KIEL – A Computer System for Visualizing the Execution of Functional Programs

Rudolf Berghammer and Ulf Milanese

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel
Email: `rub|umi@informatik.uni-kiel.de`

A very prominent way to define the operational semantics of functional programs (which is especially qualified for beginners) is the use of computation sequences. The elements of a computation sequence are expressions which are one after another produced by substitutions and simplifications. Substitution corresponds to routine calls or, more precisely, to applications of a computation rule. Such a rule determines which calls have to be replaced by the body, where additionally in each case within the latter the formal parameters are substituted by the arguments. For instance, in the classical case of routines without patters, the full-substitution rule says that all calls have to be replaced simultaneously, the leftmost-outermost rule says that from all calls which do not occur as arguments of other calls only the leftmost one has to be replaced, and the leftmost-innermost rule says that from all calls which are free from other calls only the leftmost one has to be replaced; see e.g., [4]. If patterns are allowed in routine declarations, then the computation rules are a little bit more complicated. Simplification executes operations of the pre-defined datatypes and evaluates conditionals.

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. This allows, among other things, to demonstrate the effect and efficiency of the various computation rules, the difference between strictness and non-strictness, the possibility of using infinite objects in the latter case, and the inherent "freedom of computation" in functional programs. Furthermore, it enables debugging of programs by step-wise computations to detect weak points and logical errors. User-friendly debugging of functional programs is still a research problem since many traditional approaches known for imperative programms are not suited for functional ones. And, finally, moving step by step through a computation using a system can elucidate the decisive ideas behind the algorithm a functional program implements. This may lead to new insights and improvements.

In the following, we describe a computer system for visualizing the execution of first-order functional programs via substitutions and simplifications. It is called KIEL, which is an acronym for "Kiel Interactive Evaluation Laboratory",

```
datatype stree = empty | mk of stree * int * stree;

fun insert (n : int, empty : stree) : stree = mk(empty,n,empty)
  | insert (n : int, mk(l,x,r) : stree) : stree =
    if n<=x then mk(insert(n,l),x,r) else mk(l,x,insert(n,r));

fun member (n : int, empty : stree) : bool = false
  | member (n : int, mk(l,x,r) : stree) : bool =
    if n<x then member(n,l) else if n>x then member(n,r) else true;

fun isempty (empty : stree) : bool = true
  | isempty (mk(l,x,r) : stree) : bool = false;

fun list2stree (nil : int list) : stree = empty
  | list2stree (x::s : int list) : stree = insert(x,list2stree(s));
```

Fig. 1. Implementation of binary search trees in KIEL.

and has especially been designed for teaching undergraduate students of computer science. The system is written in C and uses the libraries GTK, GDK, and GLib for its graphical user interface. 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 (like a user's manual [1], the exact hardware and software requirements, how to get the system, examples for KIEL programs etc.) and news are published on the system's Web-site http://www.informatik.uni-kiel.de/~progsys/kiel.html. On this Web-site also an online version of the system is available as a Java applet.

The KIEL system supports a simple subset of Standard ML which is closely related to the part of this programming language described in Sections 3.1–3.3, 5.1.1, 5.1.2, and 6.2 of the textbook [3]. In particular, it allows the definition of datatypes with value constructors (i.e., via the so-called `datatype`-binding) and the use of pattern matching in declarations of routines. But it does not allow the use of higher-order functions, local declarations, and polymorphism. In the following, we only sketch the system's programming language; an exact description by means of syntax diagrams can be found on the system's Web-site.

There are five pre-defined datatypes in KIEL, viz. the singleton type `unit` and the types `bool` for the truth values, `int` for the integers, `real` for the reals, `string` for the strings, and m `list` for the lists over a pre-defined or user-declared datatype m. In each case the operations are a subset of the respective operations of Standard ML. For instance, on the integers KIEL provides `+`, `-`, and `*` as three arithmetic operations, `div` and `mod` for computing integer divisions and remainders, `min` and `max` for computing minima and maxima, `abs` for computing absolute values, and `=`, `<>`, `<`, `<=`, `>`, and `>=` as six comparison operations.

A program in KIEL consists of a sequence of declarations, where a declaration either defines a single datatype resp. routine or a system of (generally mutually recursive) datatypes or routines. The form of a datatype and routine declaration is as in Standard ML with two exceptions. Due to the absence of polymorphism, but also for methodical reasons, KIEL requires routines to be fully typed, where the type m of a formal parameter or a pattern x is given within the parameter list in the form x `:` m and the type of the result is given after the parameter list,

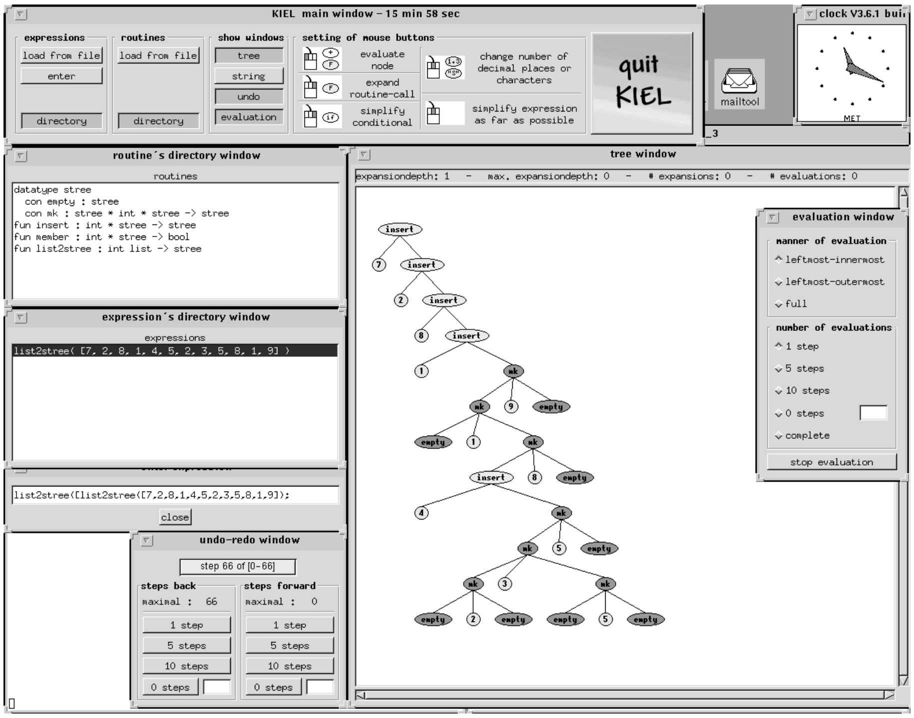

Fig. 2. The screen of KIEL.

again separated by a colon. Furthermore, the system uses for calls of routines the classical mathematical notation $F(A)$ instead of $F\ A$ or $(F\ A)$ as Standard ML allows. The main reason for this is the absence of higher-order functions. But the classical notation is also motivated by the fact that KIEL is especially used by beginners which are accustomed to it from high school. As an example, the KIEL-program of Figure 1 implements binary search trees over the integers with four routines for inserting a new element into a tree, testing for membership in a tree resp. emptiness of a tree, and generating a tree from a list of integers.

KIEL is a completely graphic-oriented computer system. Although the detailed appearance of its windows depends on resources defined in two configuration files, when working with it the screen typically looks as in Figure 2. The tree depicted in the "tree window" of this picture represents an expression which may occur when the call `list2stree([7,2,8,1,4,5,2,3,5,8,1,9])` is evaluated.

From the user's point of view, roughly spoken the KIEL system consists of two main components. The first main component is the workspace. It holds a program of the system's programming language and the expressions to be evaluated. Programs are loaded into the system from an ordinary text file. Expressions can also be loaded from a text file, but in the usual way the user enters single expressions directly via a specific window. The system's second main component is the evaluation unit. Here an expression of the workspace is evaluated based on the pre-defined datatypes and the program contained in the workspace.

Evaluation of expressions can be done step by step using the mouse, where the various actions invoked by the mouse buttons are explained by five pictures in the middle of KIEL's main window. E.g., the substitution of a routine call is possible by selecting the corresponding vertex of the tree and then pressing the right mouse button and to simplify an expression as far as possible one only has 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 these possibilities, it is very easy to simulate the operational semantics definition via computation sequences as described at the beginning of this paper.

But evaluation of expressions can also be done semi-automatically or fully-automatically. In these cases the user can choose a specific computation rule, control and count the number of computation steps, undo and redo computations etc. via some specific dialog windows. As the "evaluation window" of Figure 2 shows, the current version of KIEL allows exactly the choice between the three computation rules mentioned at the beginning of this paper, viz. leftmost-innermost substitution, leftmost-outermost substitution, and full substitution.

We have performed a lot of case studies with KIEL and the system has turned out to be a good tool for introductory courses into functional programming – or even into computer science if one follows the ideas presented in [2]. Especially the possibility of undoing and redoing steps has been proved to be very helpful. It enables "walking" through computation sequences which can be used for many purposes, e.g., visualization of program execution, debugging of programs, and testing alternative computations. To a certain extent, the KIEL system can also be used in graduate courses and practicals. Based on the experience gained, we believe that visualizing program execution introduces students far more efficiently into the intricacies of parameter passing mechanisms, the differences between the various recursion schemes, the essence of algorithm design principles, computational complexity etc. than blackboard or textbook explanations.

Because of its restricted language, however, the current version of KIEL cannot be used to teach advanced concepts of functional programming languages and important abstract functional programming concepts (like partial evaluation, deforestation, memorization, etc.) based on them. To extend the system's scope, therefore, it is planned to incorporate more advanced language constructs in future versions. As a first step in this direction, we plan to include local declarations; the next step then will be the extension by higher-order functions.

References

1. Berghammer R., Tiedt M,: Kiel Interactive Evaluation Laboratory. Bericht Nr. 9909, Inst. für Informatik und Praktische Mathematik, Universität Kiel (1999)
2. Giegerich R., Hinze R., Kurtz S.: Straight to the heart of computer science via functional programming. In: Felleisen M. et al. (eds.): Proc. FDPE '99, Report Rice COMP TR99-346, Dept. of Computer Science, Rice University, 1-5 (1999)
3. Ullman J.D.: Elements of ML programming. 2nd edition, Prentice Hall (1998)
4. Vuillemin J.: Correct and optimal implementation of recursion in a simple programming language. Journal of Computer and System Sciences 9, 332-354 (1974)