Implementing a Functional Spreadsheet in Clean

Walter A.C.A.J. de Hoon, Luc M.W.J. Rutten, Marko C.J.D 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 the 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 imperative language. As one of the early steps in verifying this claim, this paper describes a first attempt to implement a spreadsheet in a lazy, purely 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 itself a lazy, purely functional and higher order language as well, and not some special dedicated spreadsheet language. Another interesting aspect of the design is that the spreadsheet incorporates symbolic reduction and normalisation of symbolic expressions (including equations). This introduces the possibility of asking 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 higher order lazy functional language and its symbolic capabilities it may serve as an interesting candidate to fill the gap between calculators with purely functional expressions and full-featured spreadsheets with dedicated non-functional spreadsheet languages.

This paper describes the global design and important implementation issues in the development of the application. The experience gained and lessons learnt during this project are treated. Performance and use of the resulting application are compared with related work.

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. Several proposals have addressed these issues (Monads (Peyton Jones & Wadler (1993)), Fudgets (Carlsson & Hallgren (1993)), Clean I/O (Achten & Plasmeijer (1995))). This has given rise to the opinion that functional programming comes of age (Pountain (1994)). The spreadsheet project of which the results are described in this paper was set out to gather evidence to support this opinion.

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)) which is based on the underlying graph rewriting model (Barendregt *et al.* (1987), Plasmeijer & van Eekelen (1993)) can be used to indicate that upon its evaluation a function will hold the only reference to a certain (sub)argument. So, such a function can destructively use this unique argument (Smetsers *et al.* (1993)). Uniqueness also makes it possible to address system functions directly from within a purely functional program without loss of efficiency. The only required addition is that within the functional program uniqueness is maintained (this can be done e.g. by adding an extra unique dummy parameter to the Clean equivalent of the system functions that read/write the same globals; in this way the order of the calls of the system functions is determined by the standard function application mechanism).

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 (called **FunSheet**) is that as the spreadsheet specification language it has a lazy functional 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 as much as possible).

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 can be used for further computations.

An important overall intention of the design was to reuse as much available software as possible in order to keep the scope of the design and implementation within a six-month 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 abstraction available for functional window-based software (Achten & Plasmeijer (1995)), a window-based editor written by H. Huitema as a first test of the effectiveness 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 (version 0.8).

The most important choice of the design was to use a functional language as the spreadsheet cell expression language (this is treated in section 3). An interesting aspect of the chosen functional language is its capability for symbolic evaluation and for applying normalisation rules on symbolic expressions including equations. This enables the proof of symbolic equality for a large class of expressions.

2.1 Basic idea of the FunSheet application

A spreadsheet has a window in which the evaluated values and the entries are displayed. The values are contained in *cells*, indicated by squares separated by horizontal and vertical lines. Index and column information is constantly displayed in the window. A typical user's view of FunSheet is given in figure 1.

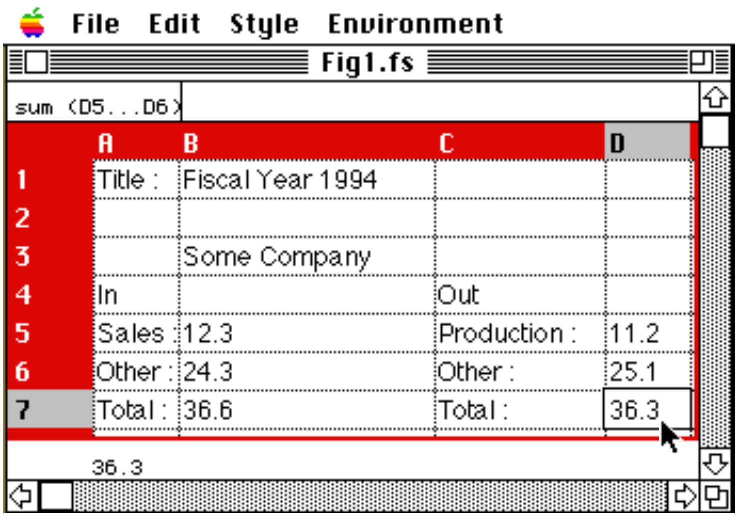


Figure 1. A user's view of FunSheet.

A spreadsheet is menu driven, which means that various actions from the menu (consisting of **File**, **Edit**, **Style** and **Environment** functions) can be applied to the (contents of the topmost) 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 of a (block of) cell(s) to a list of cells on the clipboard, Paste (when the block of cells to paste to is larger than the list of cells on the clipboard, the clipboard list is expanded with as much copies of itself as are necessary), Undo, RemoteValue (to select values defined in other sheets), *formatting* actions (to change the font of a sheet and to change row heights and column widths) and an on-line *Help* facility. The Environment menu also offers the possibility to *select user-defined or predefined functions* and to *define new functions* by switching to a built-in function editor with which for each sheet a separate set of user-defined functions can be created.

Finally, functionality is available to define, delete and select *labels* as verbose synonyms for references to a (block of) cell(s).

Using the mouse or the arrow keys the user can navigate through the cells of the sheet (evaluating cells if necessary). In the case of multiple sheets, the user can select a sheet by clicking on the sheet's window. Cells can also be selected with the mouse. When a block is selected with the mouse combined with the command key, a reference to this block is added to the previously selected cell. In the display at the top of the window, the user can edit a cell entry by moving the cursor inside it using arrow keys combined with the command key. A return will evaluate the current entry (and all those cell expressions that depend on it). The result is shown in the cell (chopped to the width of the cell) and in a special line at the bottom of the window (chopped to the width of the window).

Classical spreadsheets offer lots of additional features among which hiding, adding and deleting rows and columns, and the ability to make, import and export all kinds of *diagrams*, *print* and *report* facilities based on the information in the sheet. These functions are not included in the basic design. They are intended to be added later to extend the capabilities of the application.

2.2 The function editor

To enable the user to define functions, a function editor can be called which has a *separate user interface* that temporarily replaces the spreadsheet user interface. It 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, a window is opened which shows the functions which 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 and the user interface of the spreadsheet is re-established. Unchecked definitions will be lost. The user is asked whether re-evaluation of all cells is required.

Besides these dedicated functions, the editor contains the standard functions a window-based editor must have such as *Undo*, *Cut/Copy/Paste, Clear*, *Tab/Font Changes*, *Find/Find Next/Previous/Find Selection/Replace & Find*, *Goto Cursor/Line* and also *Bracket Balancing* and an *Auto-indent* facility.

Several *key combinations* are defined to increase the convenience of editing (character/word delete forward/backward, arrow keys to navigate across characters and lines, option-arrow keys to navigate across words, command-arrow keys to navigate to begin/end of line/page). In a similar way combinations of keys and mouse actions can be used for selecting characters, words and lines.

3 A purely functional spreadsheet language

In contrast to the macro-facilities of standard spreadsheets such as Excel or Lotus 1-2-3 which are heavily criticised because of their lack of proper abstraction mechanisms (Casimir & Rommert (1992), Litecky (1990)), FunSheet uses a *purely functional higher order* language to allow the user to describe spreadsheet computations. A function is defined (by the user) via a set of (recursive) equations with the usual *rewrite semantics*: upon evaluation of an expression, the equations are used as rewrite rules where the left-hand side of an equation serves as a pattern to determine whether the rule is applicable and the right-hand side is used to determine the result of the corresponding reduction. The order of the rules is important: they are considered as candidates for rewriting proceeding textually from top to bottom.

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. For example, an alternative of some column function `A` may be `A 1 = e`. The right-hand side `e` of this alternative defines the contents of cell `A1`, i.e. the application of column function `A` to the index `1`. These *column functions* are *first-class citizens* in the spreadsheet language. They can be used in a curried way (i.e., a column function can be used while its argument is not yet supplied). Column functions can occur as arguments and as results of functions in any cell expression.

Since symbolic evaluation will be performed and since the types of the values of cells in the same column are not necessarily the same, it was decided that the spreadsheet language should be *untyped* (no type checking at all was implemented: `'c' + 1` is not disallowed: it is just an irreducible expression).

3.1 FunSheet language syntax

The syntax of the language describes a simple language (essentially function definitions with pattern matching and guards extended with special syntax for lists, tuples, local definitions, dot-dot expressions (denoted using ..), and ZF-expressions). Most expressions would be specified similarly in commonly available functional languages. Denotations are included for integers, reals, booleans, characters and strings. Special dot-dot expressions (denoted using ...) are available to denote blocks of cells. Lists are a predefined data structure. Besides using the notation `hd : tl` for a list, the equivalent Clean-like notation `[ hd | tl ]` is also allowed. Algebraic data types can be defined. Most standard operators on these data structures have been included in the language (basic arithmetic, relational operators, function composition, list operators for construction, selection, concatenation and difference). Also, a number of standard functions is predefined (see section 3.4). The language is *untyped* and does not have an off-side rule. For more information on the language the reader is referred to (de Hoon *et al.* (1994)).

3.2 Cell references and dependencies

One of the most important features of a spreadsheet is the use of cell references. The design uses absolute references only. It distinguishes two kinds of cell references: references via *column functions* and references via *labels*. A *label* is an identifier referring to a (block of) cell(s). A label can be used anywhere in 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 the application of a column function to an index (e.g. `A 1`) the possibility is introduced to collapse such an application into a single identifier when the index is an integer literal (`A1`) which is more in conformity with classical spreadsheet references. Figure 2 shows an example of the use of standard functions in combination with curried applications of column functions.

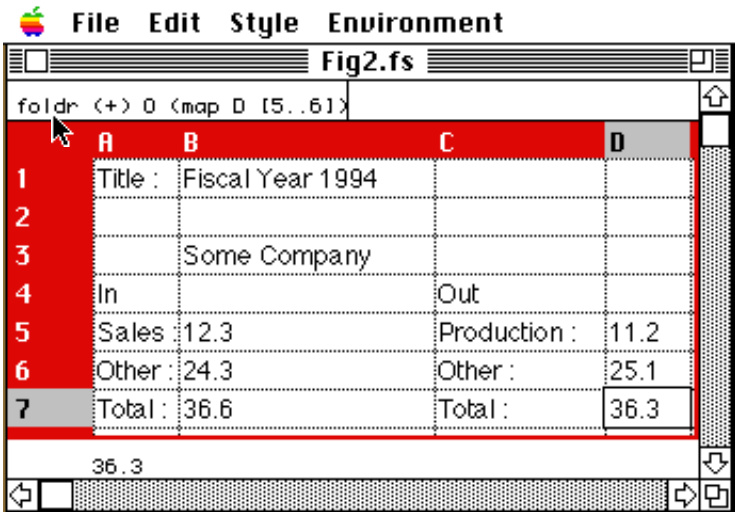


Figure 2. The use of standard and column functions.

The spreadsheet design avoids having to update the whole sheet when the entry of a cell changes. This is done by saving the addresses of the cells that depend on that particular cell (the *dependencies*). In this way, references create a *dependency structure* in which a cell depends on one or more other cells. When a cell is changed, the cells that depend on that cell are also updated, and so forth and so on. So, changing the value of one cell may produce 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 to 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 depend 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. The cycle detector guarantees that non-termination cannot be caused by cyclic dependencies of cells. It operates on partly evaluated cell expressions.

When the cell expression is parsed, standard functions and remote values are also evaluated (also see section 4.3.1). For reasons of efficiency the result of this is used as the expression to evaluate when a change occurs of other cell expressions on which this expression depends. (This is why the example in figure 2 is not prohibited: the partly evaluated cell expression of foldr (+) 0 (map D [5..6]) is D5 + D6.)

The cycle detector does allow the standard examples with e.g. sub-totals and totals in the same column. It can, however, require certain expressions that heavily use curried column functions to be put in a different column (the expression in column B of figure 3 would not be allowed in column A: its partly evaluated expression is [A(A(A(A1))), A(A(A(A2))), A(A(A(A3))), A(A(A(A4)))] which may be cyclic if put in a cell of column A).

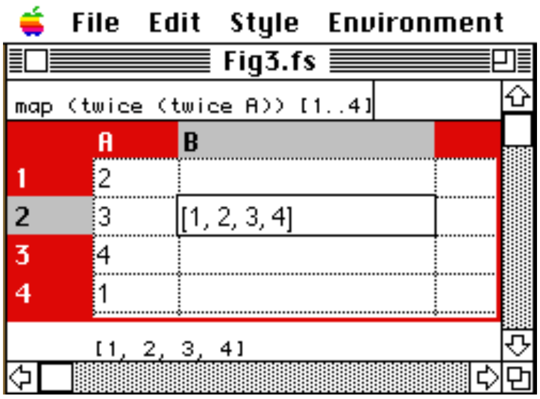


Figure 3. The use of column functions as first class citizens.

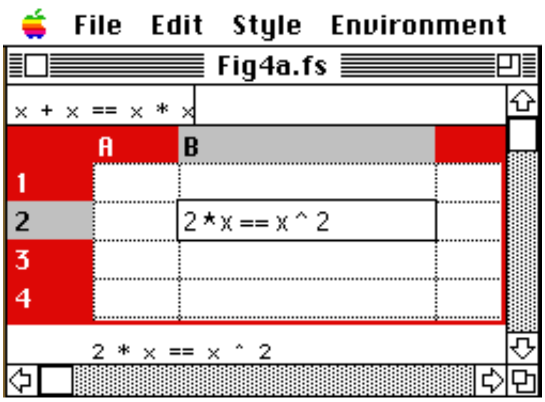
3.3 Symbolic evaluation

The evaluation of expressions in the language is done *symbolically* using rewrite semantics. Essentially there is no difference between functions and constructors. In definitions they can both occur at any position in a left-hand side of an equation (e.g. besides the usual arithmetic equations, one of the rules of the predefined basic function + is that a + (b + c) = (a + b) + c. In this rule, the function + occurs twice in the left-hand side, which is typically only allowed if rewrite semantics are used).

Evaluation of a single cell expression is chosen to correspond to evaluation of an initial term in a standard lazy functional language. So, evaluation of a single cell expression is always performed to *normal form*. Another choice could have been to make cell expressions correspond to sub-expressions in a standard lazy functional language and hence to reduce them to (weak) head normal form only. Since results can contain symbolic values, head normal forms are reached relatively quickly. So, this choice would have led to too many cases in which cell results would still contain redexes. Another option to be considered in the future would be to evaluate each cell expression only as much as is needed to print results. This is a mixture of normal form and (weak) head normal form reduction. This would allow e.g. infinite lists to occur as cell results.

Symbolic values can either be symbolic *variables* or *references to cells which are (still) empty*. The evaluation mechanism treats both cases in the same way.

When a symbolic equation cannot be solved, the equation itself, reduced as much as possible, is returned as the result (figure 4a). When instead of symbolic values, basic values are used in the same equation (this can be done by manual substitution, by adding local definitions (in the case of a symbolic variable) or by defining a cell (in the case of a reference to an undefined cell) the equation may be solved depending on the actual values (figure 4b).



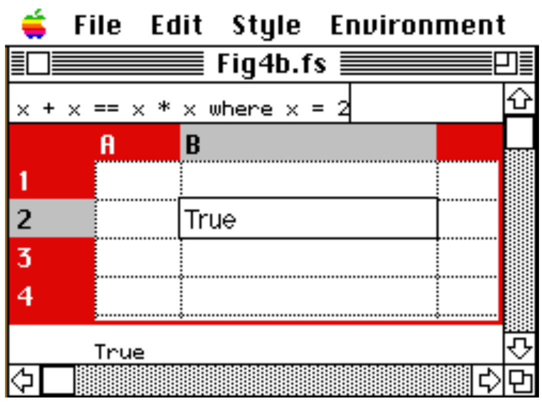


Figure 4. The use of symbolic values in equations.

For several pre-defined operators which exhibit properties like associativity, commutativity and distributivity, the symbolic evaluator includes normalisation rules. This makes it possible to symbolically solve simple algebraic equations (an example of this is given in figure 5).

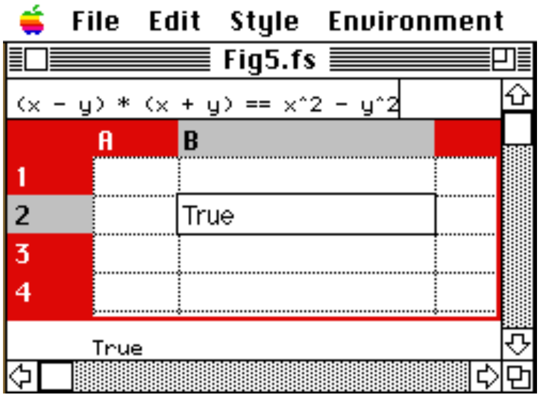


Figure 5. Solving symbolic algebraic equations.

In the symbolic evaluator it has been chosen to implement the common associativity, commutativity and distributivity rules for the arithmetic operators not excluding finite precision integers and floating point numbers. It has to be noted however that when these rules are applied on such numbers, due to (rounding) errors differences can occur between symbolically deduced results and concrete results. Arbitrary precision integer numbers may be incorporated in a future version. However, the anomaly can only be removed satisfactorily when solutions for exact real arithmetic (Cartwright & Boehm (1990), Vuillemin (1987)) become practical.

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). Figure 6 gives an example of this in which the list not only contains symbolic values but is also generated using symbolic values in a dot-dot expression.

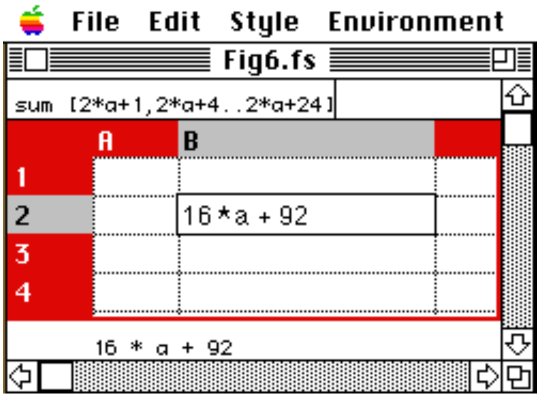


Figure 6. Symbolic evaluation of the sum of a list.

3.4 Predefined functions

Apart from the basic arithmetic functions like + and *, over 60 standard functions are predefined. These do not only include classical spreadsheet functions like sum or average but also functions that are most often used in the functional programming community, e.g. map and foldr. The definitions of the *standard* functions (the non-basic predefined functions) are contained in the *Help* files. They could have been given in exactly the same way by the user of the spreadsheet by using the ability to define a set of functions in a dedicated environment for each separate sheet.

Besides the well-known standard functions, the FunSheet application supports some special functions and constructors. There are functions to convert column indications to integers (e.g. A is converted to 1) and vice-versa. There is a function to generate cell blocks. There is a special constructor $ which acts as a prefix of a number which is maintained during arithmetic operations (useful for financial calculations).

There is a function to perform lambda-abstraction (\) of which the definition is such that $x_1\ x_2 \ldots x_n$ \ e corresponds with the lambda term $\lambda x_1.\lambda x_2 \ldots. \lambda x_n.$ e. It is also used internally to implement ZF-expressions. Furthermore, there is a function to simplify equations in which list expressions occur, by performing induction on the length of lists. This function is called ilp (an abbreviation for "induce list property"). Its definition and two small examples of its use are given below.

```
ilp p    =    p [] & (p as => p [a | as])
```

Here, a and as are symbolic values, => is logical implication (a => b is defined as b \/ ~ a), &, \/, ~ are logical and, or and not respectively. An example of the use of ilp is in the expression ilp (l \ l ++ [] == l), which reduces to True. In general if ilp is applied to a property, then a and as may appear in the result of the application. For example, ilp (l \ l ++ l == l) reduces to ~ as ++ as == as \/ as ++ [a | as] == as. This expression is False if as = [].

3.5 The use of FunSheet

Apart from being used in a way which is standard for a spreadsheet, the FunSheet application offers new opportunities to explore the use of the symbolic evaluator.

Testing properties of a particular spreadsheet set up by the user using specific values is rather error-prone (e.g. X * Y == X + Y is not true in general but for several specific values the equation does hold, see also figure 4). An important way to avoid spreadsheet errors is offered by the symbolic evaluation mechanism: the system can try to symbolically *prove* certain *properties* by simplifying equations.

An example of a commutativity diagram proof is given in figure 7. The example proves that while the cells that are referred to are still empty, the sum of the sums of rows is equal to the sum of the sums of columns. It shows how a user can *prove* that a particular set-up of a spreadsheet has a required property by adding symbolic equations.

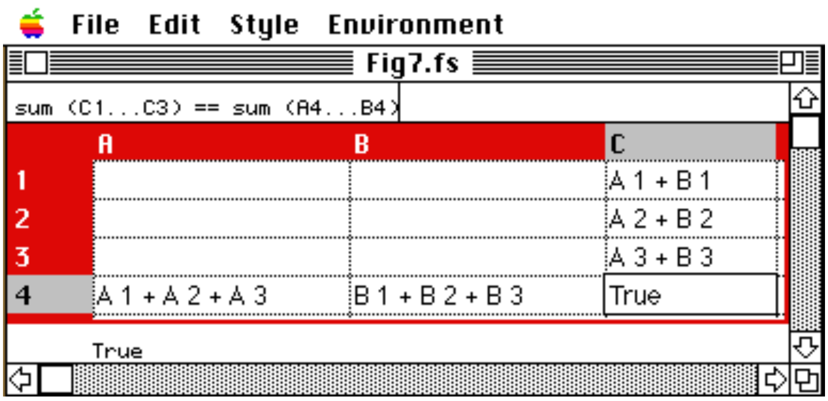


Figure 7. Proof of commutativity of summing rows and columns.

In some commercial spreadsheet applications, a similar equation as the one in figure 7 might accidentally also yield True since there the value of an empty cell is sometimes taken to be zero. This is clearly an error-prone property of such spreadsheet applications.

It is clear that such general, automatically performed proofs can greatly improve a spreadsheet's reliability. However, 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).

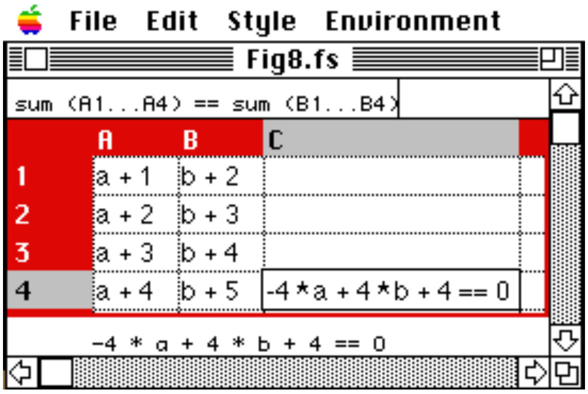


Figure 8. Returned equation can be used as a requirement to satisfy a desired property.

Another area in which FunSheet offers new opportunities is an area which is a kind of *reverse engineering*. The property that, when an equation is to be solved, the system returns an equivalent equation simplified as much as possible, can be used to inform the user what the requirements are to satisfy a certain property. In figure 8, an example is given in which an equation is returned which indicates what the relation must be between the symbolic variables a and b in order to satisfy the property that the sum of the two columns are equal.

This reverse engineering could be used in practice e.g. by filling in a tax-form spreadsheet partly symbolically. Then, e.g. near the end of a fiscal year, a user could ask the system to return the equation which states which requirements have to be met to achieve a certain threshold for getting tax returns. The only change the user would have to make to the tax-form is to type in the threshold equation in a cell. The user could then fulfil the resulting requirements (e.g. donating the right amount of money to a good cause) before the fiscal year ends.

4 Implementation issues

Since the design sets out to re-use existing software as components in the implementation, the implementation will have to be modular and highly structured. The main components (user-interface, editor, symbolic evaluator, spreadsheet structures) access each other only through interface modules defining abstract data structures with access functions.

4.1 Input/Output

The Clean I/O library makes it possible to write efficient event-based interactive programs in a purely functional language. Essentially, an interactive Clean program gets a representation of the world as an extra parameter. This world is given as an argument to a driver together with a specification of the required I/O devices which specifies what kind of device it is and what the call-back functions are for each possible event. This driver is the library function StartIO which repeatedly takes an event from the event queue and calls the corresponding call-back function. The I/O 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 (1995)).

To show how such an abstract device definition is used in the spreadsheet program, figure 9 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.

Although the spreadsheet has been written in Clean version 0.8, in the Clean program examples 1.0 syntax (which is similar to the syntax of most other functional languages) is used in order to avoid unnecessary distraction of the reader (also see sections 5.1 and 5.4).

```
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 SvAsId   "Save As…"   NoKey      Unable  SaveAs,
 MenuItem RenId    "Rename…"    NoKey      Unable  Rename,
 MenuSeparator,
 MenuItem CloseId  "Close"      (Key 'w')  Unable  Close,
 MenuItem ClsAllId "CloseAll"   NoKey      Unable  CloseAll,
 MenuSeparator,
 MenuItem HelpId   "Help"       (Key '/')  Able    Help,
 MenuSeparator,
 MenuItem QuitId   "Quit"       (Key 'q')  Able    Quit ]
```

Figure 9. FunSheet's 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 a 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 4.3.3). The second argument (of type IOState State) represents the world with which input and output is performed. Each call-back function (New, ... , Quit) delivers a tuple with a new program state and a new IOState.

```
:: *IO   =  IOState State

New     ::   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, and the event queue. When it is finished, it delivers 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 initial I/O action to start with and it takes its parent's IOState (which represents the world including the event-queue). NestIO delivers its own final program state and the original parent's IOState to continue. 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 editor's user interface (the algebraic data structure describing the main menu and its call-back functions) was also extended with a facility to check and test functions and the state of the editor had to be extended with an environment (of type Env, see the next section) to be aware of function definitions. However, due to the use of NestIO, all other function definitions of the editor program could remain unchanged. So, the function NestIO played a vital role in re-using the editor program. It dealt with switching I/O interfaces when switching from the sheet to the editor and it dealt with passing the required information about the functions between them.

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. 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
```

It is interesting to compare the definition above with the initial expression of the original stand-alone editor application which is given below (note that the definitions of the arguments of StartIO were changed as described above to be able to deal with functions).

```
StartIO IOSystemEditor InitialEditorState InitIOEditor io
```

4.2 Expressions and function definitions

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 contains the predefined standard functions. These definitions are predefined for reasons of efficiency and user convenience.

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. For each column function, the *Column* function environment contains the set of rule alternatives that correspond to the cells of the column.

Evaluation of functions from the user's environment is generally an order of magnitude less efficient than evaluation of functions from the standard environment since the user's functions are interpreted instead of compiled. So, for reasons of efficiency the predefined function definitions are given to a special Clean application which uses the spreadsheet language parser and generates a Clean definition and implementation module for each predefined function. These modules are compiled and linked in the standard way together with all other modules from which the spreadsheet application is built. An advanced user with access to all Clean sources can easily take his or her own function definitions and compile and link them to achieve a better efficiency.

Apart from the optimised compilation process (see section 4.2.4) there is no difference in the evaluation mechanism for the various environments mentioned above. Evaluation is done entirely symbolically.

4.2.1 Parsing

Lexical analysis and parsing of expressions and definitions is relatively straightforward. It was already available in the symbolic evaluator. Compound expressions adhere to an operator grammar. 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, a few adjustments had to be made to the lexical analysis present in the symbolic evaluator.

4.2.2 Representation of expressions, function definitions and environments

The symbolic evaluator implements a purely functional language which supports symbolic values. Semantically, a symbolic expression may contain one ore more *free variables*. A free variable is an identifier which is not defined as a function, constant, or constructor. To explain the meaning of functions written in the FunSheet language, we will consider their translations to Clean. The translated functions operate on arguments of type Value. Values are evaluated using the definitions from the environments rewriting their subgraphs in the same way as standard combinator graph rewriting is performed.

```
:: Value  =  EV              // Empty value
          |  F Id [Value]    // Application of a function without definition or of a free variable
                             // to a list of arguments
          |  C Id [Value]    // Application of a constructor to a list of arguments
          |  INT   Int       // Basic values
          |  REAL  Real
          |  CHAR  Char
```

```
          |     BOOL     Bool
          |     Msg   String              // Error message
          |     A Id  [Alt] [Value]       // Application of a function with definition to a list of arguments
          |     B Id Fns [Value]          // Application of a compiled function to a list of arguments
:: Fns    =     Fn0 Value                 // Nullary function; Value is the type of the result
          |     Fn1 (Value -> Value)      // Unary function
          |     FnL ([Value] -> Value)    // N-ary function with arguments in a list
:: Id     :==   String

Example: 1+1    is represented as         A "+"  {alternatives of +} [INT 1,INT 1]
```

A function environment is represented as a list of constructor and function definitions.

```
:: Env    :==   [Rule]                    // Environment is a list of rules
:: Rule   =     Cn Id [Value]             // Constructor definition
          |     Fn Id [Alt]               // Function definition
:: Alt    :==   ([Value], Value)          // Tuple with a list of patterns and a right-hand side
```

4.2.3 Interpreted symbolic evaluation of expressions

When an expression is to be interpreted, it is given as an argument to an interpreter that also takes an environment and substitutes the definitions for the function applications, reducing the expression to normal form employing symbolic evaluation lazily.

To simplify this evaluation process, all local definitions of an environment are transformed to global definitions. The local function definitions could be translated either to closures via applications of the predefined lambda abstraction function (\, see also section 3.4) or to global function definitions by adding parameters using lambda lifting. A mechanism for lambda lifting was already available in the symbolic evaluator. The use of \ is less efficient than the use of lambda lifting since \ itself is not built-in but treated like any other predefined function. So, in this case, lambda lifting was the most natural choice to deal with local function definitions.

In order to easily deal with recursion, the choice was made to let recursive applications of function definitions refer directly to their definitions. The way in which this is achieved is similar to the way recursion in combinator rewriting is usually dealt with. There, a Y-combinator is used which in an implementation is optimised 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
```

The function MakeRecursive uses this method to replace all applications of identifiers of functions (F ...) by applications of the corresponding function with its definition (A ...) or by a direct call to a standard function (B ...). 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.

4.2.4 Compiled symbolic evaluation of expressions

For reasons of efficiency, the predefined function definitions are given to a special Clean application which translates FunSheet functions to Clean code which is linked into the application so that they can be evaluated efficiently.

As free variables are not allowed in Clean, treatment of these symbolic values by compiled FunSheet functions has to be coded explicitly. A FunSheet function alternative which has a non-variable pattern in its left hand side is translated to two Clean alternatives. The first alternative is employed to catch unwanted matchings of free variables with non-variable patterns. The second alternative corresponds directly with the original alternative.

As a simple example, let us consider the following alternative.

```
f 0 = 0
```

It will be translated to the following two Clean alternatives (in which variable is a function defined below).

```
f v        |  variable v =  F "f" [v]
f (INT 0)                =  INT 0
```

Let us consider the more general case of an alternative of a function f, printed as "f", with n arguments,

```
f p1 ... pn = r
```

This alternative will be translated to the following two (schematically written) Clean alternatives

```
f v1 ... vn    |   condition =  F "f" [v1, ..., vn]
f p1 ... pn                  =  r
```

The *condition* is an expression over the free variables v_1 ... v_n. If f x_1 ... x_n is evaluated for some expressions x_1 ... x_n, *condition* is True if and only if matching some x_i with a p_i would involve matching a free variable with a non-variable pattern. The *condition* can be expressed as a function of p_1 ... p_n and v_1 ... v_n. Its implementation follows below. From the implementation it can be inferred that evaluation of a *condition* does not have an effect on the strictness (and hence termination) properties of the translated function in which the *condition* occurs, if the function is applied to arguments which do not contain free variables.

```
condition :: [Value] [Value]  ->  Value
condition [F f a | ps] [v | vs]  =  condition ps vs
condition [p | ps]     [v | vs]  =  or (pattern_condition p v) (condition ps vs)
condition []           []        =  F "False" []

pattern_condition ::  Value        Value  ->  Value
pattern_condition     p=:(C f a)   v      =   or  (F "variable" [v])
                                                  (and   (F "same_structure" [p, v])
                                                         (condition a (select_arguments a 1 v)))
pattern_condition     p            v      =   F "variable" [v];

select_arguments :: [Value] Int Value -> [Value]
select_arguments [a | as] i v    =  [F "nth_argument" [F (ToString i) [], v] | select_arguments as (i + 1) v]
select_arguments []       i v    =  []
```

The or and and functions below are used to simplify the *condition* if possible.

```
or ::   Value        Value             ->  Value
or      x            (F "False" [])    =   x
or      x            y                 =   F "||" [x, y]

and ::  Value        Value             ->  Value
and     x            (F "False" [])    =   F "False" []
and     x            y                 =   F "&&" [x, y]
```

The functions below will only be used at the run-time of a compiled FunSheet program. They are linked with the Clean code which is (partly) generated by the functions above.

```
variable :: Value -> Bool
variable (F f a)    =   True
variable x          =   False

same_structure :: Value Value -> Bool
same_structure (C f a)   (C g b)    =   f == g && # a == # b
same_structure x         y          =   False

nth_argument :: Int Value -> Value
nth_argument n (C f a)   =   select n a

select :: Int [Value] -> Value
select n [a | as]   |   n == 1        =   a
                    |   otherwise     =   select (n - 1) as
```

As a more complicated example, let us consider the following alternative.

```
f [0] = 0
```

It will be translated to the following two Clean alternatives.

```
f v       |   variable v ||
              (same_structure (C ":" [INT 0, C "[]" []]) v &&
              (variable (nth_argument 1 v) || variable (nth_argument 2 v)))      =   F "f" [v]
f (C ":" [INT 0, C "[]" []])                                                     =   INT 0
```

Here, || and && are infix operators in Clean for the "or" and "and" functions respectively.

It is possible that a non-trivial Value *value* occurs more than once in a *condition*, or that it occurs in a left hand side pattern and in the condition of the corresponding right hand side. Then in the final translated code a node identifier will be defined as *value* in a where-expression, and the original occurrences of *value* will be replaced with that node identifier. For example, if *condition* looks like ... *value* ... *value* ..., it will be translated to ... *v* ... *v* ... *where v = value*, *v* being a node identifier. This obviously saves space. It also saves time since values do not have to be rebuilt.

An example where node identifiers are generated is the following. Consider the alternative

```
f [[0]] = 0
```

It will be translated to the following two Clean alternatives:

```
f v      |   variable v ||
             (same_structure (C ":" [n1, C "[]" []]) v &&
             ((variable n2 ||
             (same_structure n1 n2 &&
                 (variable (nth_argument 1 n2) || variable (nth_argument 2 n2)))) ||
             variable (nth_argument 2 v)))                                            =   F "f" [v]
                                                                                          where
                                                                                          n1 = C ":" [INT 0, C "[]" []]
                                                                                          n2 = nth_argument 1 v
f (C ":" [C ":" [INT 0, C "[]" []], C "[]" []])                                       =   INT 0
```

Apart from generating conditions from patterns, the translation of the FunSheet language to Clean is quite straightforward. One aspect of the translation still needs to be addressed. If the set of alternatives of a FunSheet function is not exhaustive, then one extra alternative is generated at the end of its translated counterpart in Clean. If the function, say f, expects n arguments, then this extra alternative looks like

$$f\ v_1\ \dots\ v_n \quad = \quad F\ \text{"f"}\ [v_1, \dots, v_n]$$

where $v_1 \dots v_n$ are node identifiers. By adding this alternative a head normal form will be yielded when the other generated alternatives do not match.

4.3 The main data structures of the spreadsheet

The spreadsheet data structures contain 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.

4.3.1 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 in the cells after partial evaluation is performed on it as follows. The entry is first parsed and evaluated using the standard environment of the interpreter. This results in an expression (of type Value) 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. Also when cells are evaluated again after the user has changed function definitions, this partly evaluated expression is taken as the starting point of re-evaluation. In the environment (of type Env) 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 may 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 retained in the cell. This list is also used by the cycle detector. For efficient adjustment of these references, the list of cell references and label names which the entry of a certain cell *uses*, is also stored in the Cell. These lists are determined from the partly evaluated cell expression.

Via access functions, Cell is defined as an abstract type with the following concrete representation (:== indicates a type synonym definition):

```
:: Cell            =    FilledCell CellContents
                   |    EmptyCell;
:: CellContents    :==  (Entry, Expression, Uses, IsUsedBy)
:: Entry           :==  String
:: Expression      :==  Value
:: Uses            :==  ([Address], [LabelName])
::    LabelName    :==  String
:: IsUsedBy        :==  [Address]
```

4.3.2 Sheet

Sheet is an abstract type, corresponding to a concrete type which is a tuple of several components. The set of cells is represented as a *Matrix of Cells*, where Matrix is defined as a list of lists since proper arrays were not available when the program was written.

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 the Name and the WindowId in the sheet, too.

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 sheet (the text of the column functions is saved in the cells).

Furthermore, a sheet contains *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 information concerning the *interactions* between the user and the program. This information includes the *frame* (i.e. a rectangle in window co-ordinates) that is selected by the mouse, and the *input* tuple that is being edited in the cell. The input tuple contains a boolean indicating whether something has been changed, the input text, and the selected cell *block* (i.e. a rectangle in cell matrix co-ordinates).

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 or she can not get this information from the environment. Therefore this information has to be extracted from the sheet.

```
:: Sheet                  :==  (Identify, Matrix Cell, Interaction, Row, Column, [Label], ParseEnv, Font)
:: Identify               :==  (WindowId, Name)
::    Name                :==  String
:: Matrix a               :==  [[a]]
:: Interaction            :==  (Input, MouseFrame)
::    MouseFrame          :==  Rectangle
::       Rectangle        :==  (Address, Address)
::          Address       :==  (Int, Int)
:: Input                  :==  (Changed, TextBeforeCursor, TextAfterCursor, CellBlock)
::    Changed             :==  Bool
::    TextBeforeCursor    :==  String
::    TextAfterCursor     :==  String
::    CellBlock           :==  Rectangle
:: Row                    :==  [Height]
```

```
::    Height              :==   Int
::  Column                :==   [Width]
::    Width               :==   Int
::  Label                 :==   (Name, CellBlock, LabelUsers)
::    LabelUsers          :==   [Address]
::  ParseEnv              :==   (FunctionDefsText, (UserFunctionEnv, ColumnFunctionEnv))
::    FunctionDefsText    :==   String
::    UserFunctionEnv     :==   Env
::    ColumnFunctionEnv   :==   Env
```

4.3.3 State

Finally, there is the abstract program state State, containing all global information needed by the spreadsheet. This state is uniquely typed (a * is used to indicate uniqueness of the type it precedes) and it is used by all call-back functions that handle events that are generated by the user (see section 4.1). Besides a list of sheets (as defined above), the state contains information that is sheet-independent. So, the state contains the *files*-environment needed for file-IO (reading and saving files) and the *clipboard* containing a list of the entries of the copied cells.

```
::  *State        :==   (!MyFiles, [Sheet], Clipboard)
::  *MyFiles      =     NOFILES
                  |     FILES !Files
::  Clipboard     :==   [CopiedCell]
::  CopiedCell    :==   Entry
```

In the State definition above, the tuple-component 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 forced to be evaluated each time a call-back function delivers a new state.

5 Reflection

5.1 Lessons learnt during the implementation

The application was developed with version 0.8 of Clean. Intended as an intermediate language, the syntax of this version was rather poor. One of the reasons to start this project was to gain insight into the essential extensions that were needed towards an upgrade of Clean to a proper programming language.

Obviously, programming was hampered by the absence of well-known goodies such as local function definitions, infix expressions, overloading, ZF-expressions, pattern match wild cards and a lay-out rule.

More specific and of more general importance are the following:

- The Clean programming environment has only limited support for larger programs (a search facility which enables the user to open quickly definition or implementation modules or to find quickly the definition or the implementation of a selected function identifier). For larger projects, a project manager is required which keeps track of the modules that are part of the project and incorporates facilities like quickly finding all applications of a given function throughout (parts of) the project, printing (parts of) the project, changing layout or comments in a definition module without having to recompile all dependent modules, an option to add inferred strictness automatically to exported types in definition modules and a warning for specified strictness that cannot be inferred. Adding structure to the project defining layers in which definition modules can depend on each other might also be very helpful.
- In this project it turned out to be hard to keep track of the definitions that are available within a certain module since when the implicit import mechanism is used not only all definitions contained in the definition module of an imported module are imported but also all definitions that are imported by the imported module. So, either the project manager should assist the programmer in this or implicit imports should be abandoned (they are however very useful for importing a complete library throughout a project).
- The Clean 0.8 version has relatively primitive support for uniqueness typing. Uniqueness types are checked but not inferred. There are no ways to define, via a projection function, a read-only access on a (part of) a unique data structure without having to produce a tuple with the unique data struc-

ture and its projection. In other words, the concept of observation of a unique typed object is not present. Furthermore, for data structures that are defined by the user as being unique, the code generator does not generate code that makes use of this information.

- Lack of record definitions: when in the development process a type which is defined as a tuple (e.g. Cell, Sheet and State, see section 4.3) is extended, all functions that use pattern matching on this tuple have to be changed since the number of tuple elements changes. This can be avoided by defining access functions for all components. This has the disadvantage that pattern matching cannot be used anymore, which can make function definitions longer and harder to read.
- All I/O functions have the full program state as their argument. In many cases a large part of the state is needed only locally to the I/O function itself each time it is called. There are no language constructs to support this in the Clean 0.8 I/O library.
- The ability to define interleaved processes with a separate I/O interface would allow the programmer to give more structure to the program. The Help facility for example could then be redesigned in such a way that it could be always visible and run in a separate window with a separate menu bar accessible just by clicking on its window. In a similar way the function editor could be used side-by-side to the sheet itself.
- The Macintosh linker has a limit size of 32K for an object file to be linked into an application. It is a pain having to split up a module just because the linker cannot deal with the size of the generated code.
- There are no design rules for time and space efficiency of different language constructs. When writing an industry standard efficient application it may prove to be vital for the designer to know the influence of the used language constructs on the time and space behaviour.
- There is no support to extend specifications of I/O systems with call-back functions using subtyping to specifications of I/O systems with an extended program state. With such a facility, the text editor could be extended to a function editor in a very general way. Now the main menu specification itself had to be copied textually and it had to be changed and extended. With such a facility the main menu specification could be passed as an argument to a function which takes each call-back function of the algebraic data structure and replaces it with a new function (defined in terms of the original one via projections and extensions) that operates on the extended state.

All these critiques have been input to the design process of the Clean language version 1.0 and the new I/O library version 1.0. Apart from the 32K limit, structuring definition modules in layers, and abandoning implicit imports, all of them have been incorporated in the design. The required functionality for the spreadsheet served as an important test case in various stages of the design.

5.1.1 Higher order functions

Higher order functions were used throughout the implementation. The I/O library (with its algebraic data structure describing the I/O components and containing call-back functions for the possible events) could not have been written without the availability of higher order functions. Its *definition* modules contain many higher order functions.

Of course, there were also several cases in *implementation* modules of the use of (variants of) standard functions like fold and map with (curried applications of) functions as arguments where this was felt needed (in particular in the symbolic evaluator this occurred rather often). It is our experience that overall efficiency was not hampered by such use of higher-order functions (with the exception of the use of foldr which is inherently rather inefficient).

5.1.2 Lazy evaluation and graph rewriting

At many points in the implementation, lazy evaluation and explicit sharing were used. The most important use of the combination of these two techniques has already been treated in section 4.2.3 (in dealing with recursion in the symbolic evaluator).

An example of the use of lazy evaluation in the spreadsheet is the following. When a cell is changed, in principle all cells that depend on it have to be recalculated. However, for cells that are not visible in the window and of which the value is not used by cells that are visible, such recalculation is not necessary yet. Depending on the use, this recalculation will be required later (when the window is scrolled) or never (when the same cell is changed again). Lazy evaluation can take care of that with hardly any programming effort. The only thing which is required is that on the topmost level of interaction, the list of frames to be updated is restricted to the visible ones. Due to lazy evaluation, the calculations corresponding to invisible cells will then be delayed automatically.

Lazy evaluation is turned into strict evaluation at different points for various reasons. The *required behaviour* can be inherently strict (see the discussion on saving files in section 4.3.3) or the *interface to the outside world* can require arguments to be evaluated before they are passed (needed in many places in the I/O library) or the *memory management* of the resulting application would otherwise turn out to be unsatisfactory (used internally in the editor to avoid certain space leaks).

5.1.3 Clean I/O

The advantage of Clean I/O is its relatively direct way of interfacing to system calls. In particular for the relatively I/O intensive parts like scrolling (in the sheet or in the editor), this was important in order to achieve a proper efficiency of interaction.

It is our impression that using Clean I/O it is easier to modify and read I/O programs than using an imperative language. A large part of the debugging of the system was done by someone other than the original programmer. Due to referential transparency it was relatively easy to correct a bug as soon as it was identified as a wrong definition of a particular function. Only the definition of the function itself had to be considered and all required information was present via the arguments of the function. The absence of side-effects proved to be very useful for debugging the program.

Apart from having the advantage of referential transparency, the user can relatively easily define higher levels of abstraction. This can be done both on a small scale defining useful higher order extensions of the I/O library (e.g. for often used dialogues) as well as on a large scale on which a user could define a new style of I/O.

5.2 Performance and code sizes

5.2.1 Code sizes

FunSheet requires 4 Megabytes 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 spreadsheet application is constructed by combining and adapting existing software components written in a lazy functional programming language. The project described here consisted of designing and implementing the sheet and cell manipulation part (performed by an M.Sc. student) and improving and extending the symbolic evaluator part (performed by a Ph.D. student). Taken together the project took about 10 student months.

The source code of FunSheet is organised in six major parts: sheet and cell manipulation, editor, symbolic evaluator, I/O library, help tool, and standard environment (including the basic environment). The standard environment is written in the spreadsheet language. It takes about 560 lines, or about 15 kilobytes (kB). When the system is compiled, the files of the standard environment are translated to Clean modules, which are then compiled to object code. The generated implementation modules take about 99 kilobytes and the generated definition modules take about 9 kilobytes. The size of the standard environment is about 14% of the size of the corresponding generated Clean modules.

	source size			*object size*	
	lines	kilobytes	%(kB)	kilobytes	%(kB)
sheet and cell manipulation	3518	129	14	390	15
editor	4774	181	19	417	16
symbolic evaluator	2378	76	8	510	19
I/O library	11594	458	48	925	35
help tool	193	8	1	16	1
translated standard environment	1733	99	10	362	14
Total	24190	951	100	2620	100

The source sizes in the table above do not include the sizes of the definition modules. These modules take about 5200 lines of Clean code, or about 150 kilobytes. This is 16% of the corresponding implementation modules. The size of the combined implementation and definition modules is about 29400 lines, or about 1100 kilobytes. When the spreadsheet application was implemented, the editor and I/O library were already available. The size of their implementation modules is about 67% of the size of all spreadsheet implementation modules. Of course, for the required functionality of the spreadsheet it would have been possible to use many fewer lines if existing code was not reused (the editor and the I/O library are quite general). With the conversion to Clean 1.0, the number of lines is expected to decrease significantly due to the larger expressive power of the high level syntactical constructs present in Clean 1.0 (e.g. a single ZF-expression or record definition can replace several function definitions for construction, filtering, access and update of the data structures).

The application size itself is approximately 1 Megabyte. The spreadsheet application, the stand-alone version of the editor, and the Concurrent Clean System are freely available for non-commercial use via FTP (pub/Clean at ftp.cs.kun.nl) or WWW (www.cs.kun.nl/~clean). The Concurrent Clean System is available for several platforms (Macintosh, PC, Sun3, and Sun4). The FunSheet application runs on a Macintosh only since for the use of non-scrolling margins in windows, a small extension was made to the library which is not yet ported to the other platforms. This extension will be incorporated in the new library that is being made with the Clean 1.0 system.

5.2.2 Efficiency of interpreted and translated FunSheet programs

The standard environment is translated to Clean code to increase its execution efficiency. Let us take the nfib function as an example. The definition of nfib in the FunSheet language is

```
nfib n  =  1  if   n <= 1
        =  nfib (n - 1) + nfib (n - 2) + 1;
```

A measure for the number of function calls per second of an implementation is the nfib number. The nfib number is equal to the limit of nfib n divided by the time in seconds to compute nfib n, for n approaching infinity. On a 33 MHz 68030 Macintosh, the nfib number of the interpreted definition is about 700. If the definition is made part of the standard environment, it will be translated to following Clean code when the FunSheet application is built.

```
$nfib n    =  $if  ($<= n 1)
                   (INT 1)
                   ($+ ($+ ($nfib ($- n (INT 1))) ($nfib ($- n (INT 2)))) (INT 1))
```

On the same machine the nfib number of the translated definition is about 7000. Because the spreadsheet language is untyped, the translated definition is strewn with type tags. Therefore it is still two orders of magnitudes slower than the following nfib function written in Clean. Its nfib number is about 700000 on the same machine.

```
nfib n  |  n <= 1       =    1
        |  otherwise    =    nfib (n - 1) + nfib (n - 2) + 1
```

5.3 Evaluation

As an application FunSheet is positioned somewhere 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 spreadsheet using arrow keys, and scrolling the spreadsheet when necessary. With respect to these aspects the efficiency is about the same as the efficiency of Excel.

+/- The function evaluation efficiency of the spreadsheet language 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 good if one considers that symbolic evaluation is employed on untyped expressions. However, the sheet evaluation mechanism which deals with computing all effects of a cell change is an order of magnitude slower than Excel. The used representation of the matrix of cells as a list of lists is probably the main cause of this. The function evaluation mechanism could not be compared with Excel since Excel only has a macro facility which is defined in such a way that the parameters are in fact global variables, giving rise to unwanted semantics when recursion is used.

\- The current application is a first version which has not yet gone 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/print/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.

5.4 Future improvements/extensions

It is the intention to include diagram, print and report facilities in a future version of FunSheet.

Furthermore, a concept with similar capabilities as relative addressing (which means that special facilities are provided to address cells in a way which is relative to the current cell) will be incorporated. Classically, there are two ways to provide relative addressing: by including a special syntax for the address of a cell relative to the current cell (this would invalidate referential transparency) or by changing certain addresses in cell formulae in a relative way while copying these formulae to other cells (this would imply implicit context switches). It is our intention to investigate whether an *explicit* method might be a useful extension. One method to be considered is to allow the user to explicitly apply any function on a list of copied cells while pasting them: a general feature which could be used for 'relative addressing' and for other purposes. Another method to be considered is to allow the user to edit column functions: e.g. `A 1 = 1; A 2 = 1; A i = A (i - 1) + A (i - 2);` could be a way to define that the cells in column `A` have to be filled with the fibonacci numbers. This would allow changing dependencies via standard edit functions.

The code (Clean 0.8) will be converted to Clean 1.0 not just by using the automatic conversion facility but by employing the new features available in Clean 1.0. Apart from more readable code due to the availability of more syntactic sugar, an important advance is expected due to the use of observation types and of user-defined unique data structures. The use of a destructive array (defined with uniqueness types) for the cell matrix instead of a list of lists is expected to greatly improve the overall efficiency. Due to the propagation property of uniqueness (Smetsers *et al*. (1993)), the type for `Sheet` must then also be unique since it contains a unique component (destructing the component will destruct the surrounding structure too).

The interfaces between the different components are intended to be redesigned such that the interface to a component will be fully contained in one definition module while compiling the corresponding implementation module separately will yield a stand alone application of the component. In practice, this will make it easier to guarantee that the interface is kept stable while the component changes.

Finally, it is our intention to develop a distributed version in which different parts of a sheet can be changed and updated on different processors.

5.5 Related work

None of the spreadsheets we could find in the literature incorporated symbolic evaluation.

Nas (Wray & Fairbairn (1989)) is an interactive functional program which is based on ideas from spreadsheets. Its language is first order. It incorporates a way to address the previous value of a cell: a feature which made it possible to model a flip-flop constructed from NOR-gates within the system.

The concept of time is modelled as an extra dimension within a first order functional dataflow spreadsheet in (Du & Wadge (1990)). Their system is based on intensional logic.

---

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

The inherent concurrency of a spreadsheet computation is explored in (Yoder & Cohn (1994)). They allow first order recursive function definitions. Their paper argues that spreadsheet languages can offer both intuition and access to parallel computational resources.

In (Harvey & Wright (1994)), the authors describe a simple spreadsheet written in Scheme. The implementation serves as an example for Scheme students. To avoid a lot of complexity, the spreadsheet has no graphical user interface. Also, no dependency structure between cells is implemented. Therefore every cell will be re-evaluated when a cell is changed by the user of the spreadsheet. Lazy functions are not incorporated in the spreadsheet. Finally, user-defined functions can only be added to the source code of the system.

In (Boon (1994)), the author reflects on advantages and disadvantages of spreadsheets. The particular advantages of a functional spreadsheet are discussed and an implementation is described of a functional spreadsheet written in Scheme. The sheet incorporates dependency computations, user-defined functions, and higher order functions.

6 Acknowledgements

The authors would like to thank the *referees* and the *editors* for their valuable comments, corrections and suggestions. They really made a difference.

7 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 commercial spreadsheets but after extension and improvement through more extensive user experience it may find its own niche between calculators and existing commercial spreadsheets.
- Reuse and adaptation of *existing software components* (I/O library, help tool, editor and symbolic evaluator) turned out to be possible with a functional language.
- The lack of side-effects made *debugging* relatively straightforward.
- However, the experience with the project did yield a number of important aspects of the language Clean and its programming environment (version 0.8) that *hampered the software development*. The experience of this project showed the importance of incorporating these aspects in future versions of the language. Clean 1.0 will incorporate most of them.
- Considering the relatively small scale of this project the *software productivity* of the project was quite high.
- Considering the functionality with respect to user interaction and symbolic evaluation and the facts that huge parts of the software were reused and that code generation for unique data types was not available yet the *efficiency* of FunSheet is satisfactory.

References

Achten, P.M., and Plasmeijer, M.J. 1995. The Ins and Outs of Clean I/O, *Journal of Functional Programming*. **5** (1).

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. (editors), *Proceedings of Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, LNCS **259**, Vol.II., pp. 141-158. Springer-Verlag.

Barendsen, E., and Smetsers, J.E.W. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract). In Shyamasundar, R.K. (editor), *Proceedings of the Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, Bombay, India,, December 15-17, 1993. LNCS **761**, pp. 41-51. Springer-Verlag.

Boon, J. 1994. A Purely Functional Spreadsheet. Third Year Project Report. University of York.

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. (editor), *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, pp. 364-384. Springer-Verlag.

Carlsson, M., and Hallgren, Th. 1993. Fudgets - A Graphical User Interface in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* Copenhagen, Denmark, June 9-11, 1993, pp. 321-330. ACM Press.

Cartwright, R., and Boehm, B. 1990. Exact Real Arithmetic, formulating real numbers as functions. In Turner, D.A. (editor), *Research Topics in Functional Programming*. University of Texas at Austin Year of Programming Series, pp. 43-64. Addison Wesley Publishing Company. ISBN 0-201-17236-4.

Casimir, and Rommert J. 1992. Real Programmers Don't Use Spreadsheets. *ACM SIGPLAN Notices* **27** (6): pp. 10-16.

Du, W., and Wadge, W.W. 1990. The Eductive Implementation of a Three-Dimensional Spreadsheet. *Software-Practice and Experience*. **20**(11): pp. 1097-1114.

Harvey, B., and Wright, M. 1994. *Simply Scheme: introducing computer science.* MIT Press. ISBN 0-262-08226-8.

Hoon, W.A.C.A.J. de 1993. Designing a spreadsheet in a pure functional graph rewriting language, Computer Science Master Thesis nr. 300, University of Nijmegen, The Netherlands.

Hoon, W.A.C.A.J. de, Rutten, L.M.W.J., and Eekelen, M.C.J.D. van 1994. FunSheet: A Functional Spreadsheet. *Proceedings of the Sixth International Workshop on the Implementation of Functional Languages*, Norwich, UK, September 1994, pp. 11.1-11.24.

Litecky, C. 1990. Spreadsheet Macro Programming: a Critique with Emphasis on Lotus 1-2-3. *Journal of Systems and Software* **13** (3): pp. 197-200.

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. (editors), *Proceedings of Parallel Architectures and Languages Europe*, June 1991, Eindhoven, The Netherlands. LNCS **506**, pp. 202-219. Springer-Verlag.

Peyton Jones, S.L., and Wadler, Ph. 1993. Imperative Functional Programming. In *Proceedings of the Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 10-13, 1993, pp. 71-84.

Plasmeijer, M.J., and Eekelen, M.C.J.D. van 1993. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley Publishing Company. ISBN 0-201-41663-8.

Plasmeijer, M.J., and Eekelen, M.C.J.D. van 1995. Clean 1.0 Reference Manual. Technical Report, in preparation. University of Nijmegen, The Netherlands.

Pountain D. 1994. Functional Programming Comes of Age. *Byte Magazine*, **19**(8): pp. 183-184.

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. (editors), *Proceedings Workshop Graph Transformations in Computer Science*, Schloss Dagstuhl, January 4-8, 1993. LNCS **776**, Springer-Verlag.

Turner, D.A. 1990. An Approach to Functional Operating Systems. In Turner, D.A. (editor), *Research topics in Functional Programming*, pp. 199-217. Addison-Wesley Publishing Company, ISBN 0-201-17236-4.

Vuillemin (1987). Arithmétic réelle exacte par les fractions continues. Technical Report 760. Institute National de Recherche en Informatique et en Automatique. Domaine de Voluceau, Roquencourt, BP105, 78153 Le Chesnay Cedex, France.

Wray, S.C., and Fairbairn J. 1989. Non-strict languages - Programming and Implementation. *The Computer Journal*. **32** (2). pp. 142-151.

Yoder, A., and Cohn, D.L. 1994. Real Spreadsheets for Real Programmers. Technical Report 94-9. University of Notre Dame, Notre Dame.