Constructing Medium Sized Efficient Functional Programs in Clean

Marko C.J.D. van Eekelen, Rinus (M.) J. Plasmeijer
Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
e-mail: marko@cs.kun.nl, rinus@cs.kun.nl

Abstract

As functional programming comes of age, writing medium sized functional programs (i.e. programs in the range of 10.000 to 100.000 lines of source code) becomes a realistic task. As a test case for development techniques for medium sized efficient functional program these notes describe the experience with writing in the functional language Clean a functional spreadsheet, i.e. a spreadsheet which has as its cell expression language a lazy functional programming language with the ability for the user to define lazy higher-order recursive functions.

An important aim of the design was to reuse existing functional software as much as possible. The resulting application uses about 25000 lines of Clean combining general components such as a window-based text editor, a symbolic evaluator and a high-level I/O library.

The design of the spreadsheet application (FunSheet) is shortly introduced, experience with development techniques for this application is discussed and some examples are given of general techniques for writing medium-sized functional programs that may be used in future experiments.

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 guarantee 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).

Section 2 introduces the test case spreadsheet application (called **FunSheet**). The used development techniques for increasing efficiency and for writing medium sized functional programs are discussed in section 3. Some examples of techniques for writing efficient medium-sized functional programs that may be used in the future, are given in section 4 after which conclusions are drawn in section 5.

2 FunSheet: a Spreadsheet Application

The spreadsheet application FunSheet is constructed by combining and adapting existing software components written in the lazy functional programming language Clean (version 0.8). 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 project could be finished within this period partly because the lack of side-effects made debugging relatively straightforward.

In this section we introduce the reader to this application and its implementation since it has formed (and will form in the future) as a test case for techniques for writing efficient medium sized functional programs. For more information and motivation the reader is referred to De Hoon *et al.* (1995).

2.1 Design

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 (see section 2.2). 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.

Basic Idea of the FunSheet Application

Each sheet 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. Figure 1 gives a typical user's view of the program.

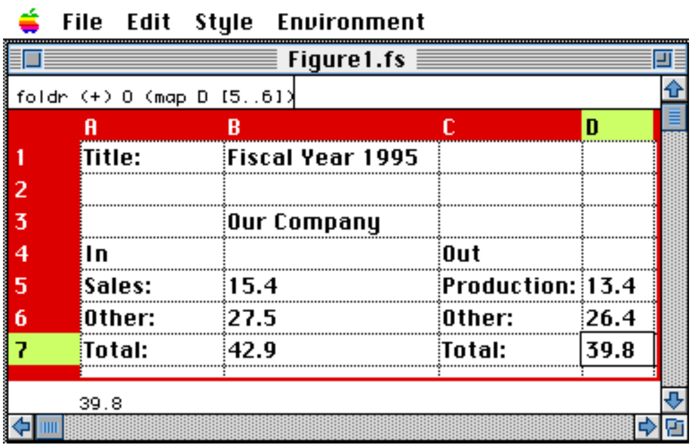

Fig. 1. A user's view of FunSheet

FunSheet 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, *sheet editing* actions, *remote values* defined in other sheets, manipulation actions for *labels* as verbose synonyms for references to a (block of) cell(s), *formatting* actions, a facility to *select* (user-defined or predefined) *functions* and an on-line *Help* facility.

An important aspect of the design is the built-in *function editor* with which the possibility is created to *define new functions* by switching to this function editor with which for each sheet a separate set of user-defined functions can be created.

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.

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 and selecting characters, words and lines.

2.2 A Purely Functional Spreadsheet Language

In contrast to the macro-facilities of standard spreadsheets 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).

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, range 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 cell range expressions (denoted using ... instead of ..) are available to denote blocks of cells. Lists are a predefined data structure. Besides using the notation hd : tl for a list, the equivalent notation [hd | tl] is also allowed. Algebraic data structures can be defined. Most standard operators on these data structures have been included in the language. A number of standard functions is predefined. The language does not have an off-side rule. For more information on the language the reader is referred to (de Hoon *et al*. (1994)).

Cell References and Dependencies

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).

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. Column functions can be curried and they can be used just as any other function in cell expressions.

The spreadsheet design avoids having to update the whole sheet when the entry of a cell changes by maintaining dependency information. 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. 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 (e.g.: 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 map (twice (twice A)) [1..4] would be allowed in a cell in column B but not 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).

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*.

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. 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.

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 (e.g. (x-y)*(x+y) == x^2 - y^2 will yield True).

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. This anomaly can be removed 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). Such a list may not only contain symbolic values but it may be generated using symbolic values in a dot-dot expression.

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 blocks of cells. 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 $x_1\ x_2\ \ldots\ x_n \setminus e$ 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.

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.

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 the case in which the sheet is set up such that while the cells that are referred to are still empty, symbolically the sum of the sums of rows is checked to be equal to the sum of the sums of columns. This shows how a user can *prove* that a particular set-up of a spreadsheet has a required property by adding symbolic equations.

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).

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.

2.3 Implementation

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.

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 2 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 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 ClsAll,
  MenuSeparator,
  MenuItem HelpId   "Help"   (Key '/') Able   Help,
  MenuSeparator,
  MenuItem QuitId   "Quit"   (Key 'q') Able   Quit ]
```

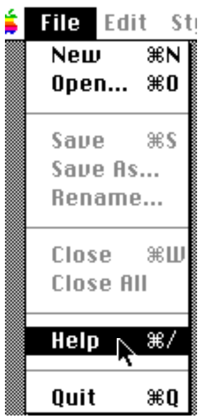

Fig. 2. FunSheet's File-menu definition

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.

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). 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)
```

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 3.8) there is no difference in the evaluation mechanism for the various environments mentioned above. Evaluation is done entirely symbolically.

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.

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
```

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 using lambda lifting.

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.

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.

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.

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 have some *identification* information for each 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 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
    :== (Ident,Matrix Cell,Interaction,Row,Col,[Label],ParseEnv,Font)
```

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. 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.

2.4 Performance

With respect to I/O the efficiency is about the same as the efficiency of Excel: 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.

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.

2.5 Code Sizes

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.

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.

---

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

2.6 Availability and applicability

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.

To execute FunSheet 4 Megabytes of free memory is required. This is quite huge for such a program. 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 and the stand-alone version of the editor are freely available for non-commercial use via FTP (pub/Clean at ftp.cs.kun.nl) or WWW (www.cs.kun.nl/~clean).

2.7 Future Improvements/Extensions

It is the intention to include in a future version of FunSheet diagram, print and report facilities and an explicit method to provide capabilities similar to relative addressing as present in standard spreadsheets.

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 (which simplify the definition of read-access on unique data structures) 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 using the techniques in section 4 in such a way 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.

3 Experience with Development Techniques for FunSheet

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. Furthermore, there were 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.

The required functionality for the spreadsheet served as an important test case in various stages of the design. Many of the critiques have been input to the design process of the Clean language version 1.0 and the new I/O library version 1.0.

3.1 Modular Design

Since the design was set out to re-use existing software as components in the implementation, the implementation is modular and highly structured.

The main components access each other only through interface modules defining abstract data structures with access functions. When in the development process such a type which was defined as a tuple (e.g. `Sheet` and `State`, see section 2.3) was extended, all functions that use pattern matching on this tuple had to be changed since the number of tuple elements changed. The presence of records in the language would have had a significant benefit on the development process.

It has proven to be good practice to present a design technically by producing the required definition modules. The module structure containing the definitions of the data types with the type definitions of the defined functions gives a good insight in the set-up of the design.

However, the implicit import facility (which imports all definitions known in a module and can be applied recursively) in definition modules made it 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.

The module structure had to be changed during the development process for technical reasons: 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.

The Clean programming environment has only limited support for larger programs consisting of many modules (all it has is 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, more programming environment support is required (see section 4.4).

Adding Process Structure

The ability to define interleaved processes with a separate I/O interface as described in Achten and Plasmeijer (1994) would allow the programmer to give more structure to the program.

In FunSheet 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.

3.2 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).

3.3 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 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 top-most 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. This will not lead to a continuous accumulation of space consumption (sometimes referred to as a space leak) because the list of update frames can never be larger than the number of cells in the sheet (which is finite).

3.4 Strictness

Lazy evaluation is turned *by the programmer* into strict evaluation at several points for various reasons. The *required behaviour* can be inherently strict (see the discussion on saving files in section 2.3) 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).

In several cases the use of the basic function foldl instead of foldr proved useful to create efficient left-recursive *derived* strict evaluation.

3.5 Uniqueness

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 structure 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.

The Clean 1.0 design has incorporated the suggested changes in this section.

3.6 Clean I/O

The advantage of Clean I/O is its 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, due to referential transparency and the use of higher order functions, using Clean I/O it is easier to modify and read I/O programs than using an imperative language.

However, 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. The Clean 1.0 library will support local state in I/O components.

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.

3.7 Debugging

A large part of the debugging of FunSheet 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. The programming environment facility to open the definition or the implementation module of a selected function (displaying the type or the full definition correspondingly) proved to be indispensible in this context. No need was felt for special debugging facilities.

3.8 Replace Interpretation by Compilation

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] = ncondition ps vs
condition [p:ps]     [v:vs] = or (pat_cond p v) (condition ps vs)
condition []         []     = F "False" []

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

select_args:: [Value] Int Value -> [Value]
select_args []       i v  =  []
select_args [a : as] i v
   =  [F "nth_argument" [F (ToString i) [],v]:select_args as (i+1) 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_arg :: Int Value -> Value
nth_arg 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_arg 1 v) || variable (nth_arg 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_arg 1 n2) || variable (nth_arg 2 n2)))) ||
    variable (nth_arg 2 v)))                     = F "f" [v]
where
      n1 = C ":" [INT 0, C "[]" []]
      n2 = nth_arg 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 v1 ... vn   =  F "f" [v1, ..., vn]
```

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

Efficiency of Interpreted and Compiled FunSheet Programs

The standard environment is translated to Clean code to increase its execution efficiency. Let us take the nfib function which produces as its result the number of times the function was called in the recursion as an example to serve as a 'poor man's benchmark' of the number of functional calls per second (the nfib number).

On a 33 MHz 68030 Macintosh, the nfib number of the interpreted definition (defined by the user) is about 700.

If the definition is made part of the standard environment, it will be translated to Clean code when the FunSheet application is built. Then, on the same machine the nfib number of the translated definition is about 7000, an order of magnitude faster than the interpreted definition.

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 nfib function when written directly in Clean. Then, its nfib number on the same machine is about 700.000, three orders of magnitude faster than the interpreted definition.

3.9 Combining Interactive Applications

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) 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 shstate io    =  (newshstate, nio)
where
   newshstate     = AdaptShFunctionEnv  newfunenv shstate
   newfunenv      = GetEdFunctionEnv edstate
   (edstate,nio)  = NestIO IOSystemEd (InitEdState funenv) InitIOEd io
   funenv         = GetShFunctionEnv sheetstate
```

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 IOSystemEd InitEdState InitIOEd io
```

Extending Interactive Applications With Re-compilation

With the nesting scheme above, in Clean 0.8 it was necessary to change the editor's program state which was a tuple. It had to be extended with an extra field containing the function environment.

```
::*Ed -> (Disk,Defaults,Clipbrd,FindInfo,EditWdIds,EditWndws,Env)

// old definition:
//::*Ed  -> (Disk,Defaults,Clipbrd,FindInfo,EditWdIds,EditWndws)

InitEdState:: Files Env -> Ed;
InitEdState files funenv
   = (files,defs,InitClipbrd,InitFindInfo,InitWdIds,InitEdWds,funenv)
where
   defs       =  (DefTabWidth,(ft,sz),DefAutoIndent)
   (ft,_,sz)  =   DefaultFont

GetFunEnv:: Ed -> (Ed, Env)
GetFunEnv ed:(fls,ds,cb,fi,is,ws,env) = (ed,env)

SetFunEnv:: Env Ed -> Ed
SetFunEnv funenv (fls,ds,cb,fi,is,ws,env)
```

```
   = (fls,ds,cb,fi,is,ws,funenv)

GetDefaults:: Ed -> (Ed, Defaults)
GetDefaults ed:(_,defaults,_,_,_,_,_) = (ed,defaults)

SetDefaults:: Defaults Ed -> Ed
SetDefaults deflts (fls,_,cb,fi,is,ws,env)
   = (fls,deflts,cb,fi,is,ws,env)
```

All access functions (including initialisation and closing down) that were used by the editor had to be changed. Since the editor's program state was implemented as an abstract data type, the rest of the editor's definitions could remain unchanged (they just had to be recompiled) with the exception of the definition of the menu system which had to be changed in order to incorporate the new functions for checking and testing function definitions. An interesting aspect of the new menu system definition is that it contains many of the unchanged old definitions which effectively only operate on the old part of the extended state.

```
menus:: DeviceSystem Editor IO
menus =  MenuSystem [file, edit, search]
where
 file
  = PullDownMenu MFileID "File" Able
    [MenuItem IHelpID "Help..."    (Key '/') Able     SSHelp,  // new
     MenuSeparator,
     MenuItem CKID "Syntax Check" (Key 'S') Able CheckFunction,// new
     MenuItem TestID "Expression Test" (Key 'E')    Able  Test, // new
     MenuSeparator,
     MenuItem IQuitID "Return to Sheet" (Key 'W') Able SSClose // new
    ]                                               // no quit anymore
 edit
  = PullDownMenu MEditID "Edit" Unable
    [MenuItem IUndoID  "Undo"      (Key 'Z')  Able     Undo, // old
     MenuSeparator,
     MenuItem ICutID   "Cut"      (Key 'X') Unable   Cut,    // old
     MenuItem ICopyID  "Copy"     (Key 'C') Unable   Copy,   // old
     MenuItem IPasteID "Paste"    (Key 'V') Unable Paste,    // old
     MenuItem IClearID "Clear"      NoKey  Unable Clear,     // old
     MenuSeparator,
     MenuItem IBalanID "Balance" (Key 'B') Able Balance,     // old
     MenuSeparator,
     MenuItem IFormaID "Format"  (Key 'J') Able  Format      // old
    ]
 search
  = PullDownMenu MSearcID "Search" Unable
    [MenuItem IFindID  "Find..."   (Key 'F')  Able     Find,     //old
     MenuItem IFindNID "Find Next" (Key 'G') Unable  FndNext,   //old
     MenuItem IFindSID "Find Selection" (Key 'H') Unable FndSel,//old
```

```
MenuItem IReplID "Replace&Find" (Key 'T') Unable Repl&Find,//old
MenuSeparator,
MenuItem IGotoCID "Goto Cursor" (Key 'L') Able GotoCursor, //old
MenuItem IGotoLID "Goto Line..."  NoKey  Able GotoLine    //old
]
```

So, for the extension of a text editor to a function editor changes were done in only two modules: the module that defined the abstract program state and the module that defined the I/O system and started the interaction. The rest was recompiled.

A disadvantage of this recompilation is that a copy has to be made if both editors co-exist on the same machine. This requires version management if changes are made in the future.

4 Development Techniques for Efficient Medium-Sized Functional Programs

In this section for several important development techniques that could have been used in the FunSheet project or may be used in future projects, examples are given of the way they can be applied in a functional language.

All program examples in this section are written in Clean 1.0. Below some general remarks are made about the Clean 1.0 and the differences with respect to Clean 0.8. The key constructs used in the examples are explained briefly. For further reference on the syntax and semantics of the constructs available in Clean 1.0 the reader is referred to FTP (pub/Clean at ftp.cs.kun.nl) or WWW (www.cs.kun.nl/~clean) where the system is freely available for non-commercial use.

About Concurrent Clean 1.0

Compared with the previous version of Clean a lot of new features are added based on experience with writing complex applications. Many of the added language constructs are similar to those commonly found in other modern lazy functional languages (such as Miranda (Turner, 1985), SML (Harper *et al.*, 1986), Haskell (Hudak *et al.*, 1992) and Gofer (Jones, 1993)). People familiar with these languages should have no difficulty to program in Clean and we hope that they enjoy the compilation speed and quality of the produced code.

In addition Clean offers a couple of very special features. Of particular importance for practical use is Cleans' uniqueness typing enabling the incorporation of *destructive updates* of arbitrary objects *within a pure functional framework* and the creation of *direct interfaces with the outside world.*

Cleans "unique" features have made it possible to predefine (in Clean) a sophisticated and efficient I/O library. The Clean I/O library enables a Clean programmer to *specify interactive window based I/O* applications on a *very high level of abstraction.* The library forms a platform independent interface to window systems which makes it possible to port window based I/O applications written in Clean without modification of source code.

In Clean it is possible to create *processes*. The new Clean I/O library takes advantage of this feature such that it is now also has become possible to develop *distributed executing interactive applications* running on several PC's/workstations connected in a network. The applications can communicate via *asynchronous as well as synchronous message passing*. Such a distributed application can be developed on one processor on which the processes will run in an *interleaved* fashion. This is very handy for testing.

The new Clean compiler still combines *fast compilation* with the generation of *efficient code* and is available on an increasing number of platforms (Mac, PC, Sun).

Major differences with Clean 0.8

Compared with the previous release (0.84b) many important changes have been made (there is a noticeable difference between an intermediate language and a programming language).

The most important changes in the language are:

- various syntactic sugar is added (infix operators, a case construct, local function definitions, lamda-abstractions, list comprehensions, lay-out rule, CAF's etc.);
- overloaded functions, type classes and type constructor classes can be defined;
- records and arrays are added as predefined data structure with handy operations (such as an update operator for arrays and records, array comprehensions etc.);
- a more refined control of strictness is possible (partially strict data structures can be defined for any type, in particular for recursive types, there is strict let construct);
- existentially quantified types can be defined;
- the uniqueness typing is refined (now polymorphic and inferred, observation of uniquely typed objects is made easier);
- there is support for destructive updates of predefined and user defined data in a pure functional context;
- the semantics for parallel evaluation is adapted for uniqueness typing and its use is simplified as well;
- the module structure is improved;
- the macro facility is extended.

Also the Clean I/O library has been changed:

- the I/O library is improved (with respect to orthogonality, modularity, extendibility, portability);
- the I/O library is extended allowing to define interactive processes running interleaved inside one application which can communicate via files, shared data, (a)synchroneous message passing, remote procedure call;

❑ one can define interactive processes which run distributed on workstations connected via a network.

The compiler and code generator have been extended and are partly rewritten.
Furthermore,

❑ the code generator is extended for parallel and distributed evaluation;
❑ the code generator is improved;
❑ more platforms are supported.

Some remarks on the new Clean syntax

Compared with the 0.84 version we have made a lot of syntactic changes to the language. The complete redesign of Clean has as consequence that Clean version 1.0 is *not* compatible with its predecessors. A Clean application is available to transform programs written in old Clean into new Clean.

The new Clean syntax is similar to the notation found in most other modern functional languages. So people familiar with these languages will have no difficuluties with programming in Clean. However, there are a couple of small differences we want to point out here for people who don't like to read language reports.

In Clean the arity of a function is reflected in its type. When a function is defined its uncurried type is specified (to avoid any confusion we want to explicitly state here that in Clean there is no restriction whatsoever on the curried use of functions).

```
The standard map function (arity 2) is specified in Clean as follows:

map::(a -> b) [a] -> [b]
map f []      = []
map f [x:xs]  = [f x : map f xs]
```

In types funny symbols can appear like `.`, `u:`, `*`, `!` which can be ignored and left out if one is not interested in uniqueness typing or strictness.

Each predefined structure such as a list, a tuple, a record or array has its own kind of brackets: lists are *always* denotated with square brackets `[...]`, for tuples the usual parenthesis are used `(...,...)`, curly braces are used for records `{...}`, and arrays look like this `{: ... :}`. There are only a few keywords in Clean leading to a heavily overloaded use of `:` and = symbols:

```
function::argstype -> restype  //  type specification of a function
function pattern | guard = rhs //  definition of a function

selector = graph               //  definition of a constant/ CAF/graph

::type args   =     type       //  an algebraic type definition
::type args   :==   type       //  a type synonym definition
::type args                    //  an abstract type definition
```

```
macro args    :==    rhs          //  a macro definition (function synonym)
```

Defining algebraic data types with existentially quantified variables

Clean incorporates the extension of the Hindley/Milner type system with the possibility of algebraic types to be existential. An *existential algebraic data type definition* is an algebraic type definition in which *existentially quantified variables* are used. These special variables are marked with "E.". Existential types are useful if one wants to create (recursive) data structures in which objects of *different types* are being stored (e.g. a list with elements of different types). Such kind of data structures are for instance used internally in Cleans' I/O library to store (program and I/O) states of different types and state transition functions defined on these states in one data structure.

```
// Existential algebraic type definition and its use

::Tree E.a = NilTr
           | NodeTr (a, a, a a -> a) (Tree Void) (Tree Void)

F :: Tree Void
F = NodeTr (3,5,(+)) (NodeTr (3.5,5.4,(+)) NilTr NilTr) NilTr

G :: (Tree Void) -> Tree Void
G (NodeTr (obj1,obj2,func) l r)=NodeTr (func obj1 obj1,obj2,func) l r
```

There are severe limitations imposed on the use of data structures of *existential types*. Once a data structure of existential type is created and is passed to another function it is generally statically not possible to determine what the actual type is of those components of the constructor that correspond to the existential quantified variables.

- Therefore, it is *not* allowed to pass such objects to other functions as argument or result *if* these functions either require or deliver this actual type. In other words, for the type inference system an existentially quantified type variable is treated as a type variable that can be unified with a concrete type (= not a type variable) *only* at the explicit creation of a data structure of this type with its defining data constructor. In all other contexts an existentially quantified type variable can only be unified with non concrete types (type variables).
- For software engineering reasons it is required that an existentially quantified type variable is instantiated with the predefined type `Void` (see the example above).

Components that correspond to the *same* existentially quantified type variable will have the *same* type. So, it is allowed to apply these components in expressions that yield an ordinary type. It is also allowed to use the components to create a new object of existential type. Furthermore, it is allowed to pass the existentially quantified type variable to polymorphic functions.

Apart from the restrictions mentioned above existential algebraic types are not different from standard algebraic types. They can be used e.g. as the basis of record types, synonym types and abstract types.

Defining record types

A *record type* is basically an algebraic data type in which exactly one constructor is defined. Special about records is

- that a *field name* is attached to each of the arguments of the data constructor;
- that they cannot be used in a curried way.

Compared with ordinary algebraic data structures the use of records gives a lot of notational convenience because the field names enable *selection by field name* instead of *selection by position*. When a record is created *all* arguments of the constructor have to be defined but one can specify the arguments in *any* order. Furthermore, when pattern matching is performed on a record, one only has to mention those fields one is interested in. Existentially quantified type variables are allowed in record types. The arguments of the constructor can optionally be annotated as being strict. The specification of uniqueness attributes is also optional.

As data constructor for a record the name of the record type is used internally.

- The semantic restrictions which apply for algebraic data types also hold for record types.
- The field names inside one record all have to be different. It is allowed to use the same field name in different records.

```
// A record definition:

::Complex  =  {  re :: Real,
                 im :: Real }
```

Defining overloaded function types and concrete instances

With an *overloaded function type definition* one defines the *type scheme* of the overloaded function. The type of a concrete function must be an instance of this type scheme. The special *type scheme variable* defines in which variable the scheme can vary. With an *instance* declaration one defines an overloaded function c.q. operator name to be a *synonym* for some *concrete function* or *operator*. In the instance definition it is specified which concrete function is ment and for which *concrete type* an instance of the overloaded function is created. The type of the concrete function must be equal to the overloaded type after uniform substitution of the specified concrete type for the type scheme variable. For a concrete function one can refer to a function which has already been defined elsewhere or one can define a new function right on the spot.

One can define as many instances as one like. Instances can be added in any module. One and the same concrete function can be used as instance for different overloaded functions (as long as the types match).

```
/*
```

```
Defining an overloaded operator and instantiations with existing
concrete operators: The types of the concrete operators +^ and +.
need to be instances of the type scheme of + (take for the type
scheme variable a of the overloaded operator + respectively Int and
Real)
*/
overload (+) infixl 6 a :: a a -> a

instance + Int   = +^
instance + Real  = +.

(+^) infixl 6 :: !Int  !Int    ->  Int
(+.) infixl 6 :: !Real !Real   ->  Real
```

When an overloaded name is encountered in an expression, the compiler will determine which of the corresponding concrete functions/operators is meant by looking at the concrete type of the expression. From this type the concrete function to apply is determined. All concrete functions/operators of an overloaded function/operator must therefore be defined on *different* instances of the type scheme (with exception of the default instance, see below). If it is clear from the type of the expression which one of the concrete function is ment the compiler will in principle substitute the concrete function application for the overloaded one such that no efficiency is lost.

A concrete function is substituted for an overloaded one: given the definitions above the function

```
inc n = n + 1
```

It will be internally transformed into

```
inc n = n +^ 1
```

If it is not clear from the type of the expression which concrete function is ment (more than one of the concrete functions fit type technically) the compiler will make specialized versions of the function of which the expression is part of. In principle one version is made for each concrete possible substitution. However, the compiler will avoid making versions which are not being used or which are not important in terms of efficiency and in any case it will avoid code explosion.

As an example of the creation of specialized versions for overloaded functions: assume the following function definition and Start rule:

```
sumlist [x:xs] [y:ys]  =  [x + y:sumlist xs ys]
sumlist x y               =  []

Start = sumlist [1..10] [11..20]
```

From the context of sumlist it is not clear which concrete instantiation of + to be used. So the compiler will in principle generate a special version of sumlist for all possible versions which are needed. In this example sumlist is only applied to integer lists. So the compiler will only generate a version of sumlist for Ints as follows:

```
sumlistI [x:xs] [y:ys]    =  [x +^ y:sumlistI xs ys]
sumlistI x y           =  []

Start = sumlistI [1..10] [11..20]
```

It is possible to specify a function as *default instance* (no concrete type instance is specified for the type scheme variable in the instance declaration in that case) which will be taken when none of the other defined instances happens to be applicable. Since such a function must work for *any* instance the type of the default function must be equivalent to the type of the overloaded function. The default function provides the possibility to define a standard interpretation for an overloaded function.

An example of defining a default instance indicating that objects are by default unequal unless specified otherwise is given below:

```
overload (==) infix 2 a :: a a -> Bool

instance ==   = DefaultEqual

(DefaultEqual) infix 2 :: a a -> Bool
(DefaultEqual) x y = False
```

An alternative equivalent solution avoiding the introduction of a new function name:

```
overload (==) infix 2 a :: a a -> Bool

instance ==
where
   (==) x y = False
```

When one exports instances of an overloaded function or operator in a definition module one may wish to hide the actual function/operator name (in the implementation module). In this way one can ensure that always the overloaded name is being used outside.

An example of defining an instance of an overloaded operator in a definition module hiding the actual operator name:

```
instance ==
```

The following restrictions apply:

- The type of a concrete function or operator must exactly match the overloaded type scheme after uniform substitution of the type scheme variable by the type as specified in the corresponding type instance declaration.
- A type instance of an overloaded type must be a *flat type*, i.e. a type of the form `T` $a_1 \ldots a_n$ where a_i are type variables which are *all* different.
- It is not allowed to use a type synonym as instance.
- All instances other than the default instance of a given overloaded type must differ from each other (be ununifyable with each other).
- If a default instance is specified the type of the corresponding concrete default function must be identical to the type of the overloaded function or operator.
- If the concrete function or operator is not specified in a definition module, it has to be defined in the corresponding implementation module.
- The start rule cannot have an overloaded type.
- Ambiguously overloading is not permitted.

Type classes

When a function is defined in terms of an overloaded function it can occur that the type system cannot decide which one of the corresponding concrete functions to apply. The new function then becomes overloaded as well. This has as consequence that an additional restriction must be imposed on the use of such a function. This is reflected in the type of the function.

For instance, the function

```
add x y = x + y
```

becomes overloaded as well because any concrete instance can be applied. So, `add` can be applied to arguments of any type, as long as addition (+) is defined on them.
In a *type class definition* one gives a name to a *set of overloaded functions* (this set actually defines a type class record, see above). The definition of the overloaded functions themselves can be directly specified in the type class definition itself but one can also refer to overloaded functions type definitions declared elsewhere. One and the same overloaded function can be a *member* of different type classes. Instances of the overloaded functions are created as described above. There is no hierarchy in type classes.

An example of a definition of a type class:

```
class Arith a
where
   (+) infixl 6 :: a a -> a
   (-) infixl 6 :: a a -> a
```

An equivalent definition:

```
class Arith a
where +; -

overload (+) infixl 6 a :: a a -> a
overload (-) infixl 6 a :: a a -> a
```

In the definition of the type of a function that has become overloaded one can now refer to a type class to impose a restricted context on the instantiation of a type variable of the function. Such a context imposes a condition (predicate) under which type variables is allowed to be instantiated. So one obtains a kind of *bounded polymorphism*. The function can only be applied if for the corresponding concrete type the indicated type classes have been instantiated.

```
/*
Use of a type class to impose a restriction on the instantiation of
type variable: the function add can only be applied on arguments for
which an instance of the class Arith is defined.
*/
add :: a a -> a | Arith a
add x y = x + y
/*
Instance declaration of which type is depending on same type class:
the function sumlist itself can also be defined as an instance for
the overloaded operator + working on arbitrary list for which the
type class Arith is defined on the list elements. sumlist must now be
defined as operator because this is expected from an instance of +.
With this definition + is defined on lists, and list of lists etc.
*/
instance + [a] | Arith a = sumlist

(sumlist) infixl 6 :: [a] [a] -> [a] | Arith a
(sumlist) [x:xs] [y:ys]   =  [x + y:xs + ys]
(sumlist) x y             =  []
```

An equivalent definition:

```
instance + [a] | Arith a
where
   (+) [x:xs] [y:ys]   =  [x + y:xs + ys]
   (+) x y          =  []
```

Cleans type system will infer contexts automatically. It will however *not* express this in terms of type classes but simple summarize the collection of functions on which the overloaded function is depending. This is caused by the fact that it is allowed that one function is defined as a member of several type classes. If a type class is specified as restricted context the type system will check the correctness of the specification (as always a type specification can be more restrictive than is deduced by the compiler).

- For an overloaded function which is exported the type (including the context) has to be defined explicitly by the programmer.
- The type checker will complain if a concrete application cannot be applied due to the fact that certain instances of type classes have not been declared.

```
//    Equality class.

class Eq a
where
   (==) infix 2 :: a a -> Bool

instance ==             = DefaultEq      //    by default, see above
instance ==   Int       = ==^            //    on integers (primitive)

instance ==   [a] | Eq a                 //    on lists
(==) [x:xs] [y:ys]  = x == y && xs == ys
(==) []       []       = True
(==) _      _       = False

instance ==   (a,b) | Eq a & Eq b        //    on tuples of arity two
(==) (x1,x2) (y1,y2) = x1 == y1 && x2 == y2
```

The members of a class consists of a set of functions or operators which logically belong to each other. It is often the case that the effect of some members (*derived members*) can be expressed in others. For instance <> can be seen as synonym for `not (==)`. For software engineering (the fixed relation is made explicit) and efficiency (one does not need to include derived members in the class record) reasons it is good to make this relation explicit. In Clean macro definitions can be used for this purpose.

```
// Classes with derived members.

definition module overloaded

class Eq a
where
   (==) infix 2 :: a a -> Bool

   (<>) infix 2 :: a   a  ->   Bool | Eq a
   (<>) x y :== not (x == y)

class Ord a
where
   (<) infix  2 :: a a    ->   Bool

   (>) infix  2 :: a a    ->   Bool | Ord a
   (>) x y :== y < x
```

```
(<=) infix  2 :: a a   ->    Bool | Ord a
(<=) x y :== not (y<x)

(>=) infix  2 :: a a   ->    Bool | Ord a
(>=) x y :== not (x<y)

min :: a a -> a | Ord a
min x y :== if (x<y) x y

max :: a a -> a | Ord a
max x y :== if (x<y) y x
```

When an overloaded function is exported a type class record has to be constructed as explained in the introduction. However, for efficiency reasons it would be nice to know which instances of the type class are known in the implementation module. When a type class is exported one can explicitly define the minimal *set of instances* which exist for this type class. For any function defined on such a type class one can now deduce that at least these instances are known.

```
// Classes with derived members.

definition module StdOverloaded

overload == a :: a a -> Bool

definition module StdInt

import overloaded

instance == Int

definition module StdReal

import overloaded

instance == Real

definition module example

import StdInt, StdReal

class Eq a
where    ==
instances Int, Real
```

4.1 Extending Interactive Applications Without Re-compilation

The use of type classes in Clean 1.0 makes it possible to extend interactive applications without the need for re-compilation of the part that is extended.

It makes it even possible to write higher order functions that extend (parts of) menus without knowing the names of the actual call-back functions that are present in the definitions.

For the extension of the function editor in the FunSheet project as described in section 3.9 this would mean that the old call-back functions would operate on the old editor state and that a general function could be used that takes any menu definition with call-back functions for the old editor and extends that to a menu definition with call-back functions for the function editor. This would be applicable for the pulldown menus for editing and searching.

```
module ExtStdEditor

Start = genexcbf cbf { e = Ex, r = { s = E, io = IO E}}

//    standard definitions for an editor

::Editor    = E
::IOState x = IO x

::State =   {  s ::Editor,
               io::IOState    Editor}

cbf::State -> State
cbf s = s

// added definitions for a state extension

class XState sup
where
   ToState ::sup -> (State,sup)
   FromState::(State,sup) -> sup

// general definition of a state extension including the extension of
// a state-transition function

instance XState ExState
where
   ToState    sup         = (sup.r,sup)
   FromState (state,sup)  = {sup & r = state}

::Extension = Ex
::ExState = { e::Extension,
              r::State }
```

```
excbf::s -> s | XState s
excbf s = FromState (nstate,s1)
where
   (state,s1) = ToState s
   nstate = cbf state

// higher order function to extend a standard call-back-function to
// the extended state

genexcbf::(State -> State) s -> s | XState s
genexcbf cbf s = FromState (nstate,s1)
where
   (state,s1) = ToState s
   nstate     = cbf state
```

4.2 Object-oriented Programming Techniques

Object-oriented programming can be useful when large software components are to be combined. Important aspects of object-oriented programming such as abstraction, encapsulation and multiple inheritance can be modelled in a functional language.

Modelling Objects

An object-oriented style of programming can be used in a functional programming language by modelling an object by a record which contains the state of the object and the methods that can be applied on the object. The use of existential types can make such a way of modelling quite general allowing functions to operate on lists of objects with different internal states as is shown in the example below.

```
module object

::Object E.x  =  {  state    ::x,
                    get      ::x -> Int,
                    set      ::x Int -> x
                 }
MyObject::Object Void
MyObject = {state = [], get = myget, set = myset}
where
   myget::[Int] -> Int
   myget [i:is] = i
   myget []     = 0

   myset::[Int] Int  -> [Int]
   myset is i = [i:is]

Get::(Object Void) -> Int
Get {state,get} = get state

Set::(Object Void) Int -> Object Void
```

```
Set o=:{state,set} i = {o & state = set state i}

Start = Get (Set (Set MyObject 1) 2)
```

Modelling Multiple Inheritance

Using type classes and conversion functions combined with overloading an expressive power can be obtained which is similar to the expressive power of subtyping. Multiple inheritance as is present in many object-oriented languages can be modelled using these facilities.

The classic example of combinations of points, colour and lines is given below.

```
module point
import StdEnv

Start    =  (moveX {cpp={x=1.0,y=2.4},cpc={c=23}} (1.5,2.5),
             mirrorlX {cplp={x=1.0,y=2.4},
                       cplc={c=23},
                       cpll={f=(0.0,1.0),
                       t=(1.0,2.0)}})

//    layer 1

::Point    =   {   x::Real,
                   y::Real  }
::Color    =   {   c::Int   }
::Line     =   {   f::(Real,Real),
                   t::(Real,Real)    }
::Dist   :==   (Real,Real)

move::Point Dist -> Point
move {x,y} (dx,dy) = {x=x+dx,y=y+dy}

moveX::spoint Dist -> spoint | XPoint spoint
moveX rec dist = cPoint (np,nrec)
where
   (p,nrec)    = sPoint rec
   np          = move p dist

invcol::Color -> Color
invcol {c} = {c=0-c}

mirrorl::Line -> Line
mirrorl {f=(xf,yf),t=(xt,yt)} = {f=(xf,yt),t=(xt,yf)}

mirrorlX rec = cLine (np,nrec)
where
   (p,nrec)    = sLine rec
   np          = mirrorl p
```

```
// Point
class XPoint su
where
   sPoint :: su -> (Point,su)
   cPoint :: (Point,su) -> su

instance sPoint Point = PtoP_P
instance cPoint Point = P_PtoP
PtoP_P::Point -> (Point,Point)
PtoP_P x = (x,x)
P_PtoP::(Point,Point) -> Point
P_PtoP (p,v) = p

// Color
class XColor su
where
   sColor ::su -> (Color,su)
   cColor ::(Color,su) -> su

instance sColor (Color) = CtoC_C
instance cColor (Color) = C_CtoC
CtoC_C::Color -> (Color,Color)
CtoC_C x = (x,x)
C_CtoC::(Color,Color) -> Color
C_CtoC (p,v) = p

// Line
class XLine su
where
   sLine ::su -> (Line,su)
   cLine ::(Line,su) -> su

instance sLine Line = LtoL_L
instance cLine Line = L_LtoL
LtoL_L::Line -> (Line,Line)
LtoL_L x = (x,x)
L_LtoL::(Line,Line) -> Line
L_LtoL (p,v) = p

// layer2: combinations of the records of layer1

::ColorPoint = { cpc::Color,cpp::Point }

class XColorPoint su
where
   sColorPoint ::su -> (ColorPoint,su)
   cColorPoint ::(ColorPoint,su) -> su
```

```
instance sColorPoint ColorPoint = CPtoCP_CP
instance cColorPoint ColorPoint = CP_CPtoCP
CPtoCP_CP::ColorPoint -> (ColorPoint,ColorPoint)
CPtoCP_CP x = (x,x)
CP_CPtoCP::(ColorPoint,ColorPoint) -> ColorPoint
CP_CPtoCP (p,v) = p

instance sPoint ColorPoint = CPtoP_CP
instance cPoint ColorPoint = P_CPtoCP
CPtoP_CP::ColorPoint -> (Point,ColorPoint)
CPtoP_CP rec=:{cpc,cpp} = (cpp,rec)
P_CPtoCP::(Point,ColorPoint) -> ColorPoint
P_CPtoCP (p,rec) = { rec & cpp=p }

instance sColor ColorPoint = CPtoC_CP
instance cColor ColorPoint = C_CPtoCP
CPtoC_CP::ColorPoint -> (Color,ColorPoint)
CPtoC_CP rec=:{cpc,cpp} = (cpc,rec)
C_CPtoCP::(Color,ColorPoint) -> ColorPoint
C_CPtoCP (c,rec) = { rec & cpc=c }

// layer3: combinations of the records of layer1 and 2

::ColorPointLine = { cplc::Color,cplp::Point,cpll::Line }

class XColorPointLine su
where
   sColorPointLine ::su -> (ColorPointLine,su)
   cColorPointLine ::(ColorPointLine,su) -> su

instance sLine ColorPointLine = CPLtoL_CPL
instance cLine ColorPointLine = L_CPLtoCPL
CPLtoL_CPL::ColorPointLine -> (Line,ColorPointLine)
CPLtoL_CPL rec=:{cpll} = (cpll,rec)
L_CPLtoCPL::(Line,ColorPointLine) -> ColorPointLine
L_CPLtoCPL (l,rec) = { rec & cpll=l }
```

Introducing more Locality in I/O Definitions

The use of existential types with the power of overloading as it is defined in Clean 1.0 makes it possible to define a new version of the I/O library which has a facility to keep a local state in a part of the graphical user interface.

An example of the application of call-back functions on objects with local state in a time-sliced manner is given below.

```
module locstate
import StdEnv

::State E.a ls = {  ps::a,
```

```
                    ms::(a,ls) -> (a,ls) }

Apply::((State Void ls),ls) -> ((State Void ls),ls)
Apply (rec=:{ps,ms},ls) = (nrec,nls)
where
   (na,nls) = ms (ps,ls)
   nrec     = { rec & ps = na }

TimeSlice::([State Void ls],ls) -> ([State Void ls],ls)
TimeSlice ([rec:rest],state) = ([nrec : nrest],endstate)
where
   (nrest,endstate)    = TimeSlice (rest,nstate)
   (nrec,nstate)       = Apply (rec,state)
TimeSlice final = final

NTimes::!([State Void ls],ls) Int -> ([State Void ls],ls)
NTimes state 0 = state
NTimes state n = NTimes (TimeSlice state) (n-1)

Start = NTimes ([MyRec1,MyRec2],0) 20

::LocalState :== Int

MyRec1 = { ps = 1.0, ms = mycbf }
where
   mycbf::(Real,Int) -> (Real,Int)
   mycbf (a,b) = (a+1.5,b+1)

MyRec2 = { ps = 'a', ms = mycbf }
where
   mycbf::(Char,Int) -> (Char,Int)
   mycbf (a,b) = (toChar ((toInt a) + 1),b+1)
```

4.3 Monads

Monads form a programming style which can be useful when higher order functions are combined in a particular pattern.

An example of a state monad is given below. It is important to note that single-threadedness of the definitions is not a supposition but it is derived by Clean's uniqueness type inference scheme.

```
module monad  //state monad

::St s a :== s -> (a,s)

return::u:a -> u:(St .s u:a)
return x = \s -> (x,s)

(`bind`) infix 0:: u:(St .s .a) u:(a -> .(St .s .b)) -> u:(St .s .b)
```

```
(`bind`) f_sta a_fstb = stb
where
   stb st = a_fstb a nst
   where
      (a,nst) =  f_sta st

Start::[.Char]
Start = result
where
   (result,s) =  (  f `bind` (\a ->
                    g `bind` (\b ->
                    return [a,b] ))) S

::S = S    // concrete state with one possible value: S

f::.a -> (.Char,.a)
f s = ('f',s)

g::.a -> (.Char,.a)
g s = ('g',s)
```

4.4 Programming Environment Support

The fact that functions have no side-effects seems to lead to a much more refined module structure than is the case in imperative programming. Therefore more demands are put on the programming environment to support tools for modular programming. A few examples of such tools to improve modular software productivity are given below.

Adding structure to the project defining layers in which definition modules can depend on each other might also be very helpful.

It can be very important to know quickly which modules use a certain function. Without such a facility easily many almost equivalent function definitions emerge since 'to be safe' new functions are made instead of changing or generalising existing functions.

It can be very useful to have a special facility to change a definition module of a library in a 'benign' way, i.e. just changing the comments or the layout or extending it with one or more new functions. In the latter case all modules that depend on that the definition module do not have to be recompiled completely but just checks on name clashes and possibly the generation of new code labels have to be performed. This may save a lot of development time.

Strictness could be exported automatically. A warning could be given for specified strictness that cannot be inferred by the system.

Facilities should be available to perform editing and formatting operations on groups of modules that form a project. Changing a module name or a function name all over a project or printing all (definition) modules that are used in a project are important examples of the use of such tools.

5 Conclusions

- The functional spreadsheet design served as an interesting *test case* for techniques for developing efficient medium sized functional programs.
- In the FunSheet project 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. Considering the relatively small scale of the project its *software productivity* was quite high. Considering the functionality 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* is satisfactory.
- 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 incorporates most of them.
- Several *well-known techniques* for writing larger programs can be *modelled* in a relatively straightforward manner in a functional language (Clean 1.0). This makes these techniques also applicable for functional languages.
- The *support* for medium sized functional programs is still low both in programming languages as well as in programming environments. There are ample opportunities for improvement.

Exercises

The first exercise is an introductory exercise to get acquainted with Clean and with the I/O library. The second exercise is a larger exercise with more demands with respect to re-use of existing software.

1. Extend using the I/O library a given pocket-calculator program with a number of extra buttons for extra functionality.
2. Use the described text editor, help facility and I/O library to create a HTML editor with support for defining hypertext links and hypertext layout. Preferably, the sources of the existing modules remain unchanged and do not even have to be re-compiled.

References

Achten, P.M., and Plasmeijer, M.J. 1994. A Framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language Clean. In Bakker E. ed. *Proc. Computing Science in the Netherlands, CSN'94*. Jaarbeurs Utrecht. The Netherlands. Stichting Mathematisch Centrum, Amsterdam. pp. 30-41.

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.

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.

Harper, R., D. MacQueen and R. Milner (1986). 'Standard ML'. Edinburgh University, Internal report ECS-LFCS-86-2.

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.

Hoon, W.A.C.A.J. de, Rutten, L.M.W.J., and Eekelen, M.C.J.D. van 1995. Implementing a Functional Spreadsheet in Clean. *Journal of Functional Programming*, Vol **3**, 1995, to appear.

Hudak, P. , S. Peyton Jones, Ph. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, W. Partain and J. Peterson (1992). 'Report on the programming language Haskell'. *ACM SigPlan notices,* **27**, 5, pp. 1-164.

Jones, M.P. (1993). *Gofer - Gofer 2.21 release notes*. Yale University.

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.

Turner, D.A. (1985). 'Miranda: a non-strict functional language with polymorphic types'. In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, ed. J.P. Jouannaud, Nancy, France. LNCS **201**, Springer Verlag, 1-16.

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.