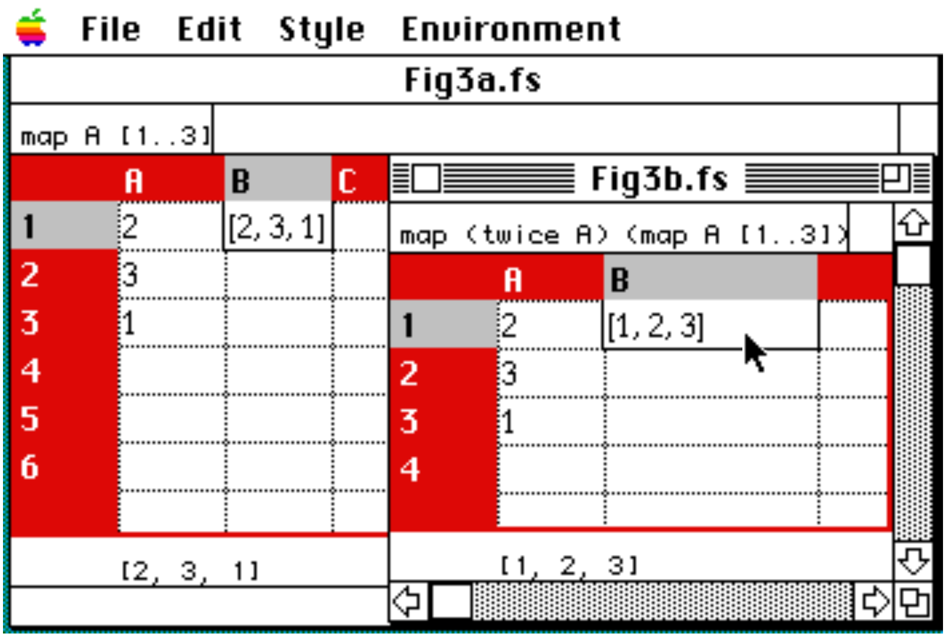

Figure 3: Abundant use of curried column functions.

The example of figure 3 shows a rather abundant use of curried column functions to illustrate the power of the generality of functional expressions.

SYMBOLIC EVALUATION

The evaluation of expressions in the language is done *symbolically* using *rewrite semantics*, i.e. there is no essential difference between functions and constructors: they can both occur at any position in a left-hand side.

Since there are some pre-defined operators which exhibit e.g. associativity and distributivity this introduces the opportunity to incorporate expressions in a spreadsheet that symbolically prove certain properties of the user's design of the spreadsheet (an example of a commutativity diagram proof is given in figure 4). This can greatly improve the reliability since testing on properties using specific values is rather error-prone (e.g. X * Y == X + Y is not true but for several specific values the equation does hold).

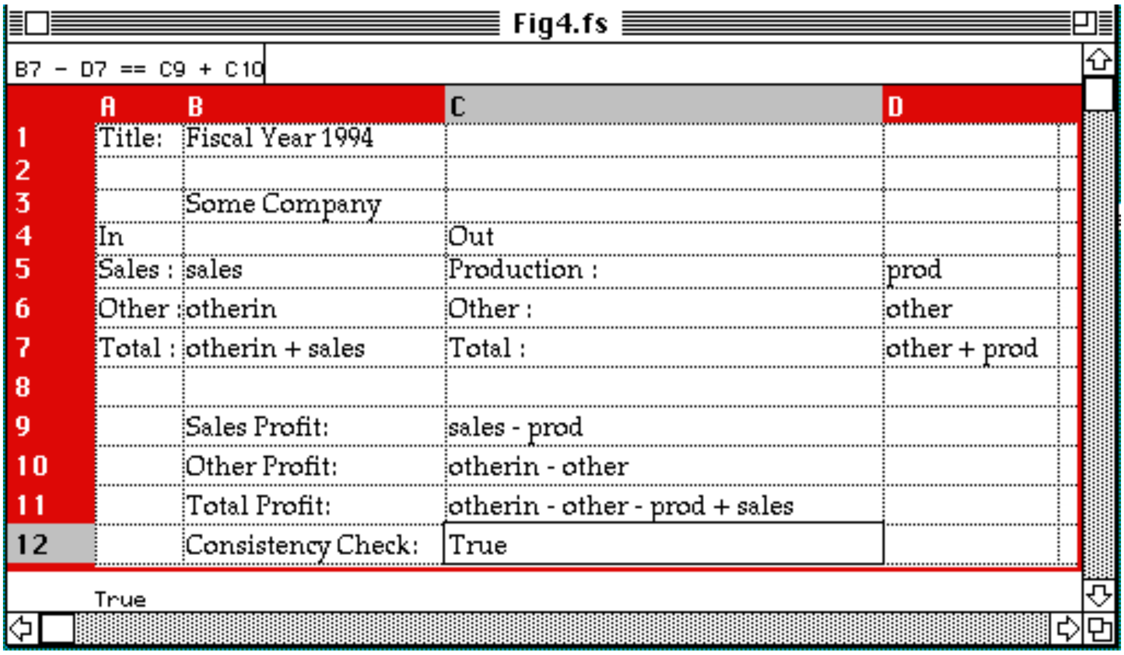

Figure 4: Proof of consistency of sum of balances

The symbolic evaluator can also be used to check properties with lists containing symbolic values (e.g. sum of one list is symbolically equal to sum of another).

Of course, the power of such a symbolic evaluator is inherently limited: the equations which it can prove are determined by the transformation rules it knows (this holds for every proof system). When the system cannot solve the equation it returns an equivalent equation which is simplified as far as possible (as is shown in figure 5).

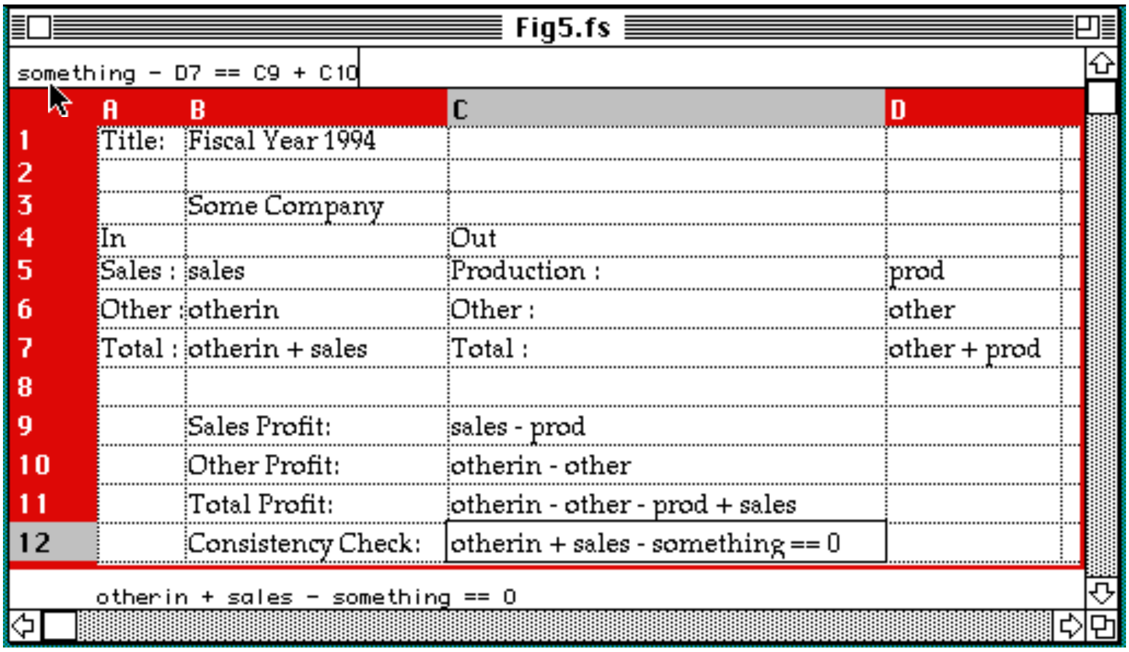

Figure 5: Equation to solve returned

SPREADSHEET LANGUAGE SYNTAX (BNF-STYLE NOTATION)

The syntax of the language describes a simple language (essentially function definitions with pattern matching and guards extended with local definitions and ZF-expressions). Most expressions would be specified similarly in commonly available functional languages. Besides using the notation hd : tl for a list the equivalent Clean-like notation [hd | tl] is also allowed.

Conventions used in the syntax specification below:

[non-term]	means that the presence of non-term is optional
{non-term}	means that non-term can occur zero or more times
terminals	are enclosed by single quotes
-	indicates a range of terminals
italics	lexical identities are italic

The syntax specification:

```
fundefs      =  [ rule {';' rule} ]
             ;
rule         =  expression '=' evaluation
             |  expression '::' expression
             ;
evaluation   =  expression ['where' localfundefs]
             ;
localfundefs =  rule
             |  '(' fundefs ')'
             ;
expression   =  basic
             |  expression operator expression
             |  identifier { expressions }
             |  '[' expression [',' expression] '..' [expression] ']'
             |  '[' expression '|'  generator { ';' qualifier} ']'
             |  '(' expression ')'
             ;
generator    =  identifier '<-' expression
qualifier    =  generator
             |  expression
             ;
basic        =  int | real | char | string | bool
             ;
```

Lexical classes used in the syntax specification above:

int	integer denotation without sign, e.g. 1, 2, 42
real	real denotation without sign, e.g. 3.3e-7
char	character denotation, e.g. 'c', '\n', '\r', '\f','\b','\t','\"','\'','\\', '\024' . Backslashes precede special character indications or three digit octal codes.
string	string denotation, e.g. "hello world\n"
bool	boolean denotation, True or False.
operator	The operators below are listed with decreasing relative priority. Operators on the same line have equal priority. A character on the beginning of each line indicates associativity: R for right-associative, L for left-associative, Pre for prefix.

```
L     ..               == within dotdot list notations [a ..], [a, b ..], [a .. c], and [a, b .. c]
R     ,                == to separate elements of lists and tuples, as in [a, b, c] and (a, b, c)
R     ;                == between qualifiers in ZF-expressions and to separate alternatives
L     ::               == between the name of an algebraic data type and its alternatives
L     |                == in ZF-expressions, to separate alternatives of an algebraic data type,
                       == and in the alternative notation [a | b] for a : b
L     <-               == employed within ZF-expressions
R     where =          == where for local definitions,
                       == = between left and right hand sides of alternatives
R     if               == within conditional alternatives
L     =>               == the boolean implication function
L     \/               == the boolean or function
L     &                == the boolean and function
Pre   ~                == the boolean not function
L     > >= == ~= <= <  == generic comparison functions
                       == for symbolic arguments an unsolved equation can be the result
R     : ++ --          == list constructor, concatenation of two lists, list difference
L     + -              == addition and subtraction of ints and reals
```

Pre	-	== negation of ints and reals
L	* / div mod	== multiplication, division, quotient, and modulus of ints and reals
R	^	== exponentiation of ints and reals
R	.	== function composition
Pre	#	== list length
L	!	== selection of elements from lists; 0 is the first index

identifier described below by a few syntactic rules:

```
identifier    =  first_idchar, {id_char};
first_idchar  =  'a' - 'z'     |   'A' - 'Z'   |   first_special;
first_special =  '$' | '_';
id_char       =  first_idchar  |   digit       |   rest_special;
digit         =  '0' - '9';
rest_special  =  ''';
```

The language is untyped and does not have an off-side rule.

2.3 The FunSheet user interface

Apart from the about dialog which includes a help facility created with the available help tool software component (for further reference the corresponding files are included in Appendix A), the FunSheet menu is composed of four separate pull-down menus. A brief description of the functionality of the menu functions listed below can be found in the help files in Appendix A.

First of all the '*file*' functions: **New**, **Read**, **Save**, **SaveAs**, **Close**, **CloseAll** and **Quit**.

Then there are the '*edit*' functions that are used to manipulate entries and labels. These are the **Cut/Copy**, **Paste** functions, the **UndoTyping** function, a function to select remote values (values defined in other sheets) and a function **Label** to manipulate labels.

Then we have the '*style*' functions. These are functions that change the format of the sheet. Functions that can be selected from this menu part are **Height**, **Width** and **Change-Font**.

And finally we have the '*environment*' functions that are concerned directly with the environment using the function editor. Here the user can perform a **Select Function** to select a (user-defined) function for use in a cell expression or the user can choose **Define/Test Function** to define new functions, test them and add them to the environment.

THE FUNCTION EDITOR USER INTERFACE

The user must be able to define functions of his own. For this purpose, a separate user interface is created that replaces the spreadsheet user interface and starts up a window based editor with some extensions in the menu to perform a **Syntax Check** of the new functions and to try an **Expression Test** to test the function by evaluating various expressions. Initially, one window is opened which shows the functions are already defined (by the user). When a new function is added to the environment, its syntax can be checked. If the function is syntactically correct the environment is updated with the new definition.

When from the editor a **Return to Spreadsheet** is performed, the adapted function environment is passed.

Besides these dedicated functions, the editor contains the standard functions a window-based editor must have such as **Undo**, **Cut/Copy/Paste**, **Tab/Font Changes**, **Find/Replace**, **Goto Line** and even **Bracket Balancing**. For further reference the reader is referred to the help file in Appendix A.

3 Implementation issues

From the design it follows that the implementation will be modular and highly structured. The different components (editor, symbolic evaluator, spreadsheet structures, user-interface) access each other only through interface modules defining abstract data structures with access functions.

3.1 Data structures

The spreadsheet data structures contain more information than we saw in the previous section: information that has to do with the efficiency of the program as well as information concerning the contents of the cells and the visual aspects of the sheet.

THE CELL

The most important information stored in the cells are the *entries*. These are the input strings given by the user. The user must be able to adjust these entries and in order to access them they have to be saved in the cells.

The *parsing information* of the entries is also stored the cells after some evaluation is performed on it in the following way. The entry is first parsed and evaluated using the standard environments of the interpreter. This results in an expression (of type Value, see section 3.2) that is evaluated as far as possible using standard functions only. Then this partly evaluated cell expression is further evaluated to its result (also of type Value), using all information available. Because it might use references it is possible (and very likely) that some of these values will change and therefore will affect the result. When one of these references changes, the entry does not have to be parsed and partly evaluated again since the *partly evaluated expression* is saved in the cell. In the environment (see section 3.2) the final result is saved in the right-hand side of the corresponding alternative of the corresponding column function.

Changing the entry of a certain cell might affect a large group of cells in the sheet. Other cells can refer to this particular cell with labels or direct references. To increase efficiency avoiding having to check the entire sheet for references to this particular cell a list of *used-by* references is obtained in the cell. For efficient checking of cyclic dependencies the references the entry of a certain cell *uses*, are also stored in the Cell.

Finally, there is the *format* information (e.g. the font which is used) which can be different for each cell. So, this information is saved in the cell with the exception of the global format information which, as described below, is put in the Sheet.

Via access functions Cell is defined as an abstract type with the following concrete representation:

```
:: Cell          = FilledCell CellContents
                 | EmptyCell;
:: CellContents  = (Entry, Format, Expression, Uses, IsUsedBy)
:: Entry         = STRING
:: Format        = (Font, Type)
:: Expression    = Value
:: Uses          = ([Address], [STRING])
:: IsUsedBy      = ([Address], [STRING])
```

NB. In the Uses and the IsUsedBy-tuples the list of addresses contains the addresses of cells and the list of strings contains the names of the labels.

THE SHEET

A Sheet is a unique abstract type in which a sheet is represented as a unique *Matrix of Cells*, where Matrix is defined as a list of lists.

Since it is possible to open more than one sheet one must be able to identify each one of those sheets. This is achieved by keeping also the *name* and the *WindowId* in the Sheet.

Each Sheet has a *local function environment*. This environment actually consists of two environments. The first one contains the *column-functions* and the second one contains the *user-defined functions*. To be able to save the latter the actual *text of the user-defined functions* is also added to the environment (the text of the column functions is saved in the cells).

Furthermore, a Sheet contains *global format information* i.e. information about the *format of groups of cells* (rows and columns). The height and width of rows and columns can be adjusted. The corresponding properties are stored in separate lists defined in the sheet.

A Sheet also has a part which contains *selection* information concerning the interactions between the user and the program. This information contains the frame that is selected by the mouse, the cell(s) being selected and if only one cell is selected, the entry of that cell.

Finally, it contains information about the *labels*. The labels are also added to the environment, but when the user needs information about (one of) the defined labels he can not get this information from the environment and therefore he has to get this information from the Sheet.

```
:: *Sheet      = (Identify,*Matrix Cell,Interface, Row,Column,[Label],ParseEnv,ContentsFont)
:: Identify    = (WindowId, Name)
:: Name        = STRING
:: Interface   = (Input, Frame)
:: Input       = (BOOL, Address, STRING, STRING)
:: Row         = [Height]
:: Column      = [Width]
:: Height      = INT
:: Width       = INT
:: Hidden      = BOOL
:: Label       = (Name, Address, [Address])
:: ParseEnv    = (STRING,(Env,Env))
```

NB. In the input-tuple the boolean indicates whether the input has changed and the two strings are the part before the cursor and the part after the cursor respectively. In the label-tuple the address list tells which cells use the label.

THE STATE

Finally there is the abstract program state State, containing all global information needed by the spreadsheet. This state is uniquely typed and it is used by all call-back functions that handle events that are generated by the user (see section 3.3). The State contains besides a list of *sheets* (as defined above), information that is sheet-independent. So, the state contains both *standard (global) environments* that are read and evaluated during the initialization of the program (see interpreter), the *files*-environment needed for file-IO (reading and saving files) and *copy-info* containing the left-top address of the copied cells and the necessary information about these cells (this is 'global' information because it is possible to copy info from one cell to an other).

```
:: *State        = (!MyFiles, StandardEnv, [Sheet], CopyInfo)
:: *MyFiles      = NoFiles
                 | Files !FILES
:: StandardEnv   = (Env,Env)
:: CopyInfo      = (Address, Matrix CopyCell)
```

USING STRICT DATA STRUCTURES

In the state definition above the tuple-element MyFiles is defined as a strict component (which itself is defined with a strict FILES part). When you write a sheet to a file (make a backup of it) you want to make sure this is done right away so that power failures will not result in losing all information. For this reason the MyFiles component is made to be evaluated each time a call-back function delivers a new state.

3.2 Values and function environments

For evaluation of function definitions and expressions several environments are important:

The following two environments are the same for all sheets. The **Basic** environment contains function definitions concerning values of basic type. These definitions include transformation rules for employing the associativity and distributivity laws of basic operators. These rules employ functions that are internal to the evaluator. Therefore it has not been made possible for the user to change or extend these definitions although they are put in a standard text file which was helpful for the ease of the development process. The **Standard** environment (see Appendix B) contains standard high-level polymorphic functions that could also have been defined by the user of the spreadsheet. These definitions are predefined for reasons of efficiency and user conveniency. For the latter environment the possibility is included for an advanced user to make extensions.

Each sheet has its own instance of the following environments. The **User-defined** function environment contains the definitions that are given by the user employing the built-in editor. The **Label** definition environment contains the definitions of labels, which are effectively just synonyms for particular cells. The **Column** function environment contains for each column function the set of rule alternatives that correspond to the cells of the column.

PARSING

Lexical analysis and parsing of expressions and definitions is relatively straightforward. It is important to note however that the corresponding parts of the symbolic evaluator could not be used without any change.

Cell references can be formulated as A 1 (an application of a column function to a row index), but also as A1. For the latter case and for recognizing and parsing remote references to values in sheets situated on disk a few adjustments had to be made to the lexical analysis of the symbolic evaluator.

LAMBDA LIFTING

Since the language allows local function definitions, a mechanism for lambda lifting must be incorporated. It was already available in the symbolic evaluator.

EAGER EVALUATION OF GENERAL ENVIRONMENTS

The general environments which are the same for all sheets (basic and standard) are eagerly evaluated by the spreadsheet application since lazy behaviour is not required for parsing and analysis of these environments. This eagerness has made it possible to greatly optimize the efficiency of these functions by generating Clean code for them and linking this code into the spreadsheet application.

EVALUATION OF EXPRESSIONS

A function environment is represented as a list of constructor and function definitions. Apart from the eagerness (and the corresponding optimized compilation process) there is no difference in the evaluation mechanism for the various environments mentioned above. All evaluation is done symbolically.

```
:: Env    =  [Rule]                  == Environment is a list of rules
:: Rule   =  Cn Id [Value]           == Constructor definition
          |  Fn Id [Alt]             == Function definition
:: Value  =  EV                      == Empty value
          |  V Id                    == Symbolic value, i.e. a free variable
          |  F Id [Value]            == Function identifier applied to a list of arguments
          |  C Id [Value]            == Constructor definition
          |  Ap Value Value          == Symbolic application
          |  Int INT                 == Basic values
          |  Real REAL
          |  Char CHAR
          |  Bool BOOL
          |  Msg STRING              == Error message
          |  A Id  [Alt] [Value]     == Function with definition, applied to arglist
          |  B Id Fns [Value]        == Link of id with compiled function, applied to arglist
:: Alt    =   ([Value], Value)       == Tuple with list of patterns and right-hand side
:: Id     =  STRING
```

Example: 1+1 is represented as F "+" [Int 1,Int 1]

Values are interpreted and evaluated using the definitions from the environments rewriting their subgraphs in the same way as standard combinator graph rewriting is performed.

RECURSION

For recursion in combinator rewriting usually a Y-combinator is used which in an implementation is optimized by creating a cyclic graph for it (so-called knot-tying). Since Clean is a graph rewriting language cyclic graph expressions can be expressed directly (see the definition of MakeRecursive). So, recursive applications in an environment are made effective by explicitly replacing them (this is done by the function DistRule) by references to the root of the environment (hence creating a cyclic reference).

```
:: MakeRecursive Env = Env
   MakeRecursive env = e      where      e = Map (DistRule e) env
```

Lazy evaluation ensures that this process is applied only when necessary. At this point graph rewriting and lazy evaluation turned out to be most useful in the implementation.

3.3 Using Clean's I/O library

The Clean I/O library makes it possible to write efficient event-based interactive programs in a pure functional language. Essentially, an interactive Clean program gets as an extra parameter a representation of the world. This world is given as an argument to a driver (the library function StartIO which repeatedly takes an event from the event queue and calls the corresponding call-back function) together with a specification of the required I/O devices which specifies what kind of device it is, what the call-back function are for each possible event. This specification is an algebraic data structure which must be an instance of the algebraic data type defined in the library. Uniqueness types (indicated by *) guarantee that an object will be privately accessed. This enables an efficient and realistic implementation of the I/O functions using destructive screen and file updates. For more information on the Clean I/O System the reader is referred to Achten and Plasmeijer (1994).

To show how such an abstract device definition is used in the spreadsheet program, figure 6 gives an example of the **File**-menu definition as it occurs in the code for the spreadsheet user interface. This definition of an algebraic data structure is an instance of the general algebraic data type which can be used in Clean to specify Menu-devices. The picture next to the definition shows the concrete device in the case of the menu definition being mapped to a Macintosh system.

```
PullDownMenu FileId "File" Able [
 MenuItem NewId      "New"        (Key 'n') Able    New,
 MenuItem OpenId     "Open…"      (Key 'o') Able    Open,
 MenuSeparator,
 MenuItem SaveId     "Save"       (Key 's') Unable Save,
 MenuItem SaveAsId   "Save As…"   NoKey  Unable SaveAs,
 MenuItem RenId      "Rename…"    NoKey  Unable Rename,
 MenuSeparator,
 MenuItem CloseId    "Close"      (Key 'w') Unable Close,
 MenuItem CloseAllId "CloseAll"   NoKey  Unable CloseAll,
 MenuSeparator,
 MenuItem HelpId     "Help"       (Key '/') Able    Help,
 MenuSeparator,
 MenuItem QuitId     "Quit"       (Key 'q') Able    Quit ]
```

Figure 6 FunSheet's initial File-menu definition

The type of each call-back function must be an instance of :: *s *(IOState *s) → (*s, *IOState *s) in which IOState is polymorphic I/O library type representing the external I/O status of the program and its event-queue. Each call-back function is a state transition function with two arguments. The first argument is the specific state of the program (for the spreadsheet program this is the type State, see section 3.1), the second argument (of type IOState State) represents the world with which input and output is performed. Each call-back function delivers a tuple with a new program state and a new IOState.

```
:: *IO    =  IOState State

New     ::    State   IO   ->  (State,IO)
Open    ::    State   IO   ->  (State,IO)
Save    ::    State   IO   ->  (State,IO)
SaveAs  ::    State   IO   ->  (State,IO)
Rename  ::    State   IO   ->  (State,IO)
Close   ::    State   IO   ->  (State,IO)
```

```
CloseAll  ::   State   IO   ->  (State,IO)
Help      ::   State   IO   ->  (State,IO)
Quit      ::   State   IO   ->  (State,IO)
```

An event-handling driver is started (usually as the main function executed by the program) with the function StartIO. As the type of StartIO shows, it takes an I/O specification, an initial program state, an initial I/O action to start with and the event queue. It delivers when it is finished the final program state and event queue.

```
StartIO   ::  (IOSystem *s) *s (InitIO *s) *Events -> (*s, *Events)
```

Event-handling drivers can be nested with the library function *NestIO* which is similar to StartIO. It takes an I/O specification, an initial program state, an intial I/O action to start with and it takes the parents IOState (which represents the world including the event-queue). NestIO delivers its own final program state and the original parents IOState to continu again. Effectively, this means that at any point in a program a sub-program can be called with its own user-interface.

```
NestIO   ::  (IOSystem *t) *t (InitIO *t) *(IOState *s) -> (*t, *IOState *s)
```

The spreadsheet program uses this nesting when calling the window-based editor of new functions with its own user interface. Since a nested I/O system returns its own program state, the IOState of the editor had to be slightly extended in order to return the new function environment. Of course, the editors user interface was also extended with a facility to check and test functions. Due to the use of NestIO all other function definitions of the editor program could remain unchanged.

Below, the definition of the call-back function SwitchToEditor is given. This call-back function is called when the user of the spreadsheet performs the command **Define/Test Function** from the **Environment** menu (see section 2.3). It employs NestIO and some access functions to transfer the definitions of the user-defined functions from the editor to the spreadsheet and vice-versa.

```
SwitchToEditor :: State IO -> (State, IO)
SwitchToEditor spreadsheetstate io
=  (newspreadsheetstate, newio)
   where
   newspreadsheetstate   =  AdaptSpreadsheetFunctionEnv  newfunctionenv spreadsheetstate
   newfunctionenv        =  GetEditorFunctionEnv editorstate
   (editorstate, newio)  =  NestIO IOSystemEditor (InitialEditorState funenv) InitIOEditor io
   funenv                =  GetSpreadsheetFunctionEnv spreadsheetstate
```

INTRODUCING HIGHER LEVELS OF ABSTRACTION FOR THE SPREADSHEET

The polymorphism employed in the Clean I/O library allows the programmer to define his own abstractions and use them without loss of generality. As an example we show the function defined below which is a user-defined generalization that captures the often occurring case that a call-back function must be applied repeatedly (e.g. each time allowing the user to type in a filename or to select a file) until he succeeds (e.g. when the file has the right characteristics).

In the spreadsheet program this abstraction was defined and used internally for various means: for opening a spreadsheet, for saving a file and to get a remote value from another spreadsheet. Its definition and an example of its use is given below.

```
tryIO ::  (*s   *(IOState *s)  -> (*s ,*IOState *s )) STRING *s  *(IOState *s)  -> (*s ,*IOState *s )
tryIO function errormessage s io
| = choice noid     =   (s,newio)
                    =   function  newio
                        where
                        (choice,newio) = OpenNotice (Notice   [errormessage,"Try again?"]
                                                              (NoticeButton yesid "Yes")
                                                              [NoticeButton noid "No"]
                                                  ) io
                        yesid  =   1
                        noid   =   2

RemoteValue ::  State IO -> (State,IO)
RemoteValue s io
|  NOT open                    =    (s1,io1)
```

```
  | NOT (RightExtension ".fs" path) =   tryIO RemoteValue "Select file with .fs extension" s1 io1
                                    =   GetField path winid s1 io1
                                        where
                                        (bl,winid, io')          =  GetActiveWindow io
                                        (files,s')               =  GetFiles s
                                        (open,path,files',io1)   =  SelectInputFile files io'
                                        s1                       =  InsertFiles files' s'
```

4 Evaluation and comparison to other work

As an application FunSheet is positioned in a category somewhere in between calculators and commercially available conventional spreadsheets. This makes comparison somehow inappropriate. Nevertheless, below an attempt is made to determine the value of its most important properties.

++ The fact that its expressions are fully functional makes it much easier to reason about than conventional spreadsheets.

++ The possibility of proving symbolic equalities can greatly improve the reliability of a user's actual spreadsheet designs.

\+ Its I/O efficiency is very good. There are no delays in editing cell or function definitions nor in 'walking' across the spreadsheets using arrow keys and scrolling the spreadsheet when necessary.

≈ Its evaluation efficiency is about the same as Miranda™[1] (varying from approximately twice as fast for standard function applications to five times as slow for user-defined function applications). The efficiency is not too bad if one considers that symbolic evaluation is employed on untyped expressions.

\- The current application is a first version which has not gone yet through the process of use and improvement which is necessary to make a proper product out of it. There is a small list of known bugs which still have to be removed.

\-- It does not (yet) incorporate diagram/report facilities nor imports from other spreadsheet applications. Rows and columns cannot be hidden. Apart from copying there are no facilities for filling a number of cells at a time.

The first two properties of FunSheet do not hold for any of the existing commercially available spreadsheets.

LAZY EVALUATION AND GRAPH REWRITING

At many points in the implementation lazy evaluation and explicit sharing were used. However, in dealing with recursion in the symbolic evaluator the combination of cyclic definitions and lazy evaluation was most important.

CODE SIZES

FunSheet requires 4 MegaByte of free memory. It will be possible to decrease the amount of necessary memory greatly when efficient code generation for general uniqueness types becomes available in Clean 1.0.

The number of lines of source code of all software components take together is approximately 22000 lines of Clean code (including the definition modules even 28000 lines). This makes the spreadsheet application one of the very few efficiently running large applications that were made by combining and adapting existing software components written in a lazy functional programming language.

The application size itself is approximately 1 MegaByte. It runs under System 7 on a Macintosh.

[1]Miranda™ is a trademark of Research Software Limited.

5 Conclusions

- The functional spreadsheet design is an interesting kind of spreadsheet with *novel properties* that can have great value for spreadsheet practice.
- The FunSheet application can be used in *everyday spreadsheet practice*. It is not a competitor for standard conventional full-blown spreadsheets but after extensions and improvement through more extensive user experience it may find its own place in between calculators and the existing commercial spreadsheets.
- Reuse and adaptation of *existing software components* (I/O library, help tool, editor and symbolic evaluator) turned out to be relatively easy with a functional language since the lack of side-effects and the high-level of abstraction provided by polymorphism and higher order functions make interfacing rather straightforward.
- Considering the relatively small scale of this project the resulting application serves as a piece of evidence that *software productivity* is greatly improved by implementing in a functional language.
- Considering the functionality with respect to user interaction and symbolic evaluation and the fact that huge parts of the software were reused the *efficiency* of FunSheet is not too bad at all.

References

- Achten, P.M. and Plasmeijer, M.J. 1994. The Ins and Outs of Clean I/O, to appear in the Journal of Functional Programming.
- Barendregt, H.P., Eekelen van, M.C.J.D., Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., and Sleep, M.R. 1987. Term Graph Rewriting. In Bakker, J.W. de, Nijman, A.J., and Treleaven, P.C. eds. *Proceedings of Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, LNCS **259**, Vol.II. Springer-Verlag, Berlin, pp. 141-158.
- Barendsen, E. and Smetsers, J.E.W. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract). In Shyamasundar, R.K. ed. *Proceedings of the Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, 15–17 December 1993, Bombay, India. LNCS **761**. Springer-Verlag, Berlin, pp. 41-51.
- Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van and Plasmeijer, M.J. 1987. Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.
- Hoon, W. de 1993. Designing a spreadsheet in a pure functional graph rewriting language, Computer Science Master Thesis nr. 300, Nijmegen University, The Netherlands.
- Microsoft Excel, Manuals, Version 4.0, 1992, Microsoft Corporation.
- Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1991. Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds. *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.
- Peyton Jones, S.L. 1987. The Implementation of Functional Programming Languages. Prentice Hall Publishing Company.
- Plasmeijer, van Eekelen 1993. Functional Programming and Parallel Graph Rewriting. Addison Wesley Publishing Company.
- Plasmeijer, M.J. and van Eekelen, M.C.J.D. 1994. Clean 1.0 Reference Manual. *Technical Report*, in preparation. University of Nijmegen, The Netherlands.
- Smetsers, J.E.W., Barendsen, E., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1993. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In Schneider H.J. and Ehrig H. eds. *Proceedings Workshop Graph Transformations in Computer Science*, Schloss Dagstuhl, January 4-8, 1993. Lecture Notes in Computer Science **776**, Springer-Verlag, Berlin.
- Turner, D.A. 1990. An Approach to Functional Operating Systems. In Turner, D.A. ed., *Research topics in Functional Programming*, Addison-Wesley Publishing Company, pp. 199-217.

Appendix A

About..

FUNSHEET: A FUNCTIONAL SPREADSHEET
Version 1.0

This application is developed using the Concurrent Clean System, a programming environment for the functional language Concurrent Clean. This system is developed by the research group on Functional Programming of the Computing Science Institute at the University of Nijmegen. To define new functions it has been integrated with a menu-driven multi-window editor which was also written in Concurrent Clean.

This application, the editor and the Concurrent Clean System are freely available for non-commercial use via FTP (pub/Clean at ftp.cs.kun.nl). The Concurrent Clean System is available for several platforms (Macintosh, PC, Sun3 and Sun4).

FunSheet Help

MENUS & COMMANDS:

FILE

- **New (Command-N)** opens a new sheet.
- **Open (Command-O)** opens a sheet on disk, filename must have extension "fs".
- **Save (Command-S)** writes the active sheet to disk.
- **SaveAs** writes the active sheet to disk using a new name.
- **Rename (Command-R)** changes the name of the sheet.
- **Close (Command-W)** closes the active window.
- **CloseAll** closes all windows.
- **Help (Command-/)** opens the window you are looking at now.
- **Quit (Command-Q)** terminates the application.

EDIT

- **UndoTyping (Command-Z)** replaces the active entry by the last one saved.
- **Cut (Command-X)** empties the selected cells after copying (see Copy) them.
- **Copy (Command-C)** gets the information from the selected cells to be pasted.
- **Paste (Command-V)** fills the selected cells using the Cut/Copy information.
- **RemoteValue** reads a value from a sheet situated on disk.
- **Label** opens a dialog from which labels can be selected, defined of deleted.

STYLE

- **Height** opens a dialog used to change the height of the rows of the selected cells.
- **Width** opens a dialog used to change the width of the columns of the selected cells.
- **ChangeFont** opens a dialog to change the font used in the sheet.

ENVIRONMENT

- **Select Function (Command-F)** shows the defined functions that can be selected for use in the entries.

- **Define/Test Function (Command-E)** opens an editor to define functions. These functions can be checked on syntax and tested whether they work correctly. This editor has his own Help File.

ACCESS TO THE SHEETS:

KEYBOARD
The keyboard is used to enter entries in the sheet. The keys that are entered are added to the entry (shown in the window). There are however some special keys.

- Using the **arrow left** of **arrow right** together with the **command** key will replace the cursor in the entry.
- Using only an **arrow key** will result in the evaluation of the entry of the current cell and selecting the cell next to it (depending on the kind of arrow).
- A **return** will result in the evaluation of the entry.

When more than one cell is selected the keyboard is disabled (you can not add an entry to more than one cell).

MOUSE
The mouse is used to select cells. This can be one single cell or a group of cells. Using the mouse together with the command key will add the reference of the selected cells of the current cell.

Editor Help

CLEAN FUNCTION ENVIRONMENT EDITOR HELP
This help file will explain all available commands and the available key- and mouse-combinations.

THE MENUS
Below follows a list of the menus of the Clean Editor with a short description of all commands. In the following 'active window' means the frontmost edit window. When your keyboard has no 'Command' key the Control key can be used instead.

THE FILE MENU:
Help... (Command-/) Open the Clean Editor Help window you are reading now or bring it to front when it is already open.
Syntax Check (Command-S) Checks the syntax of the text.
Expression Test (Command-E) Checks the syntax and then opens a dialog to edit expressions that will be evaluated. In this way the new functions can be tested.
Return To Spreadsheet (Command-W) Quits the editor. Returns to the spreadsheet. Unchecked changes/ additions in function definitions will be lost.

THE EDIT MENU:
Undo (Command-Z) Undoes the effect of all actions that change the text and/or the clipboard. The actions that can be undone are typing sequences, the Cut, Copy, Paste and Clear commands and of course the Undo command. Undoing of typing sequences has not been implemented yet!
Cut (Command-X) Cut the currently selected text in the active window from the text to the clipboard.
Copy (Command-C) Copy the currently selected text in the active window to the clipboard.
Paste (Command-V) Paste (insert) the contents of the clipboard at the current cursor location. When a piece of the text is selected in the active window it will be replaced by the contents of the clipboard.
Clear Remove the currently selected text in the active window. The text is not copied to the clipboard. Pressing backspace or delete has the same effect.
Warning: The clipboard of the Clean Editor is not the system clipboard. It is therefore not possible to copy pieces of text to or from other applications.

Balance (Command-B) Select the smallest piece of balanced text surrounding the cursor or currently selected text in the active window. This command can be used to find corresponding open and close symbols. The following pairs of open and close symbols are recognized: '(' ')', '[' ']' and '{' '}'.

Format... (Command-J) A dialog appears in which the font and font size, the tab width and the auto indent option can be set for the active window. The options that are set in the dialog can be used as default settings, such that all new edit windows will be opened using these settings. The default settings will be saved when the editor is quitted. The tab width can be any number of spaces between 1 and 99. When a number smaller than 1 resp. greater than 99 is chosen the tab width will become 1 resp. 99.

The auto indent facility of the editor can be switched on and off. When it is on the editor will remember the first non-tab non-space position in the current line. When return is typed the cursor will move automatically to the remembered position and the needed amount of spaces and tabs will be added. When Option- (or Alt-) return is typed the cursor will move to the beginning of the next (new) line. When the auto indent option is off it is the other way around: Option-return auto-indents the next (new) line, return does not.

THE SEARCH MENU:

Find... (Command-F) A dialog appears in which a text to search for and a text to replace the found text with can be specified. By default the find/replace starts from the current cursor location downwards to the end of the text. It is also possible to search backwards to the beginning of the text (check the Backward option). When Wrap Around is checked (which is the default case) the search will continue from the beginning or end of the text of the active window when the end or beginning of the text has been reached. When Ignore Case is chosen no distinction is made between lower and upper case characters. When Match Words is checked only whole words will be found. When the text to search for appears as part of another word it will be ignored. Whole words are delimited by outline characters (tabs, spaces, (semi)colons, quotation marks etcetera). When the Find button is pressed the text to be found will be found (if it exists). With the Replace button the found text will also be replaced by the text to replace with. When the Replace All button is pressed all occurrences of the text to be found will be replaced by the text to replace with.

Warning: The text to be found and the text to replace with may not contain non-printables such as tabs or newlines. When they do the find or replace commands will be ignored.

Find Next / Find Previous (Command-G) Depending on the Backward option in the Find dialog this command is Find Next or Find Previous. It will find the next (or previous) piece of text in the active window as defined in the Find dialog. The options set in the Find dialog will of course be taken into account (Wrap Around, Ignore Case and Match Words).

Find Selection (Command-H) The currently selected piece of the text will become the text to search for and the next (or previous) occurrence of this piece of text will be found. When the selected piece of text contains one or more non-printable characters (such as tabs or newlines) this command will be ignored.

Replace & Find (Command-T) When the currently selected piece of text corresponds with the text to be found it is replaced by the text to replace with and the next (or previous) occurrence of the text to be found will be found (if it exists).

Goto Cursor (Command-L) Scroll the active window in such a way that the cursor (insertion point) becomes visible. When the cursor was already visible nothing happens. When it was not, the text will scroll in such a way that the cursor will be as close as possible to the middle of the window. When a piece of text has been selected the beginning of the selection will be taken as the cursor location.

Goto Line... A dialog appears in which a line number can be entered. The cursor will move to the beginning of the indicated line. When the line number is smaller than one resp. greater than the number of lines in the active window the cursor will move to the beginning of the first resp. last line.

KEY COMBINATIONS

The following keys and key combinations have been given a more or less special meaning in the Clean Editor.

Deleting:

Backspace	delete previous character
Delete	delete next character
Option-Backspace	delete to begin of word
Option-Delete	delete to end of word

Cursor movement:
With Left, Right, Up and Down the arrow keys present on most of the keyboards are meant.

Left	one character left
Right	one character right
Up	one line up
Down	one line down
Option-Left	one word left
Option-Right	one word right
Command-Left	begin of line
Command-Right	end of line
Command-Up	top of window / previous page
Command-Down	bottom of window / next page

Selecting with the arrow keys
Shift + Any of the cursor movement combinations extend current selection

MOUSE COMBINATIONS

It is of course also possible to move the cursor and to select pieces of text with the mouse. Special mouse combinations are:

Shift-click	extend current selection to the character underneath the pointer.
Double-click	select the word surrounding the pointer.
Triple-click	select the line surrounding the pointer.

Appendix B

STANDARD FUNCTION ENVIRONMENT

The definitions below concern often used standard functions. Their definitions could have been given in exactly the same way by the user of the spreadsheet. In that case they would have been added to the user environment. However, their evaluation would be an order of magnitude less efficient since they would be interpreted instead of compiled.

The function definitions of the standard function environment are put in a standard text file (such that an advanced user with access to all spreadsheet object files, the parser and the Clean compiler can easily change or extend them to make them better suited for his purposes). The function definitions are given to the spreadsheet language parser that generates standard Clean object files that can be linked with the other object files from which the spreadsheet application is built.

```
list a  ::  (a : list a)  |  [];
tuple   ::  ();

(a : as)   --    [b]     =  as                     if a == b
                         =  a : as -- [b];
[]         --    bs      =  [];
as         --    b : bs  =  (as -- [b]) -- bs;
as         --    []      =  as;

x          ~=    y       =  ~ x == y;

[a, b      ..    c]      =  a : [b, b + b - a .. c]   if a <= b & a <= c \/ a > b & a >= c
                         =  [];
[a         ..    c]      =  a : [a + 1 .. c]       if a <= c
                         =  [];
[a, b      ..    ]       =  a : [b, b + b - a ..];
[a         ..    ]       =  [a, a + 1 ..];

abs x                    =  - x                    if x < 0
                         =  x;
```

```
all f l                 =  and (map f l);

ap f x                  =  f x;

digit x                 =  x >= '0' & x <= '9';

(f . g) x               =  f (g x);

(a : as)    ++    b     =  a : as ++ b;
[]          ++    b     =  b;

# (a : as)              =  1 + # as;
# []                    =  0;

(a : as)    !     0     =  a;
(a : as)    !     i     =  as ! (i - 1);

and                     =  foldr (&) True;

concat                  =  foldr (++) [];

const x y               =  x;

converse f a b          =  f b a;

curry f x y             =  f (x, y);

drop 0 l                =  l;
drop n (a : as)         =  drop (n - 1) as;

dropwhile f (a : as)    =  dropwhile f as          if f a
                        =  a : as;
dropwhile f []          =  [];

dup f x                 =  f x x;

filter f (a : as)       =  a : filter f as         if f a
                        =  filter f as;
filter f []             =  [];

foldl f c (a : b)       =  foldl f (f c a) b;
foldl f c []            =  c;

foldl1 f (a : as)       =  foldl f a as;

foldr f c (a : b)       =  f a (foldr f c b);
foldr f c []            =  c;

foldr1 f [a]            =  a;
foldr1 f (a : as)       =  f a (foldr1 f as);

fst (a, b)              =  a;

hd (a : as)             =  a;

id x                    =  x;

index  x                =  f 0 x
                           where (f n [] = []
                                  f n (a : as) = n : f (n + 1) as);

init [a]                =  [];
init (a : as)           =  a : init as;

integer x               =  x == entier x;

iterate f x             =  x : iterate f (f x);

last x                  =  x ! (# x - 1);

letter c                =  'a' <= c & c <= 'z' \/ 'A' <= c & c <= 'Z';

limit (a : b : bs)      =  a                       if a == b = limit (b : bs);

map f (a : as)          =  f a : map f as;
map f []                =  [];

map2 f (a : as) (b : bs) = f a b : map2 f as bs;
map2 f [] []            =  [];

max                     =  foldl1 max2;

max2 a b                =  a                       if a >= b = b;

member x a              =  or (map (== a) x);

merge (a : b) (c : d)   =  a : merge b (c : d)     if a < c
                        =  c : merge (a : b) d;
```

```
merge x []              =  x;
merge [] y              =  y;

min                     =  foldl1 min2;

min2 a b                =  a                    if a <= b = b;

mkset (a : as)          =  a : filter (~= a) (mkset as);
mkset []                =  [];

or                      =  foldr (\/) False;

perms []                =  [[]];
perms x                 =  [a : p | a <- x; p <- perms (x -- [a])];

pi                      =  4 * arctan 1;

postfix a x             =  x ++ [a];

product                 =  foldl (*) 1;

rep n x                 =  take n (repeat x);

repeat x                =  xs
                           where xs = x : xs;

reverse                 =  foldl (converse (:)) [];

scan op                 =  g
                           where g r = (r:) . rest
                                    where (rest []       = [];
                                           rest (a : x)  = g (op r a) x);

snd (a, b)              =  b;

sort x                  =  (x                   if n <= 1
                        =  merge (sort (take n2 x)) (sort (drop n2 x)))
                            where (n = # x
                                   n2 = n div 2);

spaces n                =  rep n ' ';

subtract x y            =  y - x;

sum                     =  foldl (+) 0;

take 0 l                =  [];
take n (a : as)         =  a : take (n - 1) as;

takewhile f (a : as)    =  a : takewhile f as   if f a;
takewhile f l           =   [];

tinynum                 =  2 ^ -1074;

tl (a : as)             =  as;

transpose x             =  ([]                  if x' == []
                        =  map hd x' : transpose (map tl x'))
                           where x' = takewhile (~= []) x;

uncurry f (a, b)        =  f a b;

undef                   =  error "undefined";

until f g x             =  x                    if f x
                        =  until f g (g x);

zip (a : as) (b : bs)   =  (a, b) : zip as bs;
zip [] []               =  [];
```