Using π–RED as a Teaching Tool for Functional Programming and Program Execution

Werner E. Kluge, Carsten Rathsack, Sven-Bodo Scholz

Christian-Albrechts-Universitaet Kiel, Institut fuer Informatik, D–24105 Kiel, Germany, E–mail: base@informatik.uni–kiel.d400.de

Abstract. *This paper describes an interactively controlled reduction system π–RED which may be employed to teach both basic concepts of functional programming and program execution at the undergraduate level, and advanced topics of computer science at the graduate level.*
π–RED supports the reduction languages KiR *- a sugared versions of an applied λ–calculus with a full-fledged (type-free) β-reduction. Functions (λ-abstractions) are truly treated as first class objects: they may be applied to other functions or to themselves, and λ-abstractions may be computed as function values. Name clashes are resolved by an indexing scheme similar to de Bruijn indices.*
Programs may be executed in a stepwise mode and intermediate programs may be displayed in high-level notation.
Special pattern matching constructs provide the means to directly specify in KiR *complex term rewrite systems. In conjunction with the stepwise execution mode, this feature has been extensively used, in lab courses accompanying graduate courses on computer architecture, to rapidly prototype and study various programming and (abstract) machine models, compilation schemes, type checkers, etc.*

1 Introduction

Functional or function-based programming languages such as Miranda, standard ML, Haskell or Scheme [Turn86, HaMiTo88, HuWa88, AbSus85] have become increasingly popular in teaching basic programming concepts at the undergraduate level as well as advanced topics such as compiler construction, type systems, abstract machines | language interpreters etc. in graduate CS courses.

The functional paradigm derives its appeal primarily from its declarative programming style. Functional programs are pure algorithms which are generally simpler and faster to design, more concise and less vulnerable to errors than their imperative counterparts, and can be understood without mentally executing them.

The underlying semantic models (either the λ-calculus or a combinatory calculus) define program execution as processes of meaning-preserving program transformations, i.e., program execution conceptually takes place entirely within the space of programs, not in a state space [Back78]. A computing machine which faithfully implements this concept can be made to execute, under interactive control, programs step by step, and to display intermediate programs in the same notation in which the original program was specified. Moreover, the focus of control may be freely moved about initial and intermediate programs to select other than top-level redexes for evaluation, without causing side-effects in other program parts (referential transparency).

The benefits of using such a machine in CS education are manifold. Beyond learning an elegant programming style, students can be expected to develop very quickly a conceptual understanding of how computations are actually taking place at a level of abstraction very similar to algebraic formula manipulations. All that there is visible (and relevant) at this level, and can therefore be studied in a clean setting, are the program transformations effected by the rewrite (reduction) rules of the underlying calculus, the control discipline which drives computations step by step ahead, and the operational discipline which supplies operators with operands, e.g., instantiates functions with arguments. Moving step by step through a sequence of intermediate programs also facilitates program validation and the detection of logical errors (high-level debugging). To some extent, it may even substitute for inductive proofs on program correctness, rather than doing them by tedious paper-and-pencil work.

Additional advantages from an educational point of view come with a full λ-calculus. The β-reduction rule provides a versatile mechanism for the substitution of variables by λ-terms which maintains static binding structures (scopes) even in the presence of naming conflicts. This mechanism is absolutely essential for simple symbolic computations involving free variables, and specifically for supporting higher-order functions in the sense that functions (λ-abstractions) may be freely applied to other functions (and to themselves), and that functions may be returned as function values.

Teaching an applied λ-calculus as a programming language is the key to fully understanding elementary properties of operators, their application to operands, the role of variables as place-holders, the dual roles of functions (abstractions) as operators and operands, and the systematic construction of complex operators from simpler ones. Visualizing β-reductions step by step introduces students to the intricacies of λ-calculus and of the mechanical evaluation of λ-terms far more effectively than textbook or blackboard explanations.

Another important concept of functional programming is pattern matching - a mechanism which extracts substructures from given structural contexts, and substitutes them for variables in other structural contexts. It is the key to teaching a number of advanced topics in computer science as it allows one to specify in a very concise form term rewrite systems. They may prototype compilers, type checkers, data base (expert) systems, abstract or concrete computing machines etc., or generally systems whose dynamic behavior can be described (or approximated) by discrete rule-based state changes.

The design of complex term rewrite systems benefits considerably from a stepwise execution mode. This is particularly true in education where students need to develop quickly some basic understanding of how abstract concepts or models actually work. The effects of (sequences of) rule applications can immediately be visualized in high-level notation and checked for consistency, correctness and confluence (if so required). Logical (specification) errors may thus be detected and corrected at an early stage, and progress towards achieving a course's objectives can be made without getting lost in irrelevant, often frustating details of low-level programming.

The advantages of visualizing individual steps of program execution for educational purposes have been well recognized elsewhere. Goldson describes a system called MIRACALC [Gol94] which supports a stepwise execution mode for MIRANDA programs. He points out that students can be expected to benefit considerably from viewing functional computations as rule-based evaluation processes and to enhance their '... creativity in specifying, designing and reasoning about their programs by relieving them of the need to carry out pencil-and-paper calculations ...'.

Unfortunately, almost all functional systems proposed and implemented to date do not support stepwise program execution [Turn76, John83, Car83, Plas93, Pey92]. Functions (abstractions) are usually compiled to code of some abstract or real machine which constructs and reduces graph representations of the program terms to be evaluated. Since intermediate states of program execution are distributed over program counters, stacks, registers and heap locations which render re-compilation into equivalent high-level programs an intractable problem, the code must run to completion to produce meaningful results.

Compilation to code to some extent also separates the world of functions from the world of objects they operate on: function code must be supplied with full sets of arguments to execute correctly, and the code itself cannot be modified, i.e., there is no direct way of computing new functions. Partial applications are wrapped up in closure until the missing arguments can be picked up. If the computation terminates with a closure, the user is simply notified of the fact that the result is a function, but it cannot be made visible in high-level notation. Further restrictions come with the type systems of these languages. They rule out certain higher-order functions, especially self-applications, as they cannot unify recursive types.

In this paper, we will give an outline of the reduction system π–RED which fully complies with the features of a good teaching tool as outlined above. It supports an interactively controlled execution mode for a syntactically sugared version of an applied λ-calculus called KIR (for Kiel Reduction Language), which also includes sophisticated pattern matching facilities. Rather than discussing the internal workings of this system (which is described elsewhere [GaKl94]), we will here concentrate on what the programmer can see of and how he / she can work with π–RED at the user interface, and on typical programming and program execution exercises that can be done with this system.

In the sequel we will first introduce, in section 2, a kernel of KIR , the basic idea of editing KIR programs in a syntax-directed fashion, and the means to control their stepwise execution. In section 3 we demonstrate, by means of selected examples, how stepwise execution works on λ-terms, how new abstractions can be computed from existing ones, and how naming conflicts are resolved in the course of β-reductions. In section 4 we introduce the pattern matching constructs provided by KiR , particularly those that are necessary to specify term rewrite systems on a KiR platform, and in section 5 we present, as a typical application, an interpreter for the SECD machine. In section 6 we summarize some of our experiences in using π–RED as a vehicle for teaching basic concepts of functional computing in a graduate lab course.

2 The Programming Environment of π–RED

π–RED is a complex software system of about 2 MBytes C-code which can be installed on all UNIX-based workstations. It supports the reduction language KiR – a sugared version of an applied, type-free λ-calculus with an applicative order semantics [Klu94]. The basic concept has been adopted from Berkling's string reduction machinery developed as early as 1975 [Berk75]. The current version available for distribution reduces graphs by interpretation of abstract machine code. On average, it is about ten times slower than functional languages compiled to executable code [GaKl94].

Both programming and program execution are in π–RED supported as fully interactive processes. The system provides a syntax-directed editor which allows to construct KiR programs either top-down or bottom-up by systematically inserting expressions into place-holder positions of other expressions, or by replacing expressions with others. Program execution is realized as a sequence of meaning-preserving program transformations based on a fixed set of rewrite rules, of which the most important one is a full-fledged β-reduction. Programs may either run to completion, or be reduced in a stepwise mode, with intermediate programs being displayed to the user for inspection and modification.

With a few minor exceptions, the syntax of KiR resembles that of most other functional languages. The following is a kernel of KiR [1]:

```
e = v | c | pf
    | ( e_0 e_1 .. e_n )
    | IF e_0 THEN e_1 ELSE e_2
    | LAMBDA x_1..x_n IN e
    | LET .. x  =  x_e .. IN e
    | LETREC .. f x_1 .. x_n  =  f_e .. IN s_e
    | < e_1..e_n >   .
```

This syntax defines KiR programs as expressions composed of variables (denoted as `v`), constants (denoted as `c`), primitive functions (denoted as `pf`), applications of function expressions (`e_0`) to n argument expressions (`e_1` ... `e_n`), `IF_THEN_ELSE` clauses as special applications, anonymous functions (`LAMBDA` abstractions), `LET` clauses which specify values for variables in some expression `e`, `LETREC` constructs which specify sets of mutually recursive functions, and lists

[1] As a courtesy to the HASKELL community, this syntax includes a few modifications of KiR as it is actually implemented, particularly w.r.t. the use of parentheses and keywords.

(sequences) of expressions as a means to represent structured objects, respectively.

The set of primitive functions includes the familiar binary arithmetic, logical and relational operators, and some primitive structuring and predicate functions applicable to lists.

As a convention, KiR uses upper case letters to denote keywords such as **LETREC, IF ,THEN, IN** etc., whereas all identifiers (variables) need to start with lower case letters.

KiR is an untyped language. Type compatibility checks between primitive functions and the operands to which they are actually applied are dynamically performed at run-time based on type-tags that are carried along with all objects of a program.

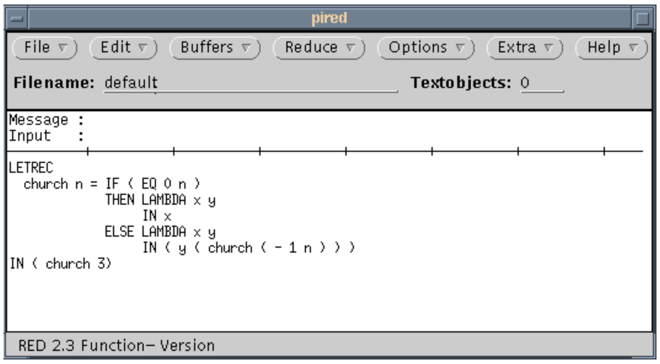

Fig. 1. X-Windows front-end of π–RED

Fig. 1 shows the screen layout of the user interface of π–RED. It consists of a window whose main part emulates an alphanumerical terminal on which programs may be edited and displayed. This terminal divides into two parts separated by a horizontal line. The upper part includes a message line and an input line, and the lower part displays the segment of the KiR program in which reductions are actually taking place (or constitutes the actual focus of control). This expression field is initialized with the symbol `#` which stands for an empty expressions.

In order to edit a program, the user may simply type some fragment of a legitimate KiR expression, say just the key word `IF`, which is immediately echoed on the input line, and the message line responds with `<expr>` , indicating that so far the input line contains a syntactically correct expression. Upon hitting the return key, both the message line and the input line will be cleared, and a syntactically complete `IF_THEN_ELSE` construct with as yet uninstantiated (or empty) components shows up in the expression field:

```
IF # THEN # ELSE #
```

The missing components may be specified in the same way, say as `( EQ 0 n`, `LAMBDA x y IN x`, and `LAMBDA x y IN ( y ( church ( - 1 n` in this sequence, with the cursor automatically moving through the empty positions `#` from left to right, to obtain [2]

```
IF ( EQ 0 n )
THEN LAMBDA x y
     IN x
ELSE LAMBDA x y
     IN ( y ( church ( - 1 n ) ) )   .
```

All there remains to be done to turn this expression into the full program shown in fig. 1 is to type in the input line

```
LETREC church n = %  ( church 3 ,
```

where the `%` is taken as a place-holder for the entire expression that is already set up in the expression field. Upon entering this input, π–RED produces in the expression field [3]

```
LETREC
  church n =
    IF ( EQ 0 n )
    THEN LAMBDA x y
         IN x
    ELSE LAMBDA x y
         IN ( y ( church ( - 1 n ) ) )
IN ( church 3 )     .
```

This program computes representations of natural numbers as Church numerals, in the particular case the Church numeral of 3.

Syntactically incorrect input is immediately rejected by the editor, and some comment explaining the likely cause of the error appears in the message line. π–RED accepts for execution only programs which are syntactically correct and complete, i.e., do not contain empty components.

The window frame provides a number of menus which may be entered by clicking the respective buttons. Among them are the menus FILE which includes a number of file handling functions to store and retrieve KIR expressions, EDIT which provides editing functions such as `copy` and `paste`, and also access to a line-oriented editing mode, REDUCE which includes some functions to control reductions, e.g., a single step mode, modes which perform pre-specified or unlimited numbers of reductions per shift, and functions which move the cursor about the program to select other than top-level redexes for evaluation.

[2] Note that in the input only opening parentheses need be specified, the editor adds automatically the respective closing parentheses if no ambiguities occur.

[3] The layout of this program is exactly as produced by the editor. However, in the sequel we had to change the editor layout occassionally by hand in order to make the programs fit the column format of this paper.

3 λ-Calculus and Stepwise Reductions

In this section we will work through a few programming exercises to show how π–RED may be used to teach an applied λ-calculus. In particular, we will demonstrate how functions (λ-abstractions) may be computed as results of function applications, and how naming conflicts are resolved by β-reductions which use a special indexing scheme to distinguish identically named variables that are bound in different scopes. By means of these programs we will also show how π–RED goes about reducing programs step by step.

We set out with the program for Church numerals as edited in the preceding section, which computes closed λ-abstractions (or combinators) as results. π–RED may be set up, by means of the REDUCE controls, to perform in one shift as many reduction steps as are necessary to move from one recursive call of the function `church` to the next, with the parameter n each time decremented by one. The number of reductions to be performed in one such shift may be found out by first doing reductions step by step and by counting the number of steps necessary to complete a recursive call of `church`. After the first shift, which in the particular case takes three steps, we thus get

```
LAMBDA x y
IN ( y LETREC
         church n =
           IF ( EQ 0 n )
           THEN LAMBDA x y
                IN x
           ELSE LAMBDA x y
                IN ( y ( church ( - 1 n ) ) )
       IN ( church 2 ) )     ,
```

i.e., the initial program is being replaced by the `ELSE`-clause of the function body of `church`, with the parameter `n` substituted by 3, with `church` substituted by the entire `LETREC`-construct, and evaluated to the point where nothing else is left to do but to reduce the function call `( church 2 )`. Another three shifts of three reduction steps (of which only two are used up by the last shift) returns the Church numeral of the value $n = 3$, which is the λ-abstraction

```
LAMBDA x y
IN ( y LAMBDA x y
       IN ( y LAMBDA x y
              IN ( y LAMBDA x y
                     IN x ) ) )  .
```

We may now make this λ-term the argument of a successor function applicable to Church numerals by typing as input

```
LET suc = LAMBDA z x y
          IN ( y z
IN ( suc % ,
```

which in the expression field returns as

```
LET suc = LAMBDA z x y
          IN ( y z )
IN ( suc LAMBDA x y
         IN ( y LAMBDA x y
                IN ( y LAMBDA x y
                       IN ( y LAMBDA x y
                              IN x ) ) ) ) .
```

Here we have an instance of a partial application of the function `suc` to the Church numeral of 3, which by means of one β-reduction transformes into the Church numeral of 4 by adding another line `IN ( y LAMBDA x y` .

The next program specifies a simple symbolic computation which also includes a self-application. It defines, under the `LET`-variable `y`, the Y-combinator of the λ-calculus as it must be used under an applicative order reduction regime as supported by π–RED:

```
LET y = LAMBDA h
        IN LET a = LAMBDA x
                   IN ( h LAMBDA y
                          IN  ( x x y ) )
           IN ( a a )
IN ( ( y f ) r )   .
```

In the body of this `LET`-construct, the variable `y` is applied to the free variables `f` and `r` which fill in for the abstraction that is to be recursively called and for some argument, respectively.

The first five β-reductions transforms this program into

```
( f LAMBDA y
    IN ( f LAMBDA y
       |   IN ( LAMBDA x
       |        IN ( f LAMBDA y
       |               IN ( x x y ) )
       |           LAMBDA x
       |           IN ( f LAMBDA y
       |                  IN ( x x y ) )
                y ) y ) r ) .
```

Further β-reduction steps reproduce the self-application marked by the vertical bars, embedded in an application `( f LAMBDA y IN ... y )` . Thus, after one more step we get

```
( f LAMBDA y
    IN ( f LAMBDA y
           IN ( f LAMBDA y
                  IN ( LAMBDA x
                       IN ( f LAMBDA y
                              IN ( x x y ) )
                          LAMBDA x
                          IN ( f LAMBDA y
                                 IN ( x x y ) )
                       y ) y ) y ) r ) ,
```

and so on, which nicely visualizes the recursive expansion of the original program. With the free variables `f` and `r` substituting for some recursive abstraction and its argument, respectively, this expression would keep expanding without bounds.

However, in a subsequent editing step, the symbolic program thus obtained may be made the body of an abstraction of the two variables `f` and `r` and applied to two as yet unknown arguments by typing as input `( LAMBDA f r IN % # #`. This completes what is already in the expressions field to

```
( LAMBDA f r
  IN ( f LAMBDA y
         IN ( f LAMBDA y
                IN ( f LAMBDA y
                       IN ( LAMBDA x
                            IN ( f LAMBDA y
                                   IN ( x x y ) )
                               LAMBDA x
                               IN ( f LAMBDA y
                                      IN ( x x y ) )
                            y ) y ) y ) r ) # # ) .
```

The editor places the cursor automatically on the first of the two empty syntactical positions `#`. This position may now be replaced, say, by an abstraction which computes factorial numbers by typing into the input line and entering

```
LAMBDA fac
IN LAMBDA n
   IN  IF ( GT 1 n )
       THEN ( * n ( fac ( - 1 n ) )
       ELSE 1 .
```

The editor then moves the cursor to the next `#` which may be replaced by an argument value for the function `fac`, say 100.

The program thus completed, when run with unlimited reduction steps, eventually returns the factorial number of 100, which is [4]

```
93326215443944152681699238856266700490715968264381621468592963895
21759999322991560894146397615651828625369792082722375825118521091
6864000000000000000000000000
```

Termination of this program is enforced by the conditional included in the abstraction `fac`, which may even cut off superfluous expansions of the Y-combinator beyond the number of recursive calls required to compute a function value.

The programs discussed so far apply functions (λ-abstractions) to functions, and return functions as values. All β-reductions are in fact done naively, i.e., without regard for naming conflicts, since all abstractions other than the successor function `suc` are applied to full sets of argument terms, and all abstractions that are passed along as arguments are closed (or combinators).

[4] Note that π–RED may be set up to perform arithmetic operations on unformatted decimal number representations which returns result values without truncation.

Full β-reductions enter the game in cases of naming conflicts between free and bound variable occurrences. Rather than renaming in such situations the bound variables involved in the conflicts, as the classical definition of the β-reduction rule demands, π–RED uses an indexing scheme similar to deBruijn indices [Brui72, BeFe82b] to distinguish different binding scopes of identically named variables: a variable occurrence, say `x`, may be preceded by some n backslashes \ which offset (or protect it against) the innermost n binders `LAMBDA x`. For instance, in an abstraction

```
LAMBDA x
IN LAMBDA x
   IN ( x ( \x \\x ) )
```

the `x` is bound to the innermost `LAMBDA x`, the `\x` is bound to the outermost `LAMBDA x` but protected against the innermost `LAMBDA x`, and the `\\x` is protected against both binders (or free in the entire term). β-reductions may add backslashes to free variables when inserting them into the scopes of binders to the same name, and may remove backshlashes when consuming binders against which they are protected.

This may be exemplified by the following program:

```
( LAMBDA x
  IN ( LAMBDA y
       IN ( LAMBDA y
            IN ( LAMBDA y
                 IN ( y x ) x ) x )
       x ) ( y z ) ) .
```

When β-reducing the outermost application, the argument term `( y z )`, in which both `y` and `z` occur free, is being substituted for free occurrences of `x` in the body of the abstraction `LAMBDA x IN ( ... )`. In doing this, the free variable `y` penetrates the scopes of three nested binders `LAMBDA y` and thus creates naming conflicts. The β-reduction rule as implemented in π–RED resolves them thus:

```
( LAMBDA y
  IN ( LAMBDA y
       IN ( LAMBDA y
            IN ( y ( \\\y z ) ) ( \\y z ) )
       ( \y z ) ) ( y z ) ) .
```

Here we find the variable `y`, which as part of the application `( y z )` has been substituted for free occurrences of `x`, preceded by three, two , one and no backslashes, according to the number of `LAMBDA y` it has crossed [5]. None of these occurrences of `y` are bound to a `LAMBDA y`, i.e., they maintain their original status of being free in the entire expression. When performing further β-reductions from outermost to innermost, backslashes are taken away as binders `LAMBDA y` successively disappear.

Thus, another three β-reductions carried out step by step return

[5] The variable `z` remains as it is since there are no binders `LAMBDA z` whose scopes are being penetrated.

```
( LAMBDA y
  IN ( LAMBDA y
       IN ( y ( \\y z) ) ( \y z ) ) ( y z ) ) ,
```

followed by

```
( LAMBDA y
  IN ( y ( \y z ) ) ( y z ))   ,
```

and terminating with

```
( ( y z ) ( y z ) )    .
```

In conjunction with a recursive version of the α-conversion function $\lambda u.\lambda v.(uv)$, this implementation of the β-reduction rule may also be used to α-convert an abstraction of some n λ-bound variables into a form in which all of them are identically named, and variable occurrences in different scopes are distinguished by different numbers of backslashes.

The program

```
LETREC
  alpha = LAMBDA u
            IN LAMBDA v
                 IN IF ( IS_FUNC u )
                 THEN ( alpha ( u v ) )
                 ELSE u
IN ( alpha LAMBDA x
             IN  LAMBDA y
                   IN  LAMBDA z
                         IN ( x ( y z x ) ) )
```

does this job, returning after a sequence of several reduction steps the abstraction

```
LAMBDA v
  IN LAMBDA v
       IN LAMBDA v
            IN ( \\v ( \v v \\v ) ) .
```

When dropping from this abstraction the variable name v altogether, we get its deBruijn representation with nameless indices, which in plain λ-term notation is $\Lambda\Lambda\Lambda.$ (#2 (#1 #0 #2)) [6].

4 Pattern Matching

Pattern matching is an operation which extracts (sub-) structures of particular shapes and elements from given structural contexts and substitutes them for place-holders in other structural contexts. It is available in all modern functional languages [Turn86, Lavi87, HuWa88] and primarily used as an elegant way

[6] Note that in order to get deBruijn indices as originally defined, these indices have to be incremented by one.

of defining functions as ordered sets of alternative equations. However, its full potential lies in complex rule-based modifications of structured objects.

Pattern matches are in KiR defined as so-called `WHEN`-clauses which have the syntactical form

`WHEN pat_1 pat_2 ... pat_n GUARD guard DO p_e`,

where `pat_i` denotes either a variable, a constant, or a (recursively nested) sequence of these items. The `WHEN` constructor, similar to `LAMBDA`, binds free occurrences of the pattern variables in the optional guard expression `guard` and in the body expression `p_e`. Applications of `WHEN`-clauses to n argument terms must be embedded in a `CASE`-construct (which in general may include several alternative `WHEN`-clauses):

```
( CASE
  WHEN pat_1 ... pat_n GUARD guard DO p_e
  END_CASE
  a_1 ... a_n )  .
```

If the syntactical structures of the patterns and of the corresponding (evaluated) argument terms match w.r.t. all pairs `( pat_i | a_i )` then all free occurrences of the pattern variables in `guard` and `p_e` are substituted by the respective argument (sub-) structures. If, in addition, the guard expression thus instantiated evaluates to `TRUE`, then the entire application is replaced by the instantiated body `p_e`. Otherwise, the arguments are considered not within the domain of the pattern (or not of a compatible type), and the application remains as it is.

A simple example may help to convey how pattern matching is implemented in π–RED:

```
( CASE
  WHEN < u v > y 2
    GUARD ( NOT_EMPTY ( TAIL y ) )
    DO ( u y )
  END_CASE
    < REVERSE TAIL > < 1 2 3 4 > 2 ) .
```

Here the `WHEN`-clause specifies a list of two variables, a single variable and one constant as simple patterns. The guard tests whether the argument substituted for the variable `y` is a list of at least two elements. Since the arguments are a list of primitive functions `< REVERSE TAIL >` in the first position, a list of four numbers in the second position, and the constant value `2` in the third position, π–RED returns after a first step

```
( CASE
  WHEN  . . .
   GUARD ( NOT_EMPTY ( TAIL < 1 2 3 4 > ) )
   DO ( REVERSE  < 1 2 3 4 > )
  OTHERWISE
    ( CASE
        WHEN < u v > y 2
```

```
        GUARD ( NOT_EMPTY ( TAIL y ) )
        DO ( x y )
      END_CASE
        < REVERSE TAIL > < 1 2 3 4 > 2 )
  END_CASE
    < REVERSE_TAIL > < 1 2 3 4 > 2 )    .
```

The patterns in the WHEN-clause are now replaced by dots, indicating that they did match the arguments, and all occurrences of the pattern variables in both the guard and the body are substituted, i.e., u by the function REVERSE and y by the list < 1 2 3 4 >. Moreover, the CASE construct has been extended by an OTHERWISE expression composed of the complete original application. This is the escape hatch for the case that the guard expression reduces to something other than the Boolean constant TRUE. However, in this particular case, another three reduction steps will evaluate the guard to TRUE and throw this alternative away, returning (REVERSE < 1 2 3 4 >) , which in one more step reduces to < 4 3 2 1 > .

In general, a CASE-expression may include several WHEN-clauses and an optional OTHERWISE expression. Applications of it have the general syntactical form

```
( CASE
   WHEN pat_11 ... pat_1n GUARD guard_1 DO p_e_1
      :
   WHEN pat_m1 ... pat_mn GUARD guard_m DO p_e_m
   OTHERWISE esc_e
  END_CASE
   a_1 ... a_n ) .
```

As usual, all patterns are tried from top to bottom. The first successful match, including a satisfiable guard, is reduced to its normal form which, by definition, is also the normal form of the entire CASE application. If none of the pattern | guard combinations matches, then the result is the normal form of the OTHERWISE expression if explicitely specified or, if this specification is missing, the CASE application itself in unreduced form.

Pattern matching is also applicable to structured objects made up from user-specified constructors. They enable the programmer to define on top of KiR meta-languages whose interpretation may be freely specified by sets of rewrite rules. This feature, among many other applications, may be used to prototype and test abstract machines, compilation schemes, type checkers etc. for other programming languages, imperative or declarative, or for KiR itself (in fact, this has been extensively done to prototype advanced versions of π–RED).

A meta-level expression has the syntactical form:

constr{e_1 e_2 ... e_n}

where constr is a character string beginning with a letter which, in conjunction with the curly brackets { and } that enclose a sequence of n KiR expressions e_1,...,e_n, is taken as an n-ary user-specified constructor. The expressions e_1,...,e_n may be meta-level expressions themselves.

A pattern that matches against such an expression must feature the same constructor name and the same arity, and may have as components only variables, constants, and recursively non-atomic patterns.

5 A Case Study : the SECD Machine

Let us now look at an interpreter for Landin's SECD machine [Lan64] to demonstrate the expressive power of pattern matching. The SECD machine, in its simplest form, is an abstract applicative order evaluator for expressions of the pure λ-calculus. It derives its name from the four stacks it uses as a run-time environment:

- S holds evaluated λ-terms in essentially the order in which they are encountered during the traversal of the original λ-expression;
- E holds name/value pairs for instantiations of λ-bound variables;
- C is a control structure which accommodates the (part of the) expression that still needs to be processed;
- D serves as a dump for actual machine states that need to be saved when entering into the evaluation of another application.

The operation of this machine is specified in terms of a state transition function

$$\tau \; : \; (\; S \; E \; C \; D) \; \rightarrow \; (\; S' \; E' \; C' \; D')$$

which maps current into next machine states. To do so, the function τ must distinguish among six basic constellations on the stack-tops of components of λ-terms. These constellations can be readily expressed in terms of pattern matches collected in a `CASE` construct.

The λ-terms to be processed by this machine may be specified in the following meta-syntax in order to distinguish them from the λ-terms of KIR proper:

$$\texttt{term} \;\rightarrow\; \texttt{var}\{\texttt{x}\} \mid \texttt{lambda}\{\texttt{x term}\} \mid \texttt{apply}\{\texttt{term term}\} \; .$$

Variables, abstractions and applications are in this syntax special constructor expressions. In addition we need closures of the general form `clos{e x term}` which define abstractions `lambda{x term}` as values in environments `e`.

The specification of the machine may be based on the representation of its state (or of the run-time environment) as a sequence of four subsequences, denoted as `< s e c d >`, which correspond to the four push-down stacks. The state transition function τ can then be implemented by means of a recursive function `secd` whose body is made up from a `CASE` construct, as shown in fig. 2. It is composed of seven pattern matches, of which the first one defines the terminal state, and the other six specify the state transition rules for λ-terms.

In this program, the function `secd` is applied to a concrete λ-term of the meta-language which is initially set up in the control structure C, with all other structures (stacks) empty. Variable names must be specified in this syntax as constants, i.e., as character strings embedded in quotation marks, otherwise

```
LETREC
  secd state =
   ( CASE
     WHEN < s e <> <> >
       DO  < < s e <> <> > 'done` >
     WHEN < < w s > e <> < ss se sc sd > >
       DO ( secd < < x ss > se sc sd > )
     WHEN < s e < var{x} c > d >
       DO ( secd < < ( lookup x e ) s > e c d > )
     WHEN < s e < lambda{x term} c > d >
       DO ( secd < < clos{e x term} s > e c d > )
     WHEN < s e < apply{e_1 e_2} c > d >
       DO ( secd < s e < e_2 < e_1 < ap{ } c > > > d > )
     WHEN < < clos{e x term} < arg s > > e < ap{ } c > d >
       DO ( secd < <> < < x arg > e > < term <> > < s e c d > > )
     WHEN < < fun < arg s > > e < ap{ } c > d >
       DO ( secd < < apply{fun < arg >} s > e c d > )
     OTHERWISE < state 'irregular state` >
     END_CASE state )
  lookup var env =
   ( CASE
     WHEN <> DO var{var}
     WHEN < < x e > z > GUARD ( EQ var x ) DO e
     WHEN < x z > DO ( lookup var z )
     OTHERWISE < env 'erroneous environment` >
     END_CASE env )
IN ( secd < <> <> < apply{apply{lambda{'u`
                                  lambda{'v` apply{var{'u`} var{'v`}}}}
                            lambda{'x` var{'x`}}} var{'w` }} <> > <> > )
```

Fig. 2. KIR specification of an SECD machine

they would become variables of KIR but not of the meta-language, and as such could get bound outside the intended context.

A rule application in this program generally takes three reduction steps, one each to perform a recursive call of `secd`, to successfully match one of the patterns against the new machine state passed along through the parameter `state`, and to produce the new state by evaluation of the body of the matching pattern. There is one exception though: evaluating the body of the third `WHEN`-clause usually takes several steps since it involves calling the recursive function `lookup` to substitute a variable appearing on top of the control structure C by the term with which it is instantiated in the actual environment.

The following shows three typical intermediate programs (in which the function definition parts are abbreviated by question marks since they remain unchanged), and the final machine state resulting from the successful application of the first pattern match.

After five state transitions of three reduction steps each π–RED returns the KIR program

```
LETREC
      ?
      ?
IN ( secd
     < < clos{<> 'u'
              lambda{'v' apply{var{'u'} var{'v'}}}}
       < clos{<> 'x' var{'x'}} < var{'w'} <> > > >
     <>
     < ap{ } < ap{ } <> > >
     <> > ) ,
```

which shows the situation immediately before reducing the first application of the meta-language λ-term, with two applicators set up on stack C, and with the two abstractions set up as closures on stack S (followed by the free variable `var{'w'}`). The next state transition step returns

```
LETREC
      ?
      ?
IN ( secd < <>
         < < 'u' clos{<> 'x' var{'x'}} > <> >
         < lambda{'v' apply{var{'u'} var{'v'}}} <> >
         < < var{'w'} <> > <> < ap{ } <> > <> > > ) ,
```

in which the application is actually reduced, with the instantiation of the variable `'u'` set up in stack E, with the abstraction `lambda{'v' ...}` set up on C, and with the remaining components of the enclosing application moved to the dump D.

Immediately before reducing the second redex of the meta-language term, the KIR program looks like this:

```
LETREC
      ?
      ?
IN ( secd < < clos{<> 'x' var{'x'}}
              < var{'w'} <> > >
         < < 'v' var{'w'} >
           < < 'u' clos{<> 'x' var{'x'}} > <> > >
         < ap{ } <> >
         < <> <> <> <> > > )  .
```

After having reduced the innermost meta-term redex, the program terminates as expected with

```
 < < < var{'w'} <> >
         < < 'v' var{'w'} >
           < < 'u' clos{<> 'x' var{'x'}} > <> > >
         <>
         < <> <> <> <> > > 'done' > .
```

It should be noted at this point that specifying the SECD machine in any other of the established functional languages, due to the availability of pattern matches, is equally straightforward and concise. What is missing, however, is the stepwise execution mode which enables the programmer to follow up on and study individual state transition steps.

6 Teaching Experiences

π–RED has been extensively used over the last five years in graduate lab courses complementing classroom teaching on various topics of functional programming and related computer architectures.

To get aquainted with the language KIR , particularly with the expressive power that comes with a full-fledged λ-calculus and with pattern matching, but also to familiarize with the syntax-directed editor, with the stepwise execution mode and with other π–RED controls, the courses always set out with some light exercises in functional programming. They emphazise recursive programming, the scoping of variables, simple symbolic computations, the use of higher-order functions, and rule-based programming.

The larger parts of these courses (which always included term projects) up to this point covered

- the implementation of several variants of SECD-machines for an applied λ-calculus (which includes δ-reduction rules for arithmetic| logic| relational and list processing functions) to study the effects of applicative order, normal order and lazy evaluation regimes on λ-terms, e.g., on different versions of the Y-combinator;
- an emulation of the G-machine, including a λ-lifter and an adaption of the G-compilation scheme as outlined in [John83] to the needs of compiling KIR to G-code;
- a compiler for KIR to C-code (which reqires some small C-routine to transform KIR output into C proper by removing quotation marks, KIR constructors, etc.);
- a language-independent parser which expects as actual parameters both the specification of a language-specific grammar and of the program to be parsed;
- the prototyping of compilers, de-compilers and interpreters for advanced versions of π–RED based on several variants of abstract stack machines.

It turned out that, after some short time (usually not exceeding two supervised sessions of four hours) of getting used to the intricacies of syntax-directed programming and of using stepwise execution as a means to debug and validate programs, students became highly productive in writing fairly large programs (more than 100 kBytes of KIR code).

With respect to correct program design it was considered very helpful that

- syntactically incorrect input is immediately rejected by the editor;
- incompletely specified constructs are completed by place-holders for terms to be inserted later on, which considerably facilitates top-down programming;

- in the process of debugging / validating programs, other than top level redexes can be selected for evaluation;
- the λ-calculus allows for symbolic computations with free variables, which is essential for doing reductions in selected subterms in which variables that are bound in larger contexts occur locally free.

Pattern matching in conjunction with higher-order functions as supported by the λ-calculus, after having developed some expertise in using these concepts, were effectively employed in writing very concise, comprehensible and well-structured programs. A good example in kind was the design of the abovementioned language-independent parser. It has been specified as a complex function of two parameters (whose body includes large sets of term rewrite rules implemented as pattern matches) which, when partially applied to just some syntax specification, reduces to the normal form of a unary function (λ-abstraction) which realizes a language-specific parser. In contrast to other implementations of functional languages, this function can be made visible as a high-level KIR program.

Criticism on π–RED in general and on the language KIR in particular primarily centered around lacking support for modularization. It made life sometimes difficult when it came to composing large programs and to understanding the effects of stepwise reductions, even when done in fairly local contexts. Occassionally, lacking interaction with a state was also felt to be a deficiency. Some difficulties were also encountered with the syntax-oriented editor, specifically with searching for and replacing multiple occurrences of certain subterms, and with some degree of randomness in introducing abbreviations, in order to make programs returning after shifts of reductions fit into the available window space. To overcome some of these problems, a more recent version of π–RED allows to switch to a line-oriented editing mode (vi) and to exercise a little more control over program layout.

We are presently also working on the implementation of KIR -compatible modules and of I/O operations.

References

[AbSus85] Abelson, H.; Sussmann, G.J.: *Structure and Interpretation of Computer Programs*, MIT Press, McGraw-Hill, New York, NY, 1985

[Back78] Backus, J.: *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, Vol. 21, No. 8, 1978, pp. 613–641

[Berk75] Berkling, K.J.: *Reduction Languages for Reduction Machines*, Proceedings of the 2nd Annual Symposium on Computer Architecture, 1975, ACM/IEEE 75CH0916-7C, pp. 133–140

[BeFe82b] Berkling, K.J.; Fehr, E.: *A Consistent Extension of the Lambda-Calculus as a Base for Functional Programming Languages*, Information and Control, Academic Press, Vol. 55, Nos. 1–3, October/November/December 1982

[Car83] Cardelli, L.; McQueen, D.: *The Functional Abstract Machine* The ML/LCF/HOPE Newsletter, AT&T, Bell Labs, Murray Hill NJ, 1983

[Brui72] DeBruijn, N.G.: *Lambda-Calculus Notation with Nameless Dummies. A Tool for Automatic Formula Manipulation with Application to the Church-Rosser-Theorem*, Indagationes Mathematicae, Vol. 34, 1972, pp. 381–392

[GaKl94] Gaertner, D., Kluge, W.E.: π–RED$^+$*: An Interactive Compiling Graph Reduction System for an Applied* λ*-Calculus*, to be published in JFP

[Gol94] Goldson, D.: *A Symbolic Calculator for Non-Strict Functional Programs* The Computer Journal, Vol. 37, No. 3, 1994, pp. 177 – 187

[HaMiTo88] Harper, R.; Milner, R.; Tofte, M.: *The Definition of Standard ML Version 3*, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1989

[HuWa88] Hudak, P.; Wadler, P. (Editors) et al.: *Report on the Functional Programming Language : Haskell*, Draft Proposed Standard, December 1988, Yale University

[John83] Johnsson, T.: *Efficient Compilation of Lazy Evaluation* SIGPLAN Compiler Construction Conference, Montreal Queb., 1984

[Klu94] Kluge, W.E.: *A User's Guide for the Reduction System* π–RED, Int. Report, No. 9419, Inst. f. Informatik, CAU KIel, 1994,

[Lan64] Landin, P.J.: *The Mechanical Evaluation of Expressions*, The Computer Journal, Vol. 6, No. 4, 1964, pp. 308–320

[Lavi87] Laville, A.: *Lazy Pattern Matching in the ML Language*, INRIA Rapporte de Recherche, No. 664, 1987

[Plas93] Plasmeijer, R., van Eekelen, M.: *Functional Programming and Parallel Graph Rewriting* Addison Wesley, 1993

[Pey92] Peyton–Jones, S.L.: *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine* Journal of Functional Programming, Vo. 2, No. 2, 1992, pp. 127 – 202

[Turn76] Turner, D.A.: *A New Implementation Technique for Applicative Languages* Software Practice and Experience, Vol. 9, No. 1,1979, pp. 31-49

[Turn86] Turner, D.A.: *An Overview of Miranda*, SIGPLAN Notices, Vol. 21, No. 12, 1986, pp. 158–166