Kiel Interactive Evaluation Laboratory

Rudolf Berghammer, Markus Tiedt

Bericht Nr. 9909
November 1999

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel

Contents

1 Introduction

Functional programming has first been made known to a wider audience by the well-known Turing Award lecture [1] of J. Backus. Since this seminal paper the functional approach to programming has been gaining increasing appeal, especially in the last 10 years. Meanwhile it is also very popular in teaching the basic concepts of programming at universities. Among other things, this is surely due to the fact that for many problems functional programs are simpler, faster to design, and, therefore, less susceptible for errors than their imperative counterparts. A further advantage is that they are very close to their semantics and this implies that it is generally more easy to reason about functional programs than imperative ones.

A very prominent way to define a formal semantics of functional programs (which is especially qualified for beginners) is an operational approach via computation sequences which describe the individual steps of program executions. More concrete, the elements of a computation sequence are expressions and one after another produced by substitutions (also called unfoldings or expansions) and simplifications. Substitutions replace some calls of a routine by its body, where in each case within the latter the formal parameters are substituted by the arguments. Which calls are to be replaced is determined by a computation rule (also called parameter-passing mechanism), e.g., leftmost-innermost substitution (call-by-value), leftmost-outermost substitution (call-by-name), or full substitution. Simplifications execute operations of the pre-defined data structures and conditionals. For more details see [6, 13].

Operational semantics defined in the just sketched manner executes a functional program by applying a sequence of meaning-preserving program transformations and can easily be implemented. The benefits of using such a transformation system in teaching computer science are manifold, in particular if there exists a good graphical representation of expressions and the user has full interactive control over the entire transformation process via a modern interface. For example, the system can be used to visualize program execution and the effect of the various computation rules. Furthermore, it enables debugging of programs by step-wise computations to detect weak points and logical errors. And, finally, moving step by step through a computation using a system can elucidate the decisive ideas behind the algorithm a functional program implements.

The advantages of interactive execution of functional programs for teaching purposes have been also recognized elsewhere and in the meantime there exist some computer systems for that purpose. For example, in [3] the system MIRACALC is described which allows the stepwise execution of Mirinda programs. Another such system is π-RED, developed at Kiel University. It supports the interactive execution of functional programs written in the Kiel reduction language KiR, a sugard version of an applied type-free λ-calculus with strict semantics; see e.g., [4, 5]. Finally, the Calculation Sheet Machine of the textbook [2] should be mentioned which has been implemented at the Institute for Software Engineering and Programming Languages of the Medical University Lübeck.

For many Lisp systems there exist debuggers. But none of the well-known implementations of modern functional programming languages (like the various Haskell, Miranda, or SML systems) supports stepwise execution and/or includes a debugging mechanism. The only investigations and implementations we are aware are debuggers for lazy functional languages using the so-called algorithmic debugging method and developed at Linköping University, Sweden, resp. the University of Melbourne, Australia. Some references are [9, 10, 8] for example.

In this report we describe an interactively controlled computer system for the execution of first-order functional programs written in a simple subset of Standard ML. It is called KIEL, which is an acronym for "Kiel Interactive Evaluation Laboratory", and has especially been designed for teaching undergraduate students of computer science. The rest of the report is organized as follows. In Section 2 we describe the subset of ML the KIEL system supports. Section 3 is primarely intended as a user's guide for KIEL. We give a short overview of the system, describe its graphical user interface, and show how to arrange it with the help of configuration files. An example for the use of the system is presented in Section 4. Concluding remarks can be found in Section 5.

2 The Programming Language

As already mentioned, the KIEL system supports a simple subset of the functional programming language ML. We assume the reader to be familiar with that part of Standard ML which is, e.g., described in the Chapters 1 through 5 of the textbook [12]. This part is closely related to the KIEL-subset of ML, called MLKIEL in the rest of this report. In this section we present MLKIEL. However, we will not give a formal description. Syntax diagrams for a formal definition of its syntax can be found in the appendix of [11] and with the exception of parameter passing (the free choice of which is just one of the main features of KIEL) the meaning of the remaining language constructs is as in Standard ML.

2.1 Datatypes

There are five basic types in MLKIEL, viz. the singleton type `unit`, the type `bool` for the truth values, the type `int` for the integers, the type `real` for the real numbers, and the type `string` for the strings.

The type `unit` has only one constant which is denoted as `()`. Truth values may have two values which are denoted as `true` resp. `false`. The constants of the type `int` are represented as in Standard ML, i.e., as strings of one and more digits in the non-negative case, with the additional sign "~" in front of it otherwise. The present version of the KIEL system allows the integers to range from -2^{30} to $2^{30} - 1$. Also the constants of the type `real` are represented as in Standard ML. I.e., they start with an optional "~" followed by an integer representation and then either a decimal point and an integer representation or the letter `E`, an optional "~", and an integer representation. A constant of the type `string`, finally is a quoted character sequence. In contrast with ML, however, special characters like "newline" or the tab character are not allowed. The maximal length of a string can be defined by means of a configuration file. See Section 3.3 for details.

The operations on the basic types are similar to those of Standard ML. They are summarized in the following. Note that each basic type m of MLKIEL is an equality type in the sense of ML which means that besides the listed operations also an equality test `=` : m `*` m `-> bool` and an inequality test `<>` : m `*` m `-> bool` are defined.

Again in contrast with Standard ML, in the KIEL system sequential conjunction and disjunction are interpreted as operations on truth values and not as abbreviations of specific conditionals. This leads to the following three operations on truth values:

`not : bool -> bool`	Negation
`andalso : bool * bool -> bool`	Sequential conjunction
`orelse : bool * bool -> bool`	Sequential disjunction

The operations on the integers, the real numbers, and the strings are in each case a subset of the respective operations of Standard ML. For the integers MLKIEL provides the following operations:

`+, -, * : int * int -> int`	Arithmetic
`div, mod : int * int -> int`	Integer division and remainder
`min, max : int * int -> int`	Minimum and maximum
`abs : int -> int`	Absolute value
`<, <=, >, >= : int * int -> bool`	Comparison operations

The operations of MlKiel for the real numbers are:

`+, -, *, / : real * real -> real`	Arithmetic
`abs : real -> real`	Absolute value
`sqrt : real -> real`	Square root
`sin, cos : real -> real`	Sinus and cosinus
`exp, ln : real -> real`	Exponential function and natural logarithm
`real : int -> real`	Conversion into reals
`truncate : real -> int`	Rounding towards zero
`<, <=, >, >= : real * real -> bool`	Comparison operations

And, finally, on strings one can use the following operations of MlKiel:

`^ : string * string -> string`	Concatenation
`size : string -> int`	Length
`chr : int -> string`	Inverse of ASCII-code
`ord : string -> int`	ASCII-code of the first character
`substring : string * int * int -> string`	Substring
`<, <=, >, >= : string * string -> bool`	Lexicographic comparison operations

In functional programming lists constitute an extremely valuable datatype and, therefore, Standard ML provides a polymorphic type for lists together with an abundant stock of operations. MlKiel does not include polymorphism. However, we have decided to implement for each of the above basic types m a type m `list` for lists whose elements are from m. Constants are represented as in ML. This means that the elements of a list are separated by commas and surround with square brackets. The empty list is represented by either the name `nil` or as pair `[]` of brackets. As operations on lists, MlKiel supports the following ones:

`::` : m `*` m `list -> ` m `list`	Append from left
`hd :` m `list ->` m	First element (head)
`tl :` m `list ->` m `list`	Removal of first element (tail)
`@ :` m `list *` m `list ->` m `list`	Concatenation
`length :` m `list -> int`	Length
`null :` m `list -> bool`	Empty test
`rev :` m `list ->` m `list`	Reversion

As in the case of strings, the maximal length of a list can be defined by means of a configuration file. See again Section 3.3.

2.2 Programs

A program of MlKiel consists of a sequence of declarations, where a declaration either defines a single routine or a system of (generally mutually recursive) routines and is terminated with a semicolon. The form of routine declarations is as in Standard ML with the exception that – due to methodical reasons – MlKiel requires full typing. The type m of a formal parameter x

is given within the parameter list in the form $x : m$ and the type of the result is given after the parameter list, again separated by a colon. Hence, a single routine F with formal parameters x_i of type m_i $(1 \leq i \leq k)$, result type m, and body E is decleared as follows:

$$\texttt{fun}\ F\ (x_1 : m_1, \ldots, x_k : m_k) : m = E\ ;$$

Note that $k = 0$ is possible, i.e., the parameter list of a routine may be empty. The body of a routine may be constructed with the help of

1. the parameters,
2. the constants and operations of the basic datatypes introduced in Section 2.1,
3. the already declared routines,
4. the conditional `if` B `then` A_1 `else` A_2,
5. recursive calls.

Here applications of basic operations are written as in Standard ML with the exception that – again for methodical reasons – MLKIEL uses the "classical" mathematical notation $f(A)$ instead of $f\,A$ or $(F\,A)$. Also applications of already defined routines and recursive calls have to be written in the classical style. Note that MLKIEL does not contain neither higher-order functions nor patterns in routine declarations, local declarations, and type declarations.

As a small example, here is a MLKIEL program which implements Hoare's quicksort algorithm for sorting a list of integers. It uses three auxiliary routines `le`, `eq`, and `gr` which filter out from a list `s` those elements which are less than `n`, equal to `n`, resp. greater than `n`.

```
fun le (s : int list, n : int) : int list =
  if null(s) then nil
             else if hd(s) < n then hd(s) :: le(tl(s),n)
                               else le(tl(s),n);
fun eq (s : int list, n : int) : int list =
  if null(s) then nil
             else if hd(s) = n then hd(s) :: eq(tl(s),n)
                               else eq(tl(s),n);
fun gr (s : int list, n : int) : int list =
  if null(s) then nil
             else if hd(s) > n then hd(s) :: gr(tl(s),n)
                               else gr(tl(s),n);
fun sort (s : int list) : int list =
  if null(s) then s
             else sort(le(s,hd(s))) @ eq(s,hd(s)) @ sort(gr(s,hd(s)));
```

The syntax for mutual recursion in MLKIEL is the same as in Standard ML. I.e., the keyword `fun` is used only at the beginning and between the routines instead of a semicolon the keyword `and` is written. We give also a small example. The following small MLKIEL program (taken from [12]) can be used to produce lists consisting of alternate elements of the list `s`:

```
fun odd (s : int list) : int list =
  if null(s) then nil
             else hd(s) :: even(tl(s))
and even (s : int list) : int list =
  if null(s) then nil
             else odd(tl(s));
```

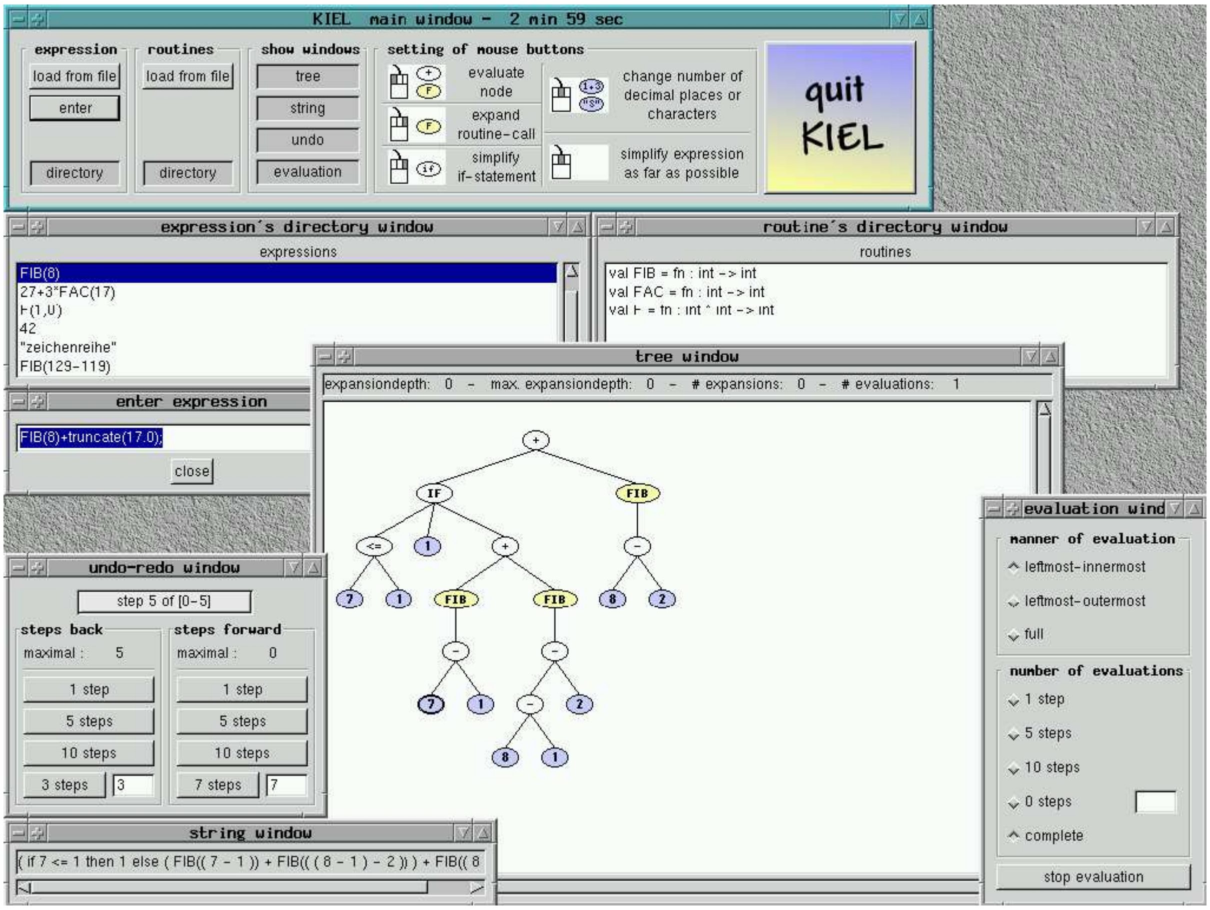
Applying `odd`, one takes the first element of a list and the alternate elements after that (i.e., the third, fifth and so on); using `even`, one starts the filter process with the second element.

3 Structure and Use of KIEL

In this section we start with a short overview of KIEL. Then we describe the system's graphical user interface. The last part of the section deals with the configuration of the system by means of two configuration files.

3.1 General Structure

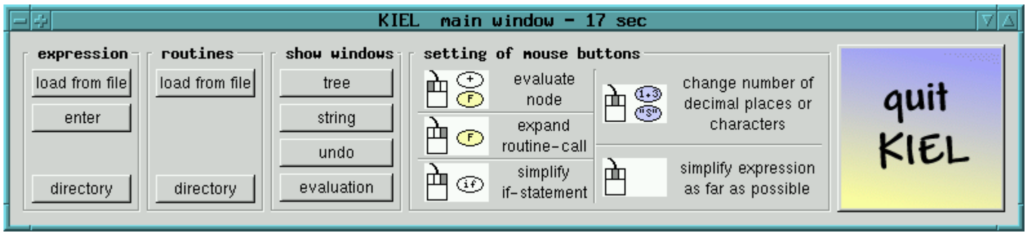
The system KIEL is written in the C programming language and uses the C-libraries GTK, GDK, and GLib for its graphical user interface. Although the detailed appearance of its windows depends on resources defined in a configuration file, when working with it the screen typically looks as in the following picture:



From the user's point of view, roughly spoken KIEL consists of two components. The first component is the "workspace". It holds MLKIEL programs and MLKIEL expressions. A program is assumed to be contained in a file with extension "`.prg`" and may then loaded into the system via the window "load program file". Expressions can also be loaded from a file with extension "`.exp`" using the window "load expression file". But the usual way the user enters single expressions directly via the window "enter expression". The system's second component is the evaluation unit resp. execution unit. Here an expression of the workspace is evaluated based on the pre-defined datatypes and the routines contained in the workspace. The main window for evaluation is the tree window. We have chosen this name because expressions are graphically depicted as trees. Evaluation of expressions can be done step by step on the tree window but also semi-automatically or full-automatically, where in the latter cases the user can choose a specific computation rule, control the number of transformation steps and so on using further windows. Details of the user interface and how to evaluate an expression are presented in Section 3.2.

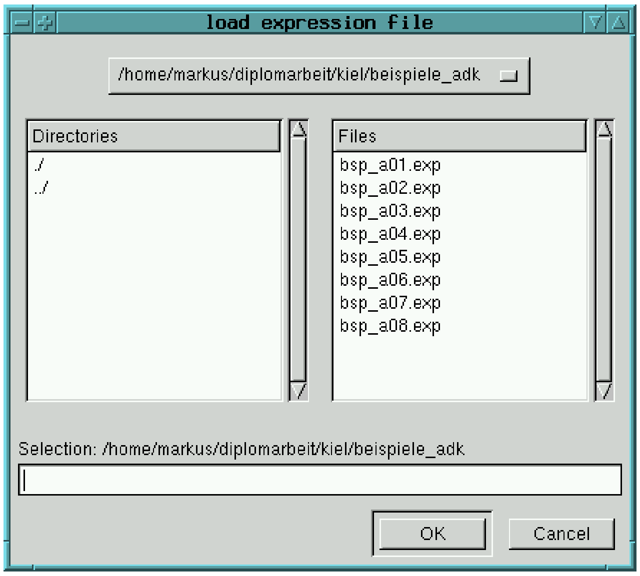
3.2 User Interface

We start the description of the user interface of KIEL with the system's main window which looks as follows:

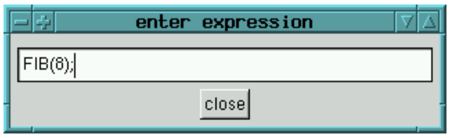


As can be seen from this picture, conceptually the main window is divided into three parts. The left part consists of three button columns which deal with the input of expressions, the input of programs, and the windows used for the evaluation of expressions. In the middle of the window there are five pictures which explain the various actions invoked by the three mouse buttons. The right part of the main window is a single big button for leaving KIEL.

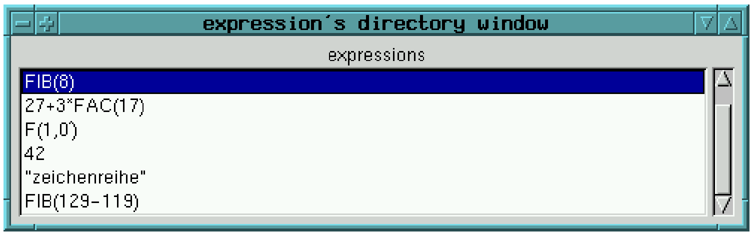
The leftmost button column of the main window consists of three buttons. Pressing its top button "load from file" opens a window for loading an expression file into the system. Typically this window looks as follows:



The left scroll list of the window "load expression file" shows the directories which are contained in the current directory and the expression files of the current directory are shown in the window's right scroll list. Selection of an expression file is possible using the mouse pointer and after that the loading process may be initiated by pressing the button "OK". But as already mentioned, the usual way is to enter single expressions. For this purpose, the following window "enter expression" is used which is opened by the button "**enter**" of the leftmost button column:

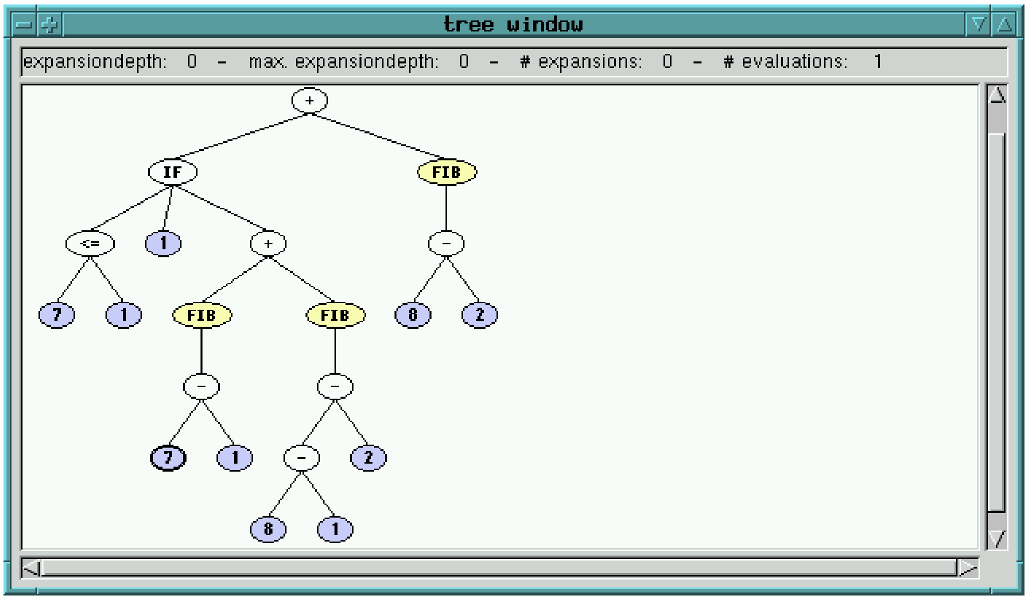


The button "**directory**" of the leftmost button column of the main window, finally opens a window which shows the workspace's actual expressions. Here is an example for its appearance:



Loading a program file into KIEL is done in the same way than loading an expression file. The corresponding button is "**load from file**" at the top of the second button column of the main window. Also the remaining button "**directory**" of this column has the same meaning as the corresponding button for expressions. We renounce the pictures of the two windows opened by these buttons.

KIEL was designed for the interactively controlled execution of functional programs. Pressing the button "**tree**" of the main window, the tree window opens which allows performing this. One starts with an expression of the workspace which is transfered into the tree window by selecting it in the expression's directory window. The following picture shows a typical appearance of the tree window; the depicted tree represents an expression obtained during the execution of a routine `FIB` which implements the well-known Fibonacci recursion.

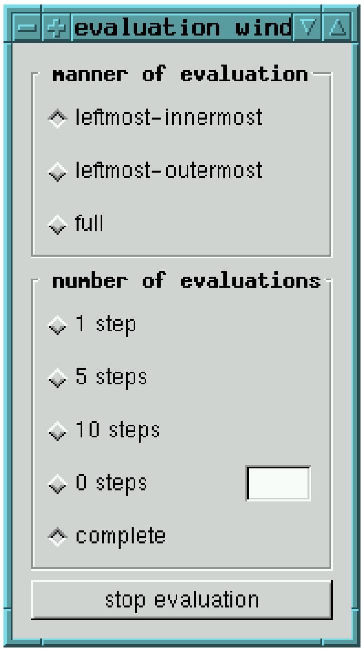


Generally a vertex of a tree depicts a constant or an operation of the pre-defined datatypes, a user-declared routine, or an error. The next four pictures show for each of these classes an example. Perhaps the reader will notice the four different shades of grey; on a colour screen they correspond to four different colours (which can be determined via a configuration file).

Normally the size of the vertices of a tree is fixed and, therefore, big numbers, strings, and sequences are only partly shown. But also the complete representation of such an object is possible. The user has only to select the corresponding vertex and to enlarge it by means of the middle mouse button. See the corresponding picture in the middle part of the main window.

As the screen dump of Section 3.1 shows, the expression of the tree window is also represented as a string in the string window. This window is opened if the button "string" of the third button column of the main window is pressed.

If one wants to evaluate an expression full-automatically or semi-automatically, then this is possible by first selecting the corresponding vertex on the tree window with the mouse pointer and then pressing the left mouse button. The manner of evaluation and – in the semi-automatical case – the number of steps to be done can be entered by a specific evaluation window which is opened by pressing the button "evaluation" of the third button column of the main window. As the following picture shows, the present version of the KIEL system allows the choice between exact the three computation rules mentioned in the introduction, viz. leftmost-innermost substitution, leftmost-outermost substitution, and full substitution.



Having described full-automatic and semi-automatic execution, now we deal with the stepwise evaluation of expressions. As we have already mentioned in the introduction, it is performed by means of three actions, viz.

1. the substitution of routine calls,
2. the execution of operations of the pre-defined datatypes,
3. the simplification of conditionals.

How these actions can be invoked is explained by the three pictures in the left picture column of the middle part of the main window of KIEL (see the screen dump of Section 3.1): The substitution of a routine call is possible by selecting the corresponding vertex of the tree and then pressing the right mouse button. An execution of an operation of the pre-defined datatypes is the same as the evaluation of an expression and has already been described. Finally a simplification of a conditional is done like a substitution of a routine call. I.e., the user has to select the corresponding vertex of the tree and afterwards to press the right mouse button.

The screen dump of Section 3.1 shows in the right picture column a further picture which explains how to simplify an expression as far as possible: One has only to point with the mouse at a position of the tree window which is not part of a vertex of the depicted tree and afterwards to press the middle mouse button. Using this in combination with substitution of routine calls, it is very easy to simulate the operational semantics definition via computation sequences as described in the introduction of this report.

Finally it should be mentioned that KIEL also allows to undo or redo steps of an execution, i.e., to "walk" through computation sequences. The corresponding window looks as follows:

This window is opened via the button "undo" of the main window.

3.3 Configuration Files

KIEL is in a wide range a user-configurable system. The positions and sizes of several windows, their automatic opening when starting the system, sizes and colours of buttons and vertices, fonts and much more can be chosen by defining resources in two configuration files. At startup time the system looks for these files in the user's home directory. If no configuration file can be found, the system uses default values for the various resources.

A detailed description of the resources supported by KIEL can be found in [11]. In the following we sketch only some of them.

The name of the first configuration file is `.kiel.rc`. Its purpose is to define the positions and sizes of the system's windows and some other layout constants like the maximal number of shown digits after the decimal point of a real number or the maximal number of shown characters of a string. As an example we assume the file `.kiel.rc` to contain the following lines:

```
#set POSITION_BAUMFENSTER                      360, 187
#set BREITE_DRAWING_AREA_BAUM                       750
#set HOEHE_DRAWING_AREA_BAUM                        630
#set BUTTON_AUSDRUCKBAUM_FENSTER_STATUS            TRUE
#set BUTTON_LI_STATUS                              TRUE
```

Then the left-upper corner of the tree window is at pixel position $(360, 187)$, its breadth is 750 pixel, its hight is 630 pixel, the window is automatically opened when KIEL is started, and the computation rule used within full-automatic or semi-automatic executions is leftmost-innermost substitution.

The second configuration file `.kiel.rc.gtk` is used to define fonts and colours. We present also a small example:

```
style "baum"="fg_farben" { font="9x15bold" }
style "fg_farben" { fg[NORMAL]=     {0.9, 0.9, 1.0}
                    fg[ACTIVE]=     {1.0, 1.0, 0.7}
                    fg[PRELIGHT]=   {1.0, 0.5, 0.5}
                    fg[SELECTED]=   {0.0, 0.0, 0.0}
                    fg[INSENSITIVE]={0.0, 0.0, 0.0} }
```

These lines define the font used in the tree window to be `9x15bold` and the colour of a tree vertex to be white if it is a vertex for a pre-defined operation or a conditional, yellow if it is a vertex for a routine, blue if it represents a computed defined value, and red if it stands for an error.

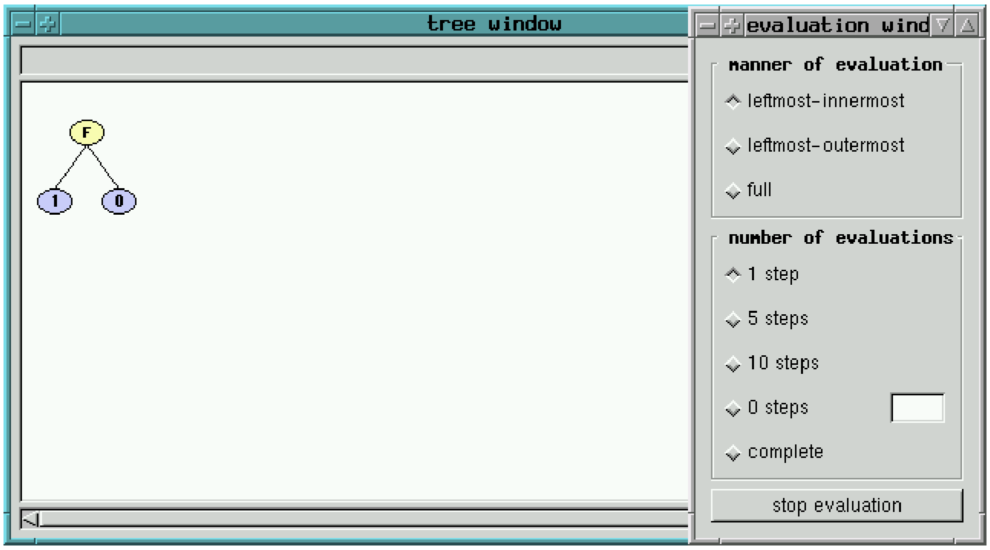
4 An Example

In this section we will deal with a very well-known example of a functional program to demonstrate how KIEL may be used in undergraduate teaching at universities. We consider the following nested-recursive routine on the integers which goes back to Morris (see [7]):

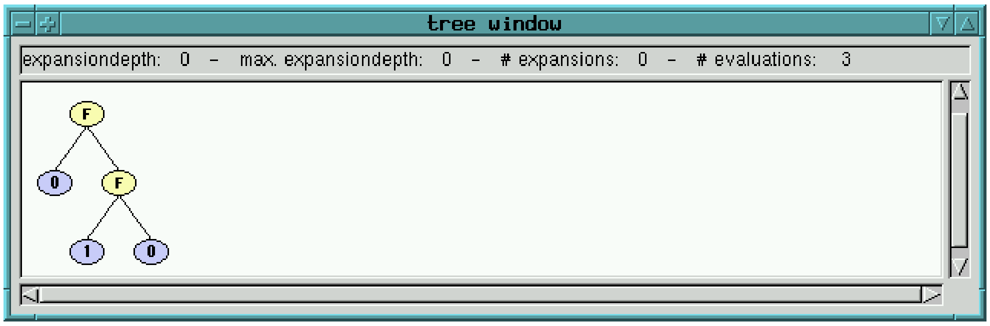
```
fun F (x : int, y : int) : int =
  if x <= 0 then 1
           else F(x-1,F(x-y,y));
```

It is well-known that for this routine different computation rules may lead to different results. E.g., the call `F(1,0)` leads to an infinite computation (i.e., yields "undefined") if leftmost-innermost substitution is applied. However, it evaluates to `1` if the substitution strategy is leftmost-outermost. In the following we show how KIEL can be used to demonstrate this difference.

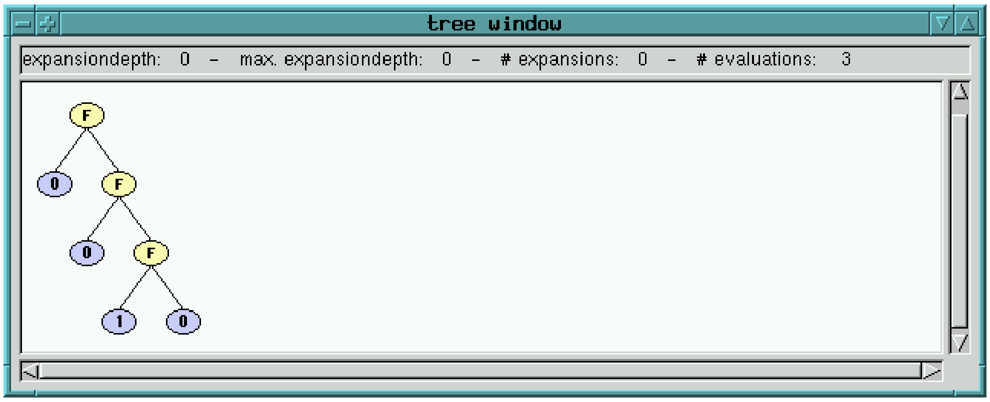
We start with the expression `F(1,0)` we want to evaluate. Furthermore, we choose leftmost-innermost as computation rule and evaluation depth one since execution should be done step by step. The following picture shows the term window and evaluation window of KIEL at the beginning of the execution:



The first two steps of the execution consist in a substitution of the call F(1,0) and after that a simplification of the resulting expression as far as possible. They lead to an expression which is depicted on the tree window of KIEL as follows:



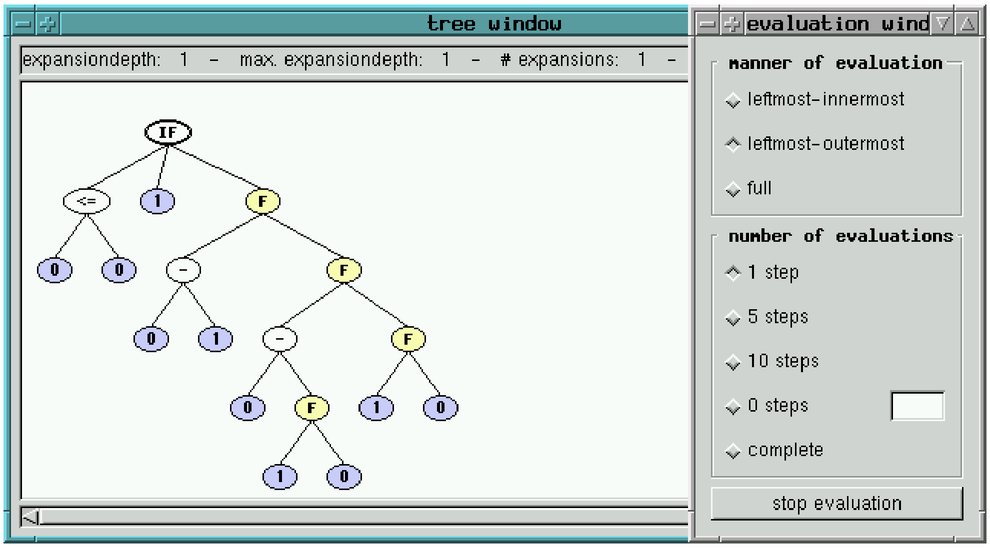
Generally, leftmost-innermost substitution means that we have to substitute the leftmost call of a routine with all arguments free of routine calls. In the present case, hence, we have to select that vertex with label F the successors of which are labeled with 1 and 0. A substitution followed by a simplification as far as possible yields the tree window shown in the following picture:



Already after these few steps it should be clear for a student how leftmost-innermost substitution works and that in the case of the expression F(1,0) its application leads to an infinite sequence of expressions, i.e., to “undefined” as result.

Now we evaluate the expression F(1,0) using leftmost-outermost substitution as computation rule. To this end we have to go back to the original tree window using, for example, the undo-mechanism of KIEL as demonstrated in the following picture.

Since we have only one occurrence of F within the expression on the screen, the first substitution-simplify step of the execution is the same as in the case of leftmost-innermost strategy. But then the situation changes since leftmost-outermost evaluation generally means that we have to substitute always the leftmost call of a routine which does not occur in any argument of another routine call. Looking at the second screen dump of this section, in our concrete case we have to select the root of the tree representing the expression F(0,F(1,0)). The result of the substitution is shown in the following picture:

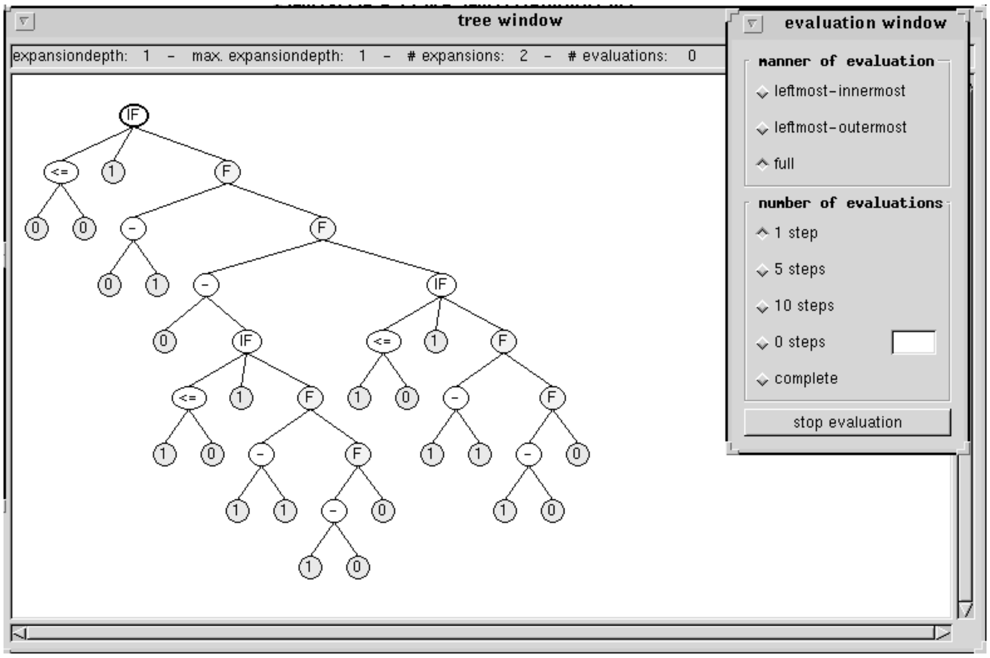


Due to the non-strict semantics of the conditional, now simplification as far as possible yields the value 1 and this result is also computed by KIEL and presented as follows:

A special feature of KIEL is that it shows during a full-automatic evaluation how many substitutions (unfoldings, expansions) of routine calls and evaluations of pre-defined operations resp. conditionals have been executed. This helps students to judge the efficiency of computation rules. It is more or less intuition that the leftmost-innermost rule is more efficient than the leftmost-outermost rule since it avoids many redundant evaluations of arguments. As pointed out by Morris in [7], however, this is not always the case and indeed its routine F is a counter-example. We conclude this section with a table showing some tests performed with KIEL.

	LI-Subst.	LI-Eval.	LO-Subst.	LO-Eval.
F(5,1)	63	125	6	21
F(5,3)	37	73	6	21
F(5,5)	33	65	6	21
F(10,1)	2047	4093	11	66
F(10,5)	1057	2113	11	66
F(10,10)	1025	2049	11	66

We have also performed some tests with the third computation rule the KIEL system supports, i.e., with full substitution. As an example, the following picture shows the tree representation of the expression which results from `F(0,F(1,0))` after a full substitution step.



Using the full substitution strategy, the expressions of the computation sequences became so huge that, for example, on a modern SUN ULTRA workstation a complete evaluation of the expression `F(3,0)` takes more than 20 minutes. In most cases even a complete evaluation with the KIEL system was impossible.

5 Concluding Remarks

In this report we have described the computer system KIEL for the interactively controlled execution of functional programs written in MlKiel, a simple subset of Standard ML. The system is primarely intended as a teaching tool for functional programming at the undergraduate level but can also be applied to debug ML programs if these are reducible to MlKiel. It can be installed on all UNIX-based systems and is available free of charge by FTP from host ftp.informatik.uni-kiel.de, where it is located in the directory pub/kiel/progdev/KIEL. Additional information is published on the WWW page http://www.informatik.uni-kiel.de/~progsys/kiel.html.

In the last months a lot of case studies have been performed with KIEL. It turns out that the system in its present version is a good tool for beginners in functional programming. Because of its restricted programming language MlKiel, however, it can not be used to teach advanced concepts of this field like datatype declarations and the definition of routines via pattern matching, local declarations, and anonymous functions (λ-abstractions). To extend the scope of the KIEL system, it is planned to incorporate such concepts in future versions.

References

[1] Backus J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21, 613-641 (1978)

[2] Bauer F.L., Goos G.: Informatik – Eine einführende Übersicht. Erster Teil, vierte Auflage, Springer (1991)

[3] Goldson D.: A symbolic calculator for non-strict functional programs. The Computer Journal 37, 177-187 (1994)

[4] Kluge W.E.: A users guide for the reduction system π-RED. Bericht Nr. 9419, Inst. für Informatik und Praktische Mathematik, Universität Kiel (1994)

[5] Kluge W.E., Rathsack C., Scholz S.-B.: Using π-RED as a teaching tool for functional programming and program execution. In: Hartel P.H., Plasmeijer M.J. (eds.): Proc. FPLE '95, LNCS 1022, Springer, 231-250 (1995)

[6] Manna Z.: Mathematical theory of computation. McGraw-Hill (1974)

[7] Morris J.H.: Lambda-calculus models of programming. Ph.D. thesis, Mass. Inst. of Technology (1968)

[8] Naish L.: Declarative debugging of lazy functional languages. Research Report 92/6, Dept. of Computer Science, University of Melbourne (1992)

[9] Nilsson H., Fritzson P.: Algorithmic debugging for lazy functional languages. Journal of Functional Programming 4, 337-370 (1994)

[10] Nilsson H.: Declarative debugging for lazy functional languages. Ph.D. thesis, Dept. of Computer and Information Science, Linköping University (1998)

[11] Tiedt M.: Kiel interactive evaluation laboratory: An environment for the visualization of term replacement semantics (in German). Master Thesis, Inst. für Informatik und Praktische Mathematik, Universität Kiel (1999)

[12] Ullman J.D.: Elements of ML programming. Prentice Hall (1994)

[13] Vuillemin J.: Correct and optimal implementation of recursion in a simple programming language. Journal of Computer and System Sciences 9, 332-354 (1974)