A User's Guide for the Reduction System π–RED

Werner E. Kluge

June 28th 1994

Contents

1 Preface and Introduction

This document is primarily intended as a user's and programmer's guide for the reduction system π–RED . However, it may also serve as an introduction to functional programming, particularly to some flavors which are not supported by most other functional (or function-based) languages and systems such as HOPE, MIRANDA, SML, HASKELL or ID [BuMQSa80, Turn85, Turn86, HaMcQMi86, HaMiTo88, HuWa88, Nikh88].

π–RED is a complex software system of more than 2.5 MByte C-code (including some options and extensions that are of an experimental nature and therefore not part of the standard version described here) which can be installed on all UNIX-based systems[1]. It provides a sophisticated syntax-directed editor for alphanumerical terminals with an optional graphical frontend based on X-windows, and a complete run-time environment for functional languages featuring the reduction semantics of an applied λ-calculus [Chur41, Bar81, HiSe86]. Both programming and program execution are supported as fully interactive processes. Programs may be constructed 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 (or reduction) rules , of which the most important one is a full-fledged β-reduction[2] [Berk75, BeFe82b]. Programs may either be run to completion 'in one go', or be reduced in a step-by-step mode, with intermediate programs being displayed to the user for inspection and modifications. In particular, reductions may be carried out in any selected part of a program without creating side effects in other parts. Thus, π–RED is an excellent tool for efficient program design and for validating program correctness, and also for teaching basic programming (language) concepts. Due to the availability of a full-fledged λ-calculus, π–RED fully supports higher-order functions: functions may be freely applied to other functions or to themselves, and new functions resulting from the evaluation of (partial) function applications may be made visible to the user in high-level notation.

π–RED is available with implementations of the functional languages KiR (for Kiel Reduction Language) or OREL/2 (for Our Reduction Language/version 2). Both are strict, statically scoped and dynamically typed. In addition to the features that are common to almost all functional languages, they include APL-like structured data types [Iver62, Abr70, Schm86, SmBlKl91a] for efficient numerical computations, and sophisticated pattern matching facilitaties which make them ideal tools for prototyping other languages and their run-time environments. Since differences between both languages are largely syntactical,

[1] An installation guide is included in appendix 3.

[2] β-reduction essentially defines the substitution of actual for formal function parameters, including the orderly resolution of naming conflicts that may arise between free variable occurrences in argument terms and variables bound in function terms.

we will introduce in this document only KiR . This choice is based on more extensive experience with this language in writing large application programs and on its slightly more amenable syntax. For a description and formal specification of OREL/2 (whose syntax closely resembles that of SML and MIRANDA), the interested reader is referred to [PlSc90].

In its present form, π–RED is the result of a complete overhaul of the λ-calculus-based reduction concept and the string reduction machinery proposed by Berkling as early as 1975 [Berk75, Berk78], which was first implemented by Hommes as a software simulator [Hom77, Hom80] and then by Kluge as an experimental hardware machine [Klu79, KlSc80]. The improvements primarily concern the inclusion of more powerful language constructs, improved editing facilities, and efficient run-time support by means of graph reduction techniques [SmBlKl91a, Zim91], fine-tuned parameter-passing mechanisms and garbage collection schemes.

Currently there are two versions of π–RED available, of which

- π–RED* is a high-level interpreter for graph representations of λ-terms (which internally are converted into supercombinators) [SmBlKl91a];
- π–RED$^+$ performs graph reduction based on compiled code for user-defined functions [Gaer91], and thus runs by a factor of about 2 to 8 faster than π–RED*, depending on the nature of the application programs.

In this document we will merely describe what you can see of π–RED at the user interface, typically an alphanumerical display terminal or a terminal emulation on a graphical window, and how you have to work with it in order to edit and execute KiR programs. As an option, there is also available an X-window-based frontend. It provides a number of menues which facilitate working with the editor and controlling the system.

Setting out with a small language kernel, the syntactical constructs of KiR and their semantics, i.e., the reduction rules by which they are being transformed, will be introduced in the order of increasing complexity and sophistication, as we proceed. Rather than giving formal definitions, we will largely work with selected example programs. In doing this, we will also show how the editing facilities may be used to recursively construct large programs from smaller pieces and how to validate programs (or parts thereof) during the design process by stepwise execution and inspection of intermediate states of computations.

A formal definition of the syntax of KiR follows in an appendix at the end of this document. A second appendix gives an overview over the π–RED editor functions and the command language, and a third appendix briefly describes how π–RED can be installed on your system.

π–RED has been developed jointly at the University of Kiel and at the GMD in St. Augustin, with major contributions by Claus Assmann, Harald Bloedorn, Ulrich Diedrichsen, Dietmar Gaertner, Heinz Mevissen, Joerg Schepers, Heinz Schluetter, Claudia Schmittgen and Ralf Zimmer.

2 A Kernel of KiR

Non-trivial functional programs are expressions composed of (mutually recursive) function definitions and function applications [AbSus85, McQMi87, BiWa88]. They are pure algorithms which, in contrast to conventional imperative programs, just specify what is to be computed rather than including detailled schedules of how to actually execute them on some underlying machinery [Back78]. The elementary operations to be performed are only partially ordered, merely reflecting the logical structure of the appplication problem. There is no notion of a memory either: as in algebra, variables are 'unknowns' which may serve as placeholders for other expressions to be substituted for them, but they do not represent changeable values.

A functional program written in KiR typically has the syntactical form

```
def
   ⋮
   F[X_1,...,X_n]  =  F_expr
   ⋮
   H[Y_1,...,Y_m]  =  H_expr
   ⋮
in S_expr .
```

This DEFINE-construct specifies a set of recursive functions in terms of which the value of the so-called goal-expression `S_expr` is to be computed. We refer to the left-hand sides of the function definitions as function headers and to the right-hand sides as function body expressions (or function bodies for short).

For instance, the function header `F[X_1, ..., X_n]` declares `F` to be a function of `n` variables (or formal parameters) `X_1, ..., X_n`. The function body expression `F_expr` specifies an algorithm for the computation of function values of `F` in terms of

- the variables `X_1, ..., X_n` and `F` (in which case the function is recursive);
- applications of subfunctions of `F`;
- applications of functions declared at the same level as `F`, e.g., the function `H`.

These DEFINE-constructs may recursively appear as function body expressions to define local subfunctions. For instance, `F_expr` may be specified as

```
def
    ⋮
    G[V_1, ..., V_m]  =  G_expr
    ⋮
in F_expr' ,
```

where the function `G` and all other functions defined at this and lower levels are subfunctions of `F`. All variables defined in a particular function are also defined in its subfunctions. All function names (or function identifiers) specified under a particular DEFINE-construct should be unique[3].

A more appropriate notion for the status of a variable is that of its binding scope. With respect to a program specified in the above form, scopes may be roughly defined as follows:

- the formal parameters `X_1, ..., X_n` are bound in the function `F` but free in its subfunctions, e.g., in the function `G`;
- the formal parameters `V_1, ..., V_m` are bound in `G` but, again, free in the subfunctions of `G`;
- function identifiers are mutually recursively bound in all functions defined at the same hierarchical level; they are also bound in the goal expression: `G` is bound in `F_expr'`, and `F` and `H` are bound in `S_expr`.

We will henceforth refer to variables which are free in subfunctions but bound higher up as being relatively free.

In addition to functions that are given names, KIR also admits non-recursive nameless (or anonymous) functions of the form

```
sub [X_1, ..., X_n] in Expr   ,
```

where `X_1, ..., X_n` again denote formal parameters and `Expr` denotes the function body expression.

Other expressions may be recursively constructed from

- function applications, which generally are denoted as

  ```
  ap  Func
  to  [ Arg_1, ... , Arg_n]
  ```

  or alternatively as

  ```
  Func[Arg_1, ..., Arg_r]   ,
  ```

[3] In fact, π–RED allows that some or all functions be named identically. Applied occurrences of function identifiers must then be preceded by backslashes whose numbers uniquely identify the positions of the respective function definitions in the DEFINE constructs.

where `Func` may be (any legitimate KiR expression which is expected to evaluate to) the name of a function defined by an equation, a nameless function (in which case only the former representation will be accepted), or a symbol representing a primitive function, and `Arg_1, ..., Arg_r` are argument expressions[4];

- LET clauses of the general form

```
let
        ⋮
        X = X_expr
        ⋮
        Y = Y_expr
        ⋮
in  Expr
```

  which specify the (values of) `X_expr` and `Y_expr` as substitutes for free occurrences of the variables `X` and `Y`, respectively, in `Expr`;

- IF-THEN-ELSE clauses of the form

```
if    Pred_expr
then  True_expr
else  False_expr
```

  of which `Pred_expr`, `True_expr` and `False_expr` are also referred to as the predicate, the consequence and the alternative, respectively.

- lists or sequences of expressions, denoted as

```
<Expr_1, ..., Expr_n>
```

  as a means for representing structured objects.

The set of primitive functions of KiR includes the familiar binary arithmetic, logical and relational operators, applications of which are usually denoted in (or by π–RED turned into) parenthesized form as[5]

```
(Arg_2 Op Arg_1)   .
```

[4] π–RED sometimes exchanges, in the course of evaluating a program, these representations against each other as a matter of convenience: it usually prefers the former if `Func` itself is an application, a nameless function, or a composite expression, and the latter if it is a function name or a variable.

[5] Note that `Op` need not necessarily be a binary operator right away: it may be any legitimate KiR expression which eventually reduces to one.

There are also a number of primitive structuring and predicate functions applicable to lists which we will introduce later on.

As a first example, here is a simple KIR program which computes the greatest common divisor of two integer numbers:

```
def
   Gcd[X,Y]     =  if    (X eq Y)
                   then  X
                   else  if    (X gt Y)
                         then  Gcd[Y,Modulo[X,Y]]
                         else  Gcd[X,Modulo[Y,X]]
   Modulo[X,Y]  =  if    (X le Y)
                   then  X
                   else  Modulo[(X - Y),Y]
in Gcd[4,6]
```

It consists of two recursive functions, of which `Gcd[X,Y]` is directly called in the goal-expression to return the greatest common divisor of actual integer values substituted for `X` and `Y` (which in this particular example are `4` and `6`, respectively), and `Modulo[X,Y]` is called as a subfunction of `Gcd[X,Y]` to compute the remainder of `X` after division by `Y`.

As a convention, KIR uses lower case letters to denote keywords such as `def`, `if`, `then`, `else`, `in` etc., whereas all identifiers introduced by the programmer (function names and formal parameters) start with upper case letters. As we will see in the next section, this is nothing that you need to worry about for this is taken care of by the π–RED editor[6].

It may be noted that KIR programs (as programs of any other functional language, for that matter) exhibit the same overall structure as imperative programs. The set of function definitions replaces the set of procedure declarations, and the goal expression replaces the statement part. This structure recursively extends into all local definitions (or declarations). LET-expressions combine local variable declarations with single assignments of values (of expressions).

Since KIR is an untyped language, its programs do not include explicit type declarations. Type compatibility checks between primitive functions and the operands to which they are actually applied are dynamically performed by the run-time system of π–RED based on type-tags that are carried along with all objects of a program.

With this basic knowledge about KIR at hand, we are now ready to have a closer look at the workings of π–RED .

[6]However, be warned that if you accidently type any of the keywords with upper case letters, say `If`, `Then`, etc., they are taken as variables.

3 Editing and Executing Simple Programs on π–RED

In this section you will learn the first steps necessary to edit and execute a simple functional program on π–RED . A good example to start with is the program given in the preceding section which computes the greatest common divisor of two integer numbers.

If properly installed on your UNIX system[7], π–RED may be called under the appropriate directory by typing the command

< `unix_prompter` >: `reduma`.

The system will respond by producing on your screen (or window) the user interface of π–RED which has the basic format shown in fig. 1. It consists of a

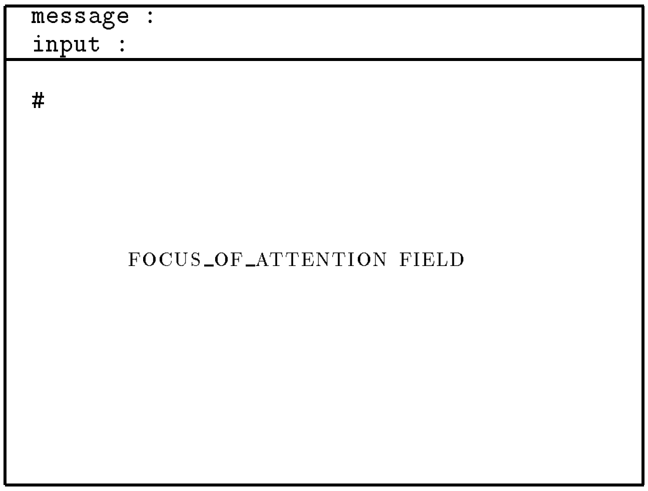

Figure 1: Layout of the π–RED User Interface

small field at the top, separated by a horizontal line from a large field which stretches over the remainder of the screen at the bottom. The field at the top is divided into a line each for messages, in which you will initially see

RED 2.3 `Function – Version` ,

and for input. The bottom field will be initialized with the symbol #, and you will also find the cursor in the position of the #. Roughly speaking, the contents

[7] See also appendix 3.

of this field represent the KIR expression that is to be actually processed by π–RED . It is therefore referred to as the focus_of_attention field (or FA field for short). The # symbol generally stands for an empty expression (or an empty syntactical position), and # as the only item in this field simply indicates that no expression is as yet loaded.

π–RED is now set up and ready for editing, which is a strictly syntax-directed process. This is to say that you always work with complete syntactical constructs of KIR in which either empty syntactical positions need to be replaced with other constructs, or the constructs must themselves be placed into specific syntactical positions of enclosing constructs. The former corresponds to designing programs top-down, and the latter corresponds to designing them bottom-up. Both directions may be used in any order of convenience. In supporting this programming style, the editor makes use of the fact that specific keywords of KIR , so-called constructors, form the root nodes of certain syntactical constructs, and that some of these constructs may even require expressions of a particular type (e.g., variables, patterns etc.) in some of their components.

To see how this works, you may start typing on your keyboard the fragment of a simple KIR expression, say, just the keyword `if`. The `if` is immediately echoed in the input line, and the message line fills with `<expr|def>`, indicating that the input line may legally contain a KIR expression or a (function) definition. If you now hit the return (or enter) key on your keyboard, both the message and the input line will be cleared, and in the FA field appears a syntactically complete `if_then_else` construct with as yet uninstantiated (or empty) component expressions, i.e., with # symbols in the respective syntactical positions. The cursor has now been moved to the topmost # position, expecting it to be replaced next by a legal KIR expression, which in this particular case should be a predicate expression that evaluates to a Boolean constant.

```
if #
then #
else #
```

You can now use this structure to construct the function body of either of the two recursive functions that make up the greatest common divisor program; so let's do the function `Modulo` first. Under the current cursor position you may therefore type

```
(x le y
```

which again echoes on the input line, with just `<expr>` on the message line. Note that the input has an opening but no closing parenthesis, and that both variables are printed as lower case letters. Upon hitting the return key, you get inserted into the cursor position a fully parenthesized expression with the variables now printed as upper case letters. In doing this, the editor takes the opening parenthesis as the constructor of an expression which, by convention, has three components and thus inserts the closing parenthesis immediately following the third component. Furthermore, since it has neither `x` nor `y` on the list of keywords of KIR , it takes them as variables and turns them into upper case letters. The string `le`, however, represents a primitive function symbol, i.e., it is a keyword, and therefore left as it is.

```
if ( X le Y )
then #
else #
```

The cursor has now moved to the next empty position, and you may simply input `x` to get

```
if ( X le Y )
then X
else #
```

Finally, you may fill out the `else` part of the clause, which can be done in one step by typing

$$\texttt{modulo}[(\texttt{x} - \texttt{y}, \texttt{y}.$$

Note that you may forget about both the closing parenthesis following the arithmetic expression in the first argument position of `modulo` and also the closing `]` following the second (and last) argument position. Upon hitting the return key you will see that both are inserted into the correct positions by the editor. Again, all character strings that cannot be identified as keywords of KIR are turned into variables, i.e., their leading letters are converted to upper case.

```
if ( X le Y )
then X
else Modulo [ ( X - Y ), Y ]
```

Looking at this expression, you should realize that something strange has happened. At this point in the program design, `modulo` is just a name for a function that is not yet defined, so its arity is not yet known. Also, a function in KIR may be applied to any number of arguments irrespective of its own arity. Thus there is no way for the editor to figure out how many arguments need actually be supplied. The closing parenthesis `]` is therefore inserted on a purely speculative basis, assuming that the current input line contains a complete argument list. However, as you will see in just a moment, there are means to add further arguments to this list or to delete some of them, if so desired.

So far, you have just followed a top-down design path by filling with concrete expressions empty positions of a syntactical construct. Since no further `#` symbols are left in the FA field expression, you will have to go a step bottom-up from here to complete the program.

Before doing this, you need to know something about the significance of the cursor position and the way the cursor may be moved around. The cursor is always positioned either on a KiR constructor, e.g., a keyword or a parenthesis `"("` which opens an application in infix notation, or on an atom, i.e., a variable, a constant, a primitive function symbol, or an empty expression `#`. Whatever expression (or fragment thereof) you type on the keyboard first shows up in the input line and, upon pressing the return key, ends up in the cursor position, i.e., it overwrites there either an atom or an entire subexpression held together by a constructor symbol. So, what really matters in the editing process is not necessarily the full expression that you see in the FA field but it is always what we will from now on refer to as the cursor expression. You may check this out at the risk of destroying your current cursor expression (unless you have already done so by accident) or, for the moment, just believe it.

In the `if_then_else` clause you have developed so far, the cursor happens to be positioned on the variable `Modulo` for it is the leading symbol of the expression you have inserted last. In order to make the entire clause the subject of further editing operations, the cursor needs to be moved up to the `if`.

Since the π–RED user interface emulates an alphanumerical terminal, the cursor movement may only be controlled by the respective keys on your keyboard, of which you usually have available one each pointing up, down, left and right[8].

As a default option you have at hand an expression-oriented cursor movement. With each of the cursor keys you may traverse in a particular way the syntax tree of a KiR expression. Here is how it works:

Starting from the current cursor position, the

- `up` key moves the cursor to the hierarchically higher constructor (or from a son node to its immediate father node);
- `down` key moves the cursor from left to right and round robin through all nodes that are tied to the same immediate father node;
- `right` key lets the cursor traverse the syntax tree in pre-order (i.e., recursively top-down and left-right);
- `left` key lets the cursor traverse the syntax tree in post-order (i.e., recursively right-left and bottom-up)[9].

[8] A mouse, if available, doesn't do anything for you other than selecting the window you want to work with on a workstation.

[9] If your keyboard includes a home key, it moves the cursor from anywhere within an expression directly to the topmost constructor.

Now you can play with the cursor keys a little more systematically and thereby find out which components of an expression are constructors, which are atoms, and which are just blind syntactical sugar. For instance, whichever way you are moving about the current FA expression, you will never get the cursor on the `else` or on any of the parentheses `),[,]`. They just let the expression look a little more structured and readable, but they are of no relevance syntactically. You may also note that, contrary to what we said earlier about variables being atoms, the identifier `Modulo` seems to be taken as a constructor. This is due to the fact that in this particular high-level representation of an application there is no explicit constructor (which would be an applicator); so the function symbol is taken for it instead.

Alternatively, the expressions may be traversed in a line-oriented mode (as every conventional editor does). This mode may be enabled with a special control command which will be introduced later on.

After this little excursion into cursor movement, you may now continue with the construction of the greatest common divisor program.

The current FA expression must become the right-hand side of a function definition for `Modulo[X,Y]`, which must in turn be embedded in a `def` construct. All you need to do to accomplish this is to maneouver the cursor to the topmost constructor (which is the `if`) and to type as input something like

$$\texttt{Modulo}[\,] = \%$$

if, for the time being, you don't want to think about the formal parameters of the function. The important part of this input is the percentage sign `%`. It acts as a placeholder for which the current cursor expression will be substituted upon entering the input as the new cursor expression. However, what you actually get when hitting the return key may not be exactly what you would expect: the editor returns as the new cursor expression a `def` construct comprising a function definition for `Modulo[ ]` (i.e., a function which as of now is without parameters) and, surprisingly enough, as a goal expression another copy of the body of this function.

```
def
  Modulo [   ] = if ( X le Y )
                 then X
                 else Modulo [ ( X - Y ), Y ]
in if ( X le Y )
   then X
   else Modulo [ ( X - Y ), Y ]
```

Being faced with a goal expression that you didn't ask for has to do with implementation details of the editor which you don't need to worry about. To proceed with your editing, it really doesn't matter whether or not you have in this place an expression other than the empty symbol #. In either case, you'll have to make this syntactical position the cursor expression and type as input (a fragment of) the expression you want to be there, which in our example would be

gcd [6, 4 .

If you then move the cursor into the position between the parentheses [ ] in the function header, type

x, y

in the input line (which brings up <var_list> in the message line), and hit the return key, you get as the new cursor expression:

```
def
  Modulo [ X, Y ] = if ( X le Y )
                    then X
                    else Modulo [ ( X - Y ), Y ]
in Gcd [ 6, 4 ]
```

Note that the list of formal parameters may be expanded by moving the cursor to the parameter position prior to or after which you wish to insert the new parameter, and by entering the input line

z, % or %, z ,

respectively (where z is taken as the identifier that is to be added). Conversely, a parameter in cursor position may be removed from the list by hitting the delete key on your keyboard. You are free to test both operations on your current FA expression without wrecking it. In the same way you may also expand or shrink the list of arguments of a function application for it features the same syntactical structure as a function header.

Let's now return to the business of completing the greatest common divisor program. What remains to be done is to include the second function definition, the one for Gcd [X,Y], into the def construct. To do so, you first need to move the cursor to the identifier Modulo which is the leading symbol in the function header of the existing definition. Whenever then you start typing anything in the input line, the message line responds with <var| fdec>, meaning that you may replace either just the identifier Modulo or the entire function definition with another expression of the same kind, neither of which you actually want to do. The third possibility is to include in the input a reference % to this cursor expression and thus embedd it bottom-up in the desired context.

When using as input

%, gcd[ ] = ,

the editor will add the fragment of the definition for `Gcd` after that for `Modulo`[10]. The resulting cursor expression looks like this:

```
def
  Modulo [ X, Y ] = if ( X le Y )
                    then X
                    else Modulo [ ( X - Y ), Y ],
  Gcd [   ] = #
in Gcd [ 6, 4 ]
```

Now you may go ahead to fill out the missing argument list and the missing function body of `Gcd` in just about the same way as for the function `Modulo`. If everything is done, you should get the cursor expression

```
def
  Modulo [ X, Y ] = if ( X le Y )
                    then X
                    else Modulo [ ( X - Y ), Y ],
  Gcd [ X, Y ] = if ( X eq Y )
                 then X
                 else if ( X gt Y )
                      then Gcd [ Y, Modulo [ X, Y ] ]
                      else Gcd [ X, Modulo [ Y, X ] ]
in Gcd [ 6, 4 ]
```

which is ready for execution.

Of course, you may edit this program in any other way you wish. For instance, you could start with the input `def` which produces in the FA field a `def` construct with an empty position each for a function definition and a goal expression. From there you can design the program strictly top-down by recursive refinement. Once you are more familiar with KIR , you will also know that there is a primitive function `mod` available which renders the defined function `Modulo` superfluous.

Before executing this program (which irrevocably destroys the original program text), you better store it away on a permanent file since you may wish to use it several times. In order to do so, it is important to get acquainted with a few control functions of π–RED that are available as logical function keys ranging from F1 to F16. Some or all of these logical keys may be assigned, as part of the π–RED initialization procedure, to physical function keys that are available

[10] As of now, there is now way of doing it the other way around, i.e., to put the definition of `Gcd` before that of `Modulo`.

on your particular workstation or terminal. If your keyboard doesn't have that many keys or some of them already have other system functions assigned to them, you may either re-define the use of the available keys according to your preference (you will be told later on in section 4.5 how this is to be done) or instead type `:n` (where n is the number of the logical function key).

Most of the logical function keys, in one way or another, relate to the particularities of the reduction semantics supported by π–RED . As pointed out before, executing a program is realized as a sequence of meaning-preserving program transformations, of which each produces (the internal representation of) a legitimate KiR program. In order to make intermediate programs of interest visible to the user for inspection, π–RED provides the means to control, by means of a so-called reduction counter, the number of transformation (or reduction) steps after which it ought to stop, and to display the program expression reached at that point. If the program is to run to completion in one shot, then this number must simply be chosen large enough to allow for a reduction sequence of the expected length.

Moreover, there are several means to store programs or program parts temporarily on various back-up buffers, or permanently on an editor file system where they survive a π–RED session. Temporary storage enables you to return quickly to initial programs or to previous states of execution. This feature is particularly helpful during the design and validation phase of a program.

The operations that control the number of reduction steps are assigned to the function keys[11] as follows:

F1 performs at most one reduction step on the current cursor expression and returns the resulting expression in its place; [12];

F9 performs in one go at most 1000 reduction steps on the cursor expression and, again, returns the resulting expression in its place;

F8 gets you in a general command mode in which you are asked on the message line to enter a particular command on the input line. In this mode you may type `r n` (note that there must be a blank between the `r` (which stands for reductions) and the `n`) and then hit the return key to have at most some n reductions performed on the cursor expression[13]. The phrase '..at most..' refers to the fact that the reduction of the cursor expression may terminate with a constant expression before exhausting the number of steps that you allowed for. The number of reduction steps that has actually been performed always shows up in the message line upon returning the resulting expression.

[11] Whenever we refer, fron now on, to function keys, we mean the logical keys of π–RED .

[12] The number of reductions performed under F1 can be re-defined by changing the respective entry in the initialization file for your π–RED version (see also appendix 2)

[13] The upper limit for n is basically given by the integer format of the machine on which you have π–RED installed.

The following function keys are available for storing and retrieving cursor expressions:

F10 gets you into an output mode in which you are asked on the message line to enter the name of an editor file into which the cursor expression is to be written. You simply type the desired name (legitimate file names depend on the underlying operating systems; usually all character strings starting with a letter will do) and hit the return key. If the file name is new, the system will respond in the message line with

`wrote 1 expression to filename.ed` ,

if a file with this name already exists, it will ask you to verify that you want it to be overwritten.

F2 gets you into an input mode in which you are asked on the message line to enter the name of an editor file. If the file exists, the system responds on the message line with

`1 expression read from filename.ed`

and replaces with this expression the actual cursor expression. Otherwise it will tell you that a file with the given name can not be opened.

F14 or F15 writes the cursor expression in a first or a second auxiliary buffer, respectively;

F6 or F7 replaces the cursor expression with an expression read from a first or second auxiliary buffer, respectively;

F13 writes the cursor expression in another back-up buffer, which is implicitely done whenever you start a reduction sequence;

F5 exchanges the cursor expression with the expression held in the back-up buffer[14]

Note that if you accidently get into the wrong mode our your input cannot be accepted under the current mode for reasons of which you will be appropriately informed on the message line, you may get out of this situation by first clearing the input line with the `delete` key and then hitting the return key, which will also clear the message line.

After you've stored your Gcd program away on a permanent file, say gcd, you are all set to execute the program.

[14]Note that each of these buffers can hold at most one expression: whenever you write in succession two or more expressions in a buffer, only the last expression can be retrieved, all others are lost.

If you are confident about the program's correctness, you may press the function key F9, thus allowing for at most 1000 reduction steps, and you will immediately get as a new cursor expression the correct value 2. On the message line you will also find that it took 27 reduction steps to get there.

You may now reload the program from the `gcd` file, and run it again, e.g., now in a stepwise mode. To do this, you just push the function key F1 several times and look at what comes up in the FA field.

The expression returned after the first step looks like this:

```
if ( 6 eq 4 )
then 6
else if ( 6 gt 4 )
     then ap def
               Modulo [ X , Y ] = if ( X le Y )
                                  then X
                                  else Modulo [ ( X - Y ) , Y ] ,
               Gcd [ X , Y ] = if ( X eq Y )
                               then X
                               else if ( X gt Y )
                                    then Gcd [ Y , Modulo [ X , Y ] ]
                                    else Gcd [ X , Modulo [ Y , X ] ]
             in Gcd
          to [ 4 , ap def
                        Modulo [ X , Y ] = if ( X le Y )
                                           then X
                                           else Modulo [ ( X - Y ) , Y ]
                      in Modulo
                   to [ 6 , 4 ] ]
     else ap def
               Modulo [ X , Y ] = if ( X le Y )
                                  then X
                                  else Modulo [ ( X - Y ) , Y ] ,
               Gcd [ X , Y ] = if ( X eq Y )
                               then X
                               else if ( X gt Y )
                                    then Gcd [ Y , Modulo [ X , Y ] ]
                                    else Gcd [ X , Modulo [ Y , X ] ]
             in Gcd
          to [ 6 , ap def
                        Modulo [ X , Y ] = if ( X le Y )
                                           then X
                                           else Modulo [ ( X - Y ) , Y ]
                      in Modulo
                   to [ 4 , 6 ] ]
```

What has happened here is the following: The application `Gcd[6,4]` in the goal expression of the original program has been expanded by the function body of `Gcd`, i.e.,by the right-hand side of its definition, which consists of two nested `if_then_else` clauses. Moreover, in this body all occurrences of the formal parameters `X` and `Y` have been substituted by the actual parameters (or argument values) `6` and `4`, respectively, and – this is the most interesting part – all applications of

- the function identifier `Gcd` are embedded in applications of copies of the entire original `def` construct,
- the function identifier `Modulo` are embedded in applications of copies of a `def` construct from which the definition of `Gcd` is deleted.

Both applications are returned by π–RED in the syntactical form

`ap def` ... `in Gcd` | `Modulo to`[..., ...]

where `ap` is the explicit applicator introduced in chapter 2. The original `def` construct has disappeared.

If you take a close look, you will also note that the `def` constructs thus distributed over the original goal expression have inserted in their own goal term positions just the identifiers of the functions that need to be applied to the arguments in the enclosing applications. Identifiers in this position merely act as selectors for a function from the set of definitions. Moreover, this set is stripped down to just the functions that are actually referenced from within the goal expressions: if you have `Gcd` there, both `Gcd` and `Modulo` are required since the latter is referenced from within the former; if you have `Modulo` there, the definition for `Gcd` is dropped since `Modulo` recursively just calls itself.

The next four reduction steps on the cursor expression are quite simple: they evaluate the two nested `if_then_else` clauses and return the application shown on top of the next page. Here you have in fact an application of the function `Gcd` to a first argument value `4` and to a second argument value which is to be computed from an application of `Modulo` to the arguments `6` and `4`. To understand what happens next, you need to know that π–RED realizes a so-called applicative order (or call_by_value) reduction regime: before a function is applied to its arguments, the arguments themselves are being evaluated first. So, in our particular situation, the reduction process will continue with the application of `Modulo` in the argument position of `Gcd`.

The expression that returns after one more reduction step therefore has this application replaced by the instantiated body of `Modulo` which, in another three steps, reduces to an application of `Modulo` to the arguments `2` and `4`. The value of this is obviously `2`, which you reach after three more steps.

```
ap def
     Modulo [ X , Y ] = if ( X le Y )
                        then X
                        else Modulo [ ( X - Y ) , Y ] ,
     Gcd [ X , Y ] = if ( X eq Y )
                     then X
                     else if ( X gt Y )
                          then Gcd [ Y , Modulo [ X , Y ] ]
                          else Gcd [ X , Modulo [ Y , X ] ]
   in Gcd
to [ 4 , ap def
              Modulo [ X , Y ] = if ( X le Y )
                                 then X
                                 else Modulo [ ( X - Y ) , Y ]
            in Modulo
         to [ 6 , 4 ] ]
```

Now you have completed a full cycle through the reduction of an application of the function `Gcd`. As a result you get, in slightly different notation, the original program which, however, has its goal expression replaced by `Gcd[4,2]`. The difference relates to the representation of this application: the entire `def` construct with `Gcd` as a function selector in its goal expression is now applied, by means of an explicit applicator, to the arguments `4` and `2`.

```
ap def
     Modulo [ X , Y ] = if ( X le Y )
                        then X
                        else Modulo [ ( X - Y ) , Y ] ,
     Gcd [ X , Y ] = if ( X eq Y )
                     then X
                     else if ( X gt Y )
                          then Gcd [ Y , Modulo [ X , Y ] ]
                          else Gcd [ X , Modulo [ Y , X ] ]
   in Gcd
to [ 4 , 2 ]
```

Knowing that it takes five reduction steps to get from an application of `Gcd` to arguments, of which the first is greater than the second, to the point where its nested `if_then_else` clauses are evaluated, you may simply skip all the tedious intermediate steps by first entering the command mode with the function key F8 and then typing into the input line `r 5`. Upon hitting the return key, you will get the expression:

```
ap def
     Modulo [ X , Y ] = if ( X le Y )
                          then X
                          else Modulo [ ( X - Y ) , Y ] ,
     Gcd [ X , Y ] = if ( X eq Y )
                       then X
                       else if ( X gt Y )
                            then Gcd [ Y , Modulo [ X , Y ] ]
                            else Gcd [ X , Modulo [ Y , X ] ]
   in Gcd
to [ 2 , ap def
              Modulo [ X , Y ] = if ( X le Y )
                                   then X
                                   else Modulo [ ( X - Y ) , Y ]
            in Modulo
         to [ 4 , 2 ] ]
```

You may now wish to have evaluated the application of `Modulo` in the second argument position of the outermost application without bothering about the necessary number of reduction steps on the one hand (which depends on the actual number of recursive calls for `Modulo`), and without having the reduction either stop short of or proceed beyond that point on the other hand. This is quite easy to accomplish since functional programs are referentially transparent, i.e., the result of reducing a subexpression is invariant against the context in which this is done. So, all you need to do is to first make the application of `Modulo` the new cursor expression by moving the cursor down under the applicator and then hit the function key F9 to trigger a reduction sequence of at most 1000 steps (which is way more than necessary to reach a function value). Without effecting any changes elsewhere, π–RED will return, after only seven reduction steps, the correct value `2` in this argument position and thus complete the second call of `Gcd`,as is shown on the next page.

From here on it is quite easy to figure out how many more steps are required to terminate the program, given that both arguments of `Gcd` are now equal. It takes one step to instantiate the function `Gcd` one more time, one step to evaluate to `true` the predicate of the outermost `if_then_else` clause in the body of `Gcd`, and one last step to select as the final result for the greatest common divisor the value `2` in the consequence of that clause. But: before you initiate these steps, don't forget to move the cursor back to the topmost applicator of the expression displayed in the FA field, otherwise nothing will happen.

```
ap def
     Modulo [ X , Y ] = if ( X le Y )
                        then X
                        else Modulo [ ( X - Y ) , Y ] ,
     Gcd [ X , Y ] = if ( X eq Y )
                     then X
                     else if ( X gt Y )
                          then Gcd [ Y , Modulo [ X , Y ] ]
                          else Gcd [ X , Modulo [ Y , X ] ]
   in Gcd
to [ 2 , 2 ]
```

A session with π–RED may be ended either the brutal way, i.e., by typing `CTR-c` or `CTR-z`, or by entering the command mode and then typing `exit`.

Now you have a pretty good idea of how π–RED basically works, but there is lots more to learn about both the language KiR and the handling of the editor.

4 More about the Editor and the π–RED System

In this chapter we will address a few more things you ought to know about the π–RED editor and the system by which it is backed up. They'll make life a little easier for you when it comes to editing and running in a step-by-step mode large programs, and to filing your programs.

There is also a help file available which you may consult if you get lost. When pressing the function key F4, you will get an overview over nine help keys which will guide you further down into several groups of editor and system functions. In addition to that you will also find a complete definition of the syntax of KIR and, in short form, a description of how to edit and execute KIR programs.

4.1 The Layout of KIR Programs on your Display

The syntax-directed editor of π–RED works with a screen of fixed size on which a given program expression is to be laid out. Unlike line-oriented editors, it does not include a scrolling mechanism. Thus, if the full layout of an expression expands beyond the screen size, the editor must introduce abbreviations in various syntactical positions in order to make it fit.

The greatest common divisor program is a very good-natured example in this sense. Since both functions used in the program are tail-end recursive, it never expands beyond one function call and thus almost (!) always completely fits onto the screen. However, depending on its actual size you may or may not have noticed in the expanded function body of Gcd that is being returned after the first reduction step syntactical positions here and there which are filled with question marks ?. They are abbreviations of the subexpressions which the editor was unable to lay out in the remainig space.

You can see more of these question marks if the programs are getting bigger in size or if they are specified in terms of recursive functions that are not tail-end and thus recursively expand on the screen when executing them.

A good example in kind is the factorial function. Initially, it fits conveniently into the FA field, with lots of space left.

```
def
  F [ N ] = if ( N gt 1 )
            then ( N * F [ ( N - 1 ) ] )
            else 1
in F [ 20 ]
```

When reducing this program step-by-step you will see that it builds up nested multiplications from left to right, with the application of the function definition to the decrementing argument shifting away in the same direction.

```
( 20 * ( 19 * ( 18 * ( 17 * ap def
                                 F [ N ] = if ( N gt 1 )
                                           then ( N * F [ ( N - 1 ) ] )
                                           else 1
                               in F
                            to [ 16 ] ) ) ) )
```

Rather than letting it spill over on the right-hand side of the screen, the editor lets this application 'disappear' under a question mark so that a full top-level expression (i.e., the outermost syntactical structure) remains visible.

```
( 20 * ( 19 * ( 18 * ( 17 * ( 16 * ( 15 * ( 14 * ( 13 * ? ) ) ) ) ) ) ) )
```

As you proceed with further reductions, the expression continues to expand under the question mark (provided the argument value is large enough), but what you can see of it on the screen remains as it is.

In order to display what is hidden behind a question mark you simply need to move the cursor either directly under it or to a constructor that is close to it, say the innermost opening parenthesis by which it is preceded. If you now press the function key F3, the FA field fills with the new cursor expression (i.e., the cursor re-appears in the left upper corner of the screen), and the surrounding context disappears. This expression may contain more question marks behind which you may look by proceeding in the same way. In doing this, you recursively climb down the syntax tree of your (intermediate) program to the sub-expression that you are interested in. As explained before, you may perform reductions in this part of the program without worrying about undesirable side-effects in other parts: there simply aren't any.

The easiest way of climbing up in the syntax tree and thus returning to larger contexts is to move the cursor to the topmost constructor of the expression that is currently in the FA field (if it is not already there) and to hit the up key. Every time you do this, you move upwards to the next father node of the syntax tree as usual, but you also get displayed on your screen the full expression of which this node is the topmost constructor, possibly with increasingly larger sub-expressions again disappearing behind question marks as you move on in this direction. If you wish to return in one step to the top of the entire program, simply type on your keyboard `CTR-n`. There is no other way of moving upwards, e.g., by reversing the steps you have taken downwards.

Another problem for which a remedy is necessary arises in connection with large sets of fairly complex function definitions, possibly nested over several levels.

These definitions may easily take up the entire space that is available on the screen, in many cases requiring lots of abbreviations in the form of question marks. This situation is getting worse when performing just a few reduction steps after which an expanded intermediate program is to be displayed. It will likely contain, in even more abbreviated form, several basically identical copies of `def` constructs which have been (recursively) substituted for function identifiers in the original goal expression. All abbreviations that occur in such a program representation are introduced by the editor, mainly based on the screen space that is left in a particular situation, not with respect to components that are of primary interest, say the changeable program parts where reductions are actually taking place. Thus, valuable screen space is generally wasted for things that you may not be interested in, whereas the important parts disappear behind question marks.

In order to have a little more control over what is being displayed and how it is laid out, the editor provides under the command mode F8 a few commands that introduce abbreviations in dedicated places (syntactical positions) of a program. They include

`abb_def` which leaves the topmost level of function definitions intact but abbreviates all others;

`abb_idef` which abbreviates only function definitions that are local to others (i.e. `def` constructs in `def` constructs);

`dgoal` which displays the goal expression of an abbreviated `def` construct;

`dfct` which displays the headers of abbreviated function definitions;

`small` which reduces the space demand of an expression by suppressing blanks in the layout whereever possible without creating ambiguities;

`vertical` which changes the layout of syntactical constructs from horizontal to vertical;

Once you are in the command mode (after having pressed F8), you may type into the input line any one of these commands followed by `on` or `off` (with one blank in between), whichever way you want it. Any combination of these layout control modes is legitimate and applies to all components of the expression that is actually in the FA field. A particular mode combination is sustained until explicitely changed, i.e., it may apply to several programs on which you work in succession.

The best way of getting acquainted with these layout commands is by means of a suitable program example which in a nutshell clearly exposes their effects. A good one is the following nonsense program: It consists of three nested function definitions which in some seemingly weird way recursively call on each other, making full use of the higher-order function feature of KIR : all applications are

```
ap def
     F [ U, V ] = def
                     G [ U, V ] = def
                                    H [ U, V ] = G [ F, H ]
                                  in F [ G, H ]
                   in G [ F, G ]
   in F [ F, F ]
to [ C ]
```

solely specified in terms of function identifiers both in function and argument positions, i.e., as applications of functions to functions which return new functions as results. So, it is worth the while to study the program behavior also from this point of view.

When performing a few reductions step by step, you will note that the program expands to some extent due to the substitution of function identifiers by the respective complete function definitions, but otherwise it behaves reasonably well insofar as it cycles, in periods of six reduction steps, through the same sequence of intermediate programs. Thus, the program is in fact tail-end recursive. Since it obviously never terminates, it is also meaningless.

However, let's return to the original purpose of introducing this program. Since it produces in a rather intricate way nesting levels of function defintions and function applications, it is also ideally suited to study on any of the intermediate programs which parts of it are being suppressed or exposed by turning the `abb_def`, `abb_idef`, `dgoal` and `dfct` modes on and off, and how the layout changes when activating or deactivating the `vertical` mode[15].

Having played with these layout modes for a while, you may have gotten a little confused as to which of them are actually on and which are off. To find out what the current situation is, you may enter the command `editparms` which brings up on your screen a number of editor parameters, of which those that are of interest at the moment are at the bottom of the list. Hitting the `enter` (or `return`) key gets you back to the standard π–RED user interface.

4.2 Deleting Terms from a KIR Program

By means of the above program we may also get a little more specific about the deletion of (sub-) terms from an existing program. We indicated in the preceding section that this can be done by making a term to be deleted the cursor expression (i.e., move the cursor to its topmost constructor) and then

[15]The `small` mode has no effect on the layout of this program, you will have to wait until we introduce more KIR constructs to find out what it does.

hitting the `delete` key on your keyboard: the term will disappear and the surrounding program will generally change its syntactical structure accordingly. So, let's try a few `delete` operations on the program to see what happens.

The best way of systematically studying the effects of the `delete` function is to set out with the components of the innermost `def` construct, say, by deleting the formal parameter `U` from the header of the function `H` or by deleting the argument `H` from the application on the right-hand side of this definition. In either case, the term simply disappears together with the comma by which it was separated from the other term inside the brackets. Next you may wish to delete the entire application `G[F]` from the right-hand side of the definition. So, you move the cursor under the function term `G` and hit the `delete` key, whereupon the editor returns the empty term `#` in its place. The same happens if you do this to the goal expression `G[F,G]`. In both cases, the selected term disappears, but otherwise the syntactical structure of the `def` construct is sustained for there are still fragments left that cannot simply go down the drain with what you intended to take away.

Moving the cursor under a `def` constructor and hitting the `delete` key replaces the entire `def` construct by what originally was its goal (or body) expression, i.e., it deletes all the function definitions and thus turns into free variables all occurrences of function identifiers left over in the goal. You may observe this effect over several steps by removing the `def` constructs in your program systematically from innermost to outermost. If there is just one function defined under a `def`, the same can be accomplished by moving the cursor under the function name in the header of the definition. If it comprises more function definitions, only the one thus selected is removed from the entire construct.

As a general rule, the `delete function`

- removes from an n-ary structure the selected component and thus shrinks it to a structure of arity $(n-1)$, provided the component is one of those whose numbers may vary, e.g., a list element, a sub-term of an application, or a function definition in a `def` construct;
- replaces with an empty term `#` an essential component of a construct, e.g., the goal expression in a `def` construct, the right-hand side of a function definition, or any component in a construct with a fixed arity, e.g., in an `if_then_else` clause.

4.3 Some Other Useful Editor Commands

While strictly syntax-directed editing greatly facilitates writing syntactically correct programs, there are also, and inevitably so, some inconveniences associated with it, of which one is the movement of the cursor, and the other one relates to searching for and replacing certain (sub-) expressions, possibly in several places.

The default cursor movement as explained on pages 10 and 11 in fact follows the n-ary tree structures built up by the constructor syntax that is used for the internal representation of KIR -programs as λ-terms. Since this syntax does not in an obvious way relate to the syntax of KIR as you see it on your screen, the cursor movements effected by pressing the respective keys sometimes appear to be rather unusual and unexpected relative to what you are used to from line-oriented editing.

To remedy this problem, you have available under the command mode (to be entered with the function key F8) an editor command `curmode[par]` which, when used with the parameter

`par = off` enables the syntax-oriented (tree traversing) cursor movement;

`par = on` enables a line-oriented mode in which the cursor is moved as indicated by the arrows on the keys.

Another three editor commands may be used to search for and replace (sub-) expressions by others. They are

`findexpr[string]` which searches, from the current cursor position downwards, for the first occurrence of a subexpression starting with `string`, i.e., the subexpression need not be fully specified. The command `next` may be used to search for subsequent occurrences.

`find` works as `findexpr`, but takes as search pattern the topmost expression held in the first auxiliary buffer.

`replace[expr_1;expr_2; mode]` which searches, again from the current cursor position downwards, for occurrences of the subexpression `expr_1` and replaces them with the expression `expr_2`. The parameter `mode` may be set to

- `mode = all` to have replaced all occurrences of `expr_1` from the cursor position downwards;
- `mode = ask` in which case the editor will ask you what to do next, with the following options:
  - **y** make a replacement;
  - **n** skip and move on to the next occurrence of the search expression;
  - **a** terminate the `ask` mode and do all replacements from the current position downwards;
  - **q** quit the `replace` command.

If the parameters of the `replace` command are not specified, the editor will ask you to do so. After completion of a `replace` command the system tells you on the message line how many replacements have actually been made.

4.4 The Filing System

In order to provide some essential file services, π–RED makes use of the underlying UNIX file system. All files which you create while working with the system are stored under the same directory under which you've called the executable file `reduma` for π–RED itself. If π–RED is installed under your own user directory, then you can find `reduma` under the path name

`../user/red/red.o` .

Now, there are basically two ways of calling `reduma`. You can do this

- either directly under your `user` directory with

  `< unix_prompt >: red/red.o/reduma` ,

  in which case all files are placed under `user`;

- or by first climbing down the directory path with

  `< unix_prompt >: cd red/red.o`

  and then calling it with

  `< unix_prompt >: reduma` ,

  in which case you'll find all your files under the directory path `user/red/red.o`[16].

The first item applies equivalently if π–RED is installed under another user's directory which you are allowed to access. Then you may call it with

`< unix_prompt >:~ /other_user/red/red.o/reduma` ,

but all files which you create are again allocated under your own `user` directory.

There are several file types which are distinguished by appropriate default extensions to the file names. These extensions are automatically added by the editor in compliance with the modus of creation.

You know already about the most important file type (or format), so-called editor files, whose names are extended by the suffix `.ed`. This file type may be used to create libraries of KIR expressions, e.g., of complete programs or frequently used function definitions, or as an intermediate store for program components in the process of designing large programs. Creating and accessing edit files is controlled by the function keys F10 and F2. The former enables the

[16] Note that the compiled version π–RED $^+$ requires a special file directory `ed/` under the directory `red/` (and at the same level as the executable file `reduma*`) under which all editor files are being allocated. If not yet existing, this directory must be opened up with the UNIX command `mkdir ed`, otherwise no new editor files can be created under the output mode.

output mode under which the current cursor expression (which may be different from the FA-field) may be written into a new or an existing edit file; the latter enables the input mode under which the current cursor expression may be replaced by the KIR expression held in an existing edit file.

Assembling large programs from several KIR expressions that are stored away in edit files can be elegantly supported by yet another input mode. Rather than explicitely copying the files, they may simply be referenced from within a program by so-called dollar variables of the form `$file_name`, i.e., by their names preceded by a dollar sign. Before actually executing a program, the preprocessor of π–RED substitutes all dollar variables by the respective file contents. This is done systematically from outermost to innermost since all expressions thus inserted may contain further references to other edit files. Note that some care must be taken on the part of the programmer to avoid cyclic references.

You may test this feature on your greatest common divisor program, say, by first copying the function body of `Gcd` into a file named `Body`, and then replacing it with `$Body`. The program then looks like this:

```
def
  Modulo [ X , Y ] = if ( X le Y )
                     then X
                     else Modulo [ ( X - Y ) , Y ] ,
  Gcd [ X , Y ] = $Body
in Gcd [ 6 , 4 ]
```

If you now trigger the system to perform one reduction step, you will see that the complete body of `Gcd` again shows up in all occurrences of the `def` construct, i.e., the program behaves exactly as if the body would have been directly included.

At this point you may wonder why there isn't a dynamic version of this input mode which expands dollar variables by the respective edit file contents at run_time, and only if and when absolutely necessary. Though this mode may save a lot of memory space, there are severe problems with it which largely arise from potential occurrences of free variables in expressions that are filed away. They cannot be naively substituted into a running program without a high risk of changing binding scopes and thus its meaning (or semantics).

Another type of input/output files are so-called red files. The KIR expressions stored in them are treated as macros in the following sense: whenever they are reduced in the context of a large program, they decrement the reduction counter just by one (or count as one instance of reduction), irrespective of the actual number of reduction steps that must be performed. They are distinguished by the suffix `.red` following the file name. You may use (under the command mode F8) the editor commands

`store[file_name]` to store the cursor expression under the file `file_name.red`;

load[file_name] to overwrite the cursor expression with the contents of the file file_name.red .

Similar to edit files, these red files may be referenced from within a program with _file_name, i.e., with an underscore rather than a $ preceding the file name.

You may also copy your cursor expression into an ASCII file, using the command print[file_name], or overwrite the cursor expression with the contents of an existing ASCII file for KiR expressions, using read[file_name]. ASCII file names have the suffix .asc appended to them.

There is yet another file type called prettyprint in which KiR expressions may be stored in the same layout and fond in which you can see them on your screen. These files may either be directly printed or included in other text files. In fact, all the example programs in this manual are, through prettyprint files, copied from the screen. The command pp file_name creates a new (or overwrites an existing) prettyprint file with the cursor expression and appends the suffix .pp to the file name.

4.5 Setting the System Parameters

In order to be able to deal with certain memory space problems that may occur when running large programs, you need to have at least some rudimentary understanding of the internal workings of π–RED . There are also some particularities about the arithmetic subsystem used in π–RED that you ought to know about.

π–RED is an applicative order graph reduction system which uses the following run_time data structures [SmBlKl91b, Gaer91]:

- a program graph (held in a heap segment) whose inner and outer nodes roughly correspond to constructors and string representations of atomic expressions (such as variables, constants, primitive function symbols etc.), respectively, and whose edges correspond to pointers connecting these nodes;
- an array of node descriptors which, as an integral part of the program graph, contains information about class, type and shape of the (sub-) expressions represented by the graph nodes;
- one or several run_time stacks which accommodate the activation records, workspaces etc. for instantiations of defined functions;
- (as part of the interpreter version π–RED *) a shunting yard comprising three stacks about which graph structures are traversed in search for instances of reductions.

Both the heap and the descriptor array are managed based on a reference counting scheme which releases as early as possible the space occuppied by graph structures that are no longer needed. However, as heap space is allocated and de-allocated in pieces of variable lenghts, it may require compaction from time to time in order to satisfy a particular space demand for the placement of a new (sub-) graph. All descriptors have the same size and may therefore be placed into any array entry.

When calling π–RED , the sizes of the memory segments which accommodate these run_time structures are initialized from a `red.init` file with pre-specified values. The editor command `redparms` will display on the screen the memory configuration (together with some other system parameters) with which you are working.

You have several commands available to modify this configuration, say, if one of your programs has been aborted because it has run out of space[17]:

`heapsize no_of_bytes` specifies, in numbers of bytes, the size of the heap segment;

`heapdes no_of_entries` specifies the number of entries in the descriptor array;

`qstacksize no_of_el` specifies, in numbers of stack elements (of four bytes), the size of the larger stacks of the shunting yard;

`mstacksize no_of_el` specifies the size of the smaller stack of the shunting yard;

`istacksize no_of_point` specifies, in numbers of pointers (again of four bytes length), the size of the run_time stack.

You should change these parameters if and only if you really understand what you are doing, i.e., you know about the internal program representation and the way programs are reduced in π–RED . However, as a rule of thumb, you should have at least 500 kBytes of heap space allocated to run programs of modest size, and the ratios between all other parameters should be roughly as follows:

$$heapsize : heapdes : qstacksize : mstacksize : istacksize = 500 : 5 : 20 : 10 : 20 .$$

As already indicated, another set of system parameters relates to the way arithmetic operations are performed in π–RED . You have the choice between a decimal mode which works with variable length number representations[18] and a binary mode which, as usual, works with the fixed integer and floating point formats of your host system. The command

[17] You will always be notified in the message line of the exact cause of the problem, i.e., in which run_time structure the overflow has occured.

[18] Internally, all numbers are represented relative to a base of 10000 (as opposed to BCD codes) for efficiency reasons.

varformat enables the decimal mode which performs all arithmetic operations with potentially unlimited precision (it is the mode that you get as a default option when starting π–RED);

fixformat enables the binary mode, under which all decimal numbers specified in your high-level KiR program are automatically converted to floating point formats if they include a decimal point, and to integer formats if they don't.

In the decimal mode you can compute breathtakingly long numbers which fill the entire FA-field, e.g., if you run the factorial program with an argument value of 1000 or more. However, you may also wish to, and in the case of division you have to (!), put an upper limit on the precision of your result values, i.e. primarily on the number of digits to the right of the decimal point. So, π–RED has parameterized precisions for the results of multiply and divide operations, but also with respect to the maximal number of digits that are displayed on the screen. The commands which may be used to re-define them other than initialized at start-up time are:

div_prec no_of_digits for the divide precision;

mult_prec no_of_digits for the multiply precision;

trunc no_of_digits for the upper limit on the number of digits displayed on the screen.

There is one more parameter called beta_count by which you may determine what the reduction counter is supposed to keep track of. You have the choice between

beta_count on , in which case are counted only reductions of defined function applications which require the instantiation of formal by actual parameters (or β-reductions);

beta_count off , in which case are counted all instances of reduction (i.e., including applications of primitive functions).

All changes effecting the system parameters are immediately confirmed on the message line. Alternatively, you may inspect all parameters at once by means of the command redparms. It will also show you the times used up by your last sequence of reductions for pre-processing, processing and post-processing.

4.6 Defining the Function Keys on your Keyboard

As pointed out in chapter 3, the function keys used by π–RED are logical, i.e., the functions they are supposed to perform are not necessarily assigned to the physical function keys that you find on the keyboard of your workstation or terminal. Some of these keys may be used otherwise by the underlying host system, and there may not even be as many physical keys as π–RED has logical keys. The easiest way to get around this problem is to use only the logical keys by typing `:n` (where `n` is the logical key number), which echoes in the input line as `Fn`.

However, if you wish to assign at least some of the logical keys, e.g., those most frequently used, to physical keys on your keyboard, you may do so by entering the command mode and then typing the command **`defkeys`**. It will bring up on your display screen the actual table of key definitions as it is currently available on your system. This table comprises three columns, of which

- the first simply enumerates all logical keys of π–RED , with indices running from 0 to 20;
- the second lists the logical keys associated with these indices, with the first five keys controlling the cursor movement, and the remaining 16 keys defining the function keys (of which SHIFT_F1 .. SHIFT_F8 stand for F9 .. F16, respectively);
- the entries of the third column may or may not contain some weird codes which identify the physical keys assigned to the particular logical keys (if an entry is empty, then no physical key is assigned).

Typical table entries on a SUN workstation look like this:

```
 0 :        UP : 0x1b 0x5b 0x41

 8 :        F4 : 0x1b 0x5b ox32 0x32 0x37 0x7a

14 : SHIFT_F2 : 0x1b 0x5b 0x32 0x33 0x32 0x7a
```

On top of this table you are asked to enter the index of the logical key to which you wish to assign a physical key. Having done so, you are asked to hit the particular physical key, followed by two blanks, which will enter the code for this key in the table (thereby possibly overwriting an existing assignment). This works only for physical function keys (including those reserved for cursor movement) that are not yet otherwise pre-assigned by the host system.

You may do these assignments for several keys in succession. If you are finished, simply hit the return key, and you will get back to the standard π–RED screen layout. The key assignments, together with all other system parameters, are entered into the π–RED initialization file `red.init`.

5 Full β-Reductions and Higher-Order Functions

Let us now return to writing programs in KIR since there is lots more to learn about its constructs and features, particularly about those that are not available in most other functional or function-based languages.

One concerns the implementation of a λ-calculus as an essential pre-requisite for simple symbolic computations, particularly when it comes to exploiting the full potential of higher-order functions, including self-applications and partial applications. Parameter passing by full β-reductions guarantees the orderly resolution of naming conflicts between (relatively) free and bound variable occurrences. This problem potentially arises whenever, in the course of reducing defined function applications, argument terms other than basic values are being substituted into the scopes of other functions (or abstractions). If this is done naively, occurrences of free variables in the argument terms may get parasitically bound and thus change the meaning of the entire program. Even worse, depending on the order in which reductions are performed in a particular program, this phenomenon may or may not occur, i.e., the referential transparency which allows you to perform reductions in any part of a program would be lost.

Two simple examples may help to illustrate this problem.

The first one relates to the use of the function

$$\mathtt{twice}[\mathtt{f},\mathtt{x}] = \mathtt{f}[\mathtt{f}[\mathtt{x}]]$$

which applies its first argument (the one substituted for `f`) twice to its second argument (the one substituted for `x`). If this function of two parameters is just applied to itself, i.e., we have both a self-application and a partial application, we eventually get another function of two parameters which in fact doubles the application of `twice`, i.e., it applies its first argument four times to its second argument.

A KIR program which first computes this function and then applies it to another function, say `square[x] = (x * x)`, and to an argument value, say 2, looks like this:

```
let Double_twice = def
                     Twice [ F , X ] = F [ F [ X ] ]
                   in Twice [ Twice ] ,
    Square = sub [ X ]
             in ( X * X )
in Double_twice [ Square , 2 ]
```

Note that this program has the functions `Double_twice` and `Square` defined as `let` variables since both are non-recursive. This ensures that the right-hand side of the definition for Double_twice is evaluated, i.e., a new function is actually computed, before it is applied in the outermost goal expression[19]. Moreover, when defining Double_twice in this form, you don't need to think about names for its formal parameters. As you will see in just a moment, they are implicitely already defined.

If you now select the right-hand side of the definition for Double_twice, i.e., the def-construct for Twice, as the actual cursor expression and reduce it step-by-step, you will end up, after six reductions, with a new defining expression for Double_twice (don't worry for the moment what happens in between for the expressions that come up may be rather confusing on first sight):

```
let Double_twice = sub [ X ]
                   in sub [ X ]
                      in \X [ \X [ \X [ \X [ X ] ] ] ] ,
    Square = sub [ X ]
             in ( X * X )
in Double_twice [ Square , 2 ]
```

Double_twice is here returned as a so-called curried function of two parameters, i.e., two nested functions of one parameter each, since its body is embedded in two nested sub constructs. But strangely enough, both subs bind the same variable name X. Moreover, if you have a close look at the body of this function, you will notice that there are four occurrences of \X, i.e., of the variable X preceded by a back_slash, and one occurrence of X without it. Now, just what is this supposed to mean?

The answer to this is very simple: The back_slashes are generated as an integral part of the β-reduction rule as it is implemented in π–RED . Rather than introducing other variable names (as the classical definition of β-reduction requires [Bar81, HiSe86]), this rule realizes a dynamic indexing scheme which renders conflicting variable occurrences distinguishable and sustains a static (or lexical) scoping rule. The back_slashes preceding the occurrence of a particular variable, say X, simply count the number of binders sub[X] for the same variable name that, on a straight path in the syntax tree, are between this occurrence and the binder sub[X] that actually binds it.

In our example, this means that

- the variable occurrence X is bound to the innermost sub[X] for there must be no other sub[X] in between;

[19]Remember that the semantics of let constructs prescribes the substitution of let-defined variables by the values (or normal forms) of the respective defining expressions, whereas the recursively bound variables of a def construct are always substituted by the function bodies as they are.

- all variable occurrences \X are bound to the outermost sub[X] for there must be one binder sub[X] in between (which is the innermost sub[X]).

Thus we have two identically named parameters X in this function whose occurrences in the body can nevertheless be uniquely related to different binding levels (or occurrences of binders sub[X])[20].

When continuing with further step-by-step reductions of the entire program, i.e., after having moved the cursor back underneath the let-constructor, π–RED first substitutes in the goal expression all formal by actual parameters, returning the application:

```
ap sub [ X ]
   in sub [ X ]
      in \X [ \X [ \X [ \X [ X ] ] ] ]
to [ sub [ X ]
     in ( X * X ) , 2 ]
```

The next step reduces the outermost application and thereby substitutes all occurrences of \X which are bound to the outermost sub[X] by the Square function, returning the application of a unary function to the remaining argument 2:

```
ap sub [ X ]
   in ap sub [ X ]
         in ( X * X )
      to [ ap sub [ X ]
              in ( X * X )
           to [ ap sub [ X ]
                   in ( X * X )
                to [ ap sub [ X ]
                        in ( X * X )
                     to [ X ] ] ] ]
to [ 2 ]
```

Another reduction step substitutes by the argument 2 all occurrences of X in the function body that are bound to the remaining sub[X]. This is just the X in the argument position of the innermost of the nested applications:

[20] You will find a precise definition of binding levels and of the β-reduction rule in terms of these back_slashes (or protection keys) further down in this chapter, after we have worked our way through the second example and thus developed a better intuitive understanding as to how back_slashes are being manipulated.

```
ap sub [ X ]
   in ( X * X )
to [ ap sub [ X ]
        in ( X * X )
     to [ ap sub [ X ]
             in ( X * X )
          to [ ap sub [ X ]
                  in ( X * X )
               to [ 2 ] ] ] ]
```

From this point on, the applications of `Square` are reduced from innermost to outermost, yielding `65536` as the final result.

If the system would perform `naive substitutions` instead of full β-reductions, things would turn out quite differently. Starting with the above program and again making the defining expression for `Double_twice` the cursor expression, we would get, after six reduction steps, another function body for `Double_twice`[21]:

```
let Double_twice = sub [ X ]
                   in sub [ X ]
                      in X [ X [ X [ X [ X ] ] ] ] ,
    Square = sub [ X ]
             in ( X * X )
in Double_twice [ Square , 2 ]
```

As all occurrences of `X` in its body would be bound to the innermost `sub[X]`, and there would be no variable occurrence bound to the outermost `sub[X]`, the program would produce, after three more reduction steps, the nested application

```
ap 2
to [ ap 2
     to [ ap 2
          to [ ap 2
               to [ 2 ] ] ] ]
```

which, of course, is dead wrong.

[21]Note that this function has been manipulated by hand to what it is. π–RED could not derive it by regular means from the original program.

Note that an entirely different course of actions takes place if the program is reduced strictly from outermost to innermost, i.e., with the cursor always remaining under the outermost constructor. As you may check out for yourself, π–RED then does not fully reduce the partial application `Twice[Twice]`. Instead, it just substitutes the defining expression of `Twice` in itself and applies the abstraction thus obtained directly to the arguments `Square` and `2`, as it always continues with what is actually the topmost application. However, irrespective of this and all subsequent intermediate expressions that are being brought about, the computation again terminates with the correct value `65536`.

Let us now turn to a more elaborate example which shows you how the number of back_slashes dynamically changes in the course of performing β-reductions that produce name clashes over several binding levels.

```
ap sub [ U , V ]
   in ap sub [ V ]
         in ap sub [ V ]
               in ap sub [ V ]
                     in ap sub [ V ]
                           in ap sub [ V ]
                                 in ap V
                                    to [ U ]
                              to [ ap V
                                   to [ U ] ]
                        to [ ap V
                             to [ U ] ]
                  to [ ap V
                       to [ U ] ]
            to [ ap V
                 to [ U ] ]
      to [ ap V
           to [ U ] ]
to [ V ]
```

This program simply applies a nameless function of two parameters `U` and `V` to a single argument `V`, i.e., we have a partial application to a globally free variable. When substituting this variable for free occurences of `U` in the body of this function, it is going to penetrate the scopes of several nested binders `sub[V]`, including the one that is left over from the outermost binder `sub[U,V]`.

A full β-reduction of this application yields the expression shown on top of the opposite page. Here you can clearly see that each occurrence of what originally was the free variable `V` now carries as many back_slashes as there are binders `sub[V]` into the scopes of which it has been substituted. This is to say that there is again no `sub[V]` to which these occurrences are bound, i.e., the

```
sub [ V ]
in ap sub [ V ]
      in ap sub [ V ]
            in ap sub [ V ]
                  in ap sub [ V ]
                        in ap sub [ V ]
                              in V [ \\\\\\V ]
                           to [ V [ \\\\\V ] ]
                     to [ V [ \\\\V ] ]
               to [ V [ \\\V ] ]
         to [ V [ \\V ] ]                       <=
   to [ V [ \V ] ]
```

variable maintains its status of being globally free (Note also that the editor has changed all innermost applications originally denoted as `ap V to [ U ]` into `V [\...\V]`).

Further reductions will systematically eat up from outermost to innermost the applications inside the body of the new function, unless you change this order by selecting an inner application as the actual cursor expression. So, let's do first what is currently the outermost application. It is going to substitute the argument term `V[\V]` for occurrences of `V` that are bound to the outermost sub[V] of the abstraction that is in function position of this application (which is just the `V` in the line marked by a `<=`). Thus, what we get next is this:

```
sub [ V ]
in ap sub [ V ]
      in ap sub [ V ]
            in ap sub [ V ]
                  in ap sub [ V ]
                        in V [ \\\\\V ]
                     to [ V [ \\\\V ] ]       <<==
               to [ V [ \\\V ] ]
         to [ V [ \\V ] ]
   to [ ap V [ \V ]
        to [ \V ] ]
```

The substitution has taken place as expected, but you may also notice that all occurrences of \ ...\V have lost one of their back_slashes. This is due to the fact that, together with the application, a binder **sub[V]** has disappeared too, i.e., the number of nested scopes has been reduced by one with respect to all occurrences of back_slashed `V`'s.

You are now invited to move the cursor to the innermost application for a change and perform a reduction there. Other than substituting `V[\\ \\V]` (which is the argument term in the line marked `<<==`) for the innermost `V`, it just decrements by one the number of back_slashes of the variable occurrence `\\\\\V`, but leaves all other back_slashed occurrences of `V` as they are. This reflects the fact that only the innermost binder `sub[V]` has disappeared.

Continuing in any order will finally result in an abstraction

```
sub [ V ]
in ap ap ap ap ap V [ \V ]
               to [ \V ]
            to [ \V ]
         to [ \V ]
      to [ \V ]
   to [ \V ]
```

whose body is composed of nested applications of the bound variable `V` to occurrences of the free variable `\V`.

If you now apply this abstraction to another free variable, say `U`, you get after one more β-reduction

```
ap ap ap ap ap U [ V ]
            to [ V ]
         to [ V ]
      to [ V ]
   to [ V ]
to [ V ]
```

in which no more binders and hence no more back_slashed variables occur.

The back_slashes formally define the binding status (or binding level) of a variable occurrence as follows [Brui72, BeFe82b]:

Let $\backslash_{(n)}$ `V` denote a variable occurrence preceded by n back_slashes, then with respect to a binder `sub[V]` which is k intervening binders `sub[V]` away from it (counted upwards in the syntax tree), this variable is said to be

- protected if $k < n$;
- free if $k = n$;
- bound if $k > n$.

Since the purpose of the back_slashes obviously is to protect a variable occurrence against parasitic bindings, we will also refer to them from now on more appropriately as protection keys, and to the number of protection keys preceding a variable also as its protection index.

Of course, what has been defined here for the binding status of a variable relative to a sub-binder applies equivalently to variables bound in a recursive function definition of the form

$$\texttt{F[X\_1, ... ,X\_n]} = \texttt{Expr} \ .$$

Following standard λ-notation, this definition can be equivalently written as

$$\texttt{F} = \texttt{sub[X\_1, ... ,X\_n] in Expr} \ ,$$

which is also the syntax used for the internal graph representation.

The β-reduction rule for an application of the general form

$$\texttt{ap sub[V] in Expr to [Arg]}$$

as it is implemented in π–RED can now be defined thus:

An occurrence of a variable$\backslash_{(\mathtt{n})}$`V` in the body expression `Expr`

- decrements its protection index n by one if it is protected;
- is substituted by the argument `Arg` if it is free;
- is left unchanged if it is bound.

An occurrence of $\backslash_{(\mathtt{n})}$`V` in the argument term `Arg`

- is left unchanged if it is free or bound;
- increments its protection index n by m if it is protected and substituted into the scopes of m nested binders `sub[V]` inside `Expr`.

There is an interesting facet to this dynamic indexing scheme that you should be aware of, even though it may be not too relevant from a high-level programmer's point of view: all formal parameters of a defined function may be named identically (or mechanically transformed into identical names) since occurrences of these parameters in the function body can be uniquely related to their correct binding levels solely by means of their protection indices.

To get this point across, consider first the following application of a simple function that is defined in terms of three distinctly named formal parameters `X`, `Y` and `Z`:

```
def
  F [   ] = sub [ X , Y , Z ]
            in X [ Y [ Z [ X , Y ] ] ]
in F [ A , B , C ]
```

Thus it is pretty obvious where the argument terms A, B and C of this application will be substituted. When reducing it, we get:

```
A [ B [ C [ A , B ] ] ]
```

Since occurrences of formal parameters in a function body are merely place-holders for things that eventually will be substituted for them, it is, of course, completely irrelevant which parameter names you actually choose. The result of applying the function will always be the same. So, you may change any of the parameter names as long as the changes are made consistently and do not get into conflicts with other names.

A renaming scheme which accomplishes this is known from the theory of the λ-calculus as so-called α-conversion. It is based on an abstraction

$$\texttt{Alpha} = \texttt{sub[U , V] in U[V]}$$

which, when applied to another abstraction, converts the outermost parameter of the latter to V. You may convince yourself of this by means of the following example:

```
def
  Alpha [   ] = sub [ U , V ]
                in U [ V ]
in Alpha [ sub [ X ]
           in X [ X ] ]
```

Here we have a partial application of Alpha which, in a first step, reduces to an abstraction in one parameter V. The application that has come about in its body inserts, in a second step, the variable V for occurrences of X in the body of sub[X] in X [X], as a result of which you get this abstraction with X replaced by V. Note that this is a special case of a higher-order function: Alpha is some kind of identity function which returns the functions to which it is applied with just the outermost formal parameter name exchanged, but otherwise in the same syntactical form.

Things are getting decidedly more interesting if you wish to convert all formal parameters of an abstraction into the same name, say again V. Since the α-conversion scheme employs full β-reductions, nothing should go wrong. So, let's try it with the function of three parameters introduced in the above example.

In order to have all three binders of this function converted in one pass, you should modify the α-conversion function so that it recursively eats its way through them from outermost to innermost. The following program will do this job:

```
def
  Alpha [ N , U , V ] = if ( N gt 1 )
                        then Alpha [ ( N - 1 ) , U [ V ] ]
                        else U [ V ]
in Alpha [ 3 , def
                 F [   ] = sub [ X , Y , Z ]
                           in X [ Y [ Z [ X , Y ] ] ]
               in F ]
```

The α-conversion function now has a third parameter `n` which must be set to the number of parameters that you want to be converted (this parameter simply terminates recursive calls of `Alpha`), which in this particular case are three. If you reduce this program step-by-step, you may get a little confused about the intermediate expressions that build up (that's just the way reduction works by the books), but you will end up with an abstraction of the form:

```
sub [ V ]
in sub [ V ]
   in sub [ V ]
      in \\V [ \V [ V [ \\V , \V ] ] ]
```

which has all variables converted to `V` and different binding levels distinguished by protection keys. All occurrences of

- `V` are bound to the innermost `sub[V]`;
- `\V` are bound to the `sub[V]` in the middle;
- `\\V` are bound to the outermost `sub[V]`.

To convince yourself that this is the same function as before, you may simply type as input `ap % [a,b,c` and hit the return key, which creates the application:

```
ap sub [ V ]
   in sub [ V ]
      in sub [ V ]
         in \\V [ \V [ V [ \\V , \V ] ] ]
to [ A , B , C ]
```

Three more reduction steps will produce the same resulting expression as before:

```
A [ B [ C [ A , B ] ] ]
```

So, you are free to apply any function to any other function in order to compute new functions without worrying about variable names and binding scopes. The β-reduction rule untangles correctly the messiest situations of name clashes you can think of, i.e., it maintains all scopes as originally defined and thus guarantees referential transparency and the Church-Rosser property. It also avoids the alienation of intermediate programs beyond recognition as it maintains the variable names of the original program unless you explicitely change them into others by α-conversion. However, even more important is that functions are thus truly treated as first class ojects: they cannot only be passed as parameters but also be returned in intelligible form, i.e., in KIR notation, as results of rather complex computations.

This is in sharp contrast to most other implementations of functional languages which, contrary to what is claimed, separate the world of functions from the world of objects they can be applied to [BuMQSa80, HaMcQMi86, Turn85, HuWa88]. This is primarily a consequence of compiling functions, in one way or another, to static code, which for all practical purposes precludes full β-reductions [Aug84, John87, ApMcQu87]. Thus, rather than computing new functions as results, say, of partial applications, the respective run-time systems internally form so-called closures, i.e., packages composed of pieces of code and (references to) argument objects, and notify you merely of the fact that the computation has terminated with a function of some number of arguments, but they cannot tell you what the function looks like. Also, since these systems are based on polymorphically typed languages, accepting only programs that are well typed, they cannot deal with globally free variables either.

6 Structured Objects and Pattern Matching

6.1 Primitive Operations on Lists

The simplest way of representing structured objects in KIR are lists or sequences of expressions which have the general form

$$< \texttt{expr\_1}, \ \dots \ , \texttt{expr\_n} >,$$

where $<$ is an n-ary list constructor, $>$ is an obligatory end-of-list symbol, and $expr\_1$, ... , $expr\_n$ may be any of the legitimate KIR expressions, i.e., lists (sequences) may also be recursively nested without any bounds on their depths.

There are a number of primitive structuring functions available. They require as arguments a list parameter `list` and an index parameter `index` specifying one or several index positions which are to be affected by the respective operations. These applications are reducible if both arguments substituted for the parameters are of the respective types, and the `index` value does not exceed the arity of the list. Otherwise the applications remain as they are.

In particular, the following functions are available:

`lselect[index,list]` which returns from a list the evaluated component expression found in index position `index`;

`lcut[index,list]` which cuts off as many of the leading components of a list as are specified by a positive value of `index`, and cuts off as many of the trailing components of the list as are specified by a negative value of `index`;

`lrotate[index,list]` which rotates to the right a list by as many positions as are specified by a positive `index` value, and to the left by a negative index value;

Other functions whose applications require a slightly different syntax include

`lreplace[index,expr,list]` which replaces the component in position `index` of `list` by the value of `expr`, thereby conceptually creating a new list;

`reverse[list]` which simply reverses the order of components of a list;

`lunite[list_1,list_2]` which appends the components of `list_2` to those of `list_1`, thus forming a new united list.

A simple program which searches for the occurrence of a particular value in a sequence of integer numbers may illustrate the use of two of these functions, `lselect` and `lcut` (see top of next page):

The function `Search` basically inspects what is actually the first element of a list substituted for the parameter `Sequ`, and compares it for equality with

```
def
  Search [ Sequ , El ] = if empty [ Sequ ]
                         then 'not found`
                         else if ( lselect [ 1 , Sequ ] eq El )
                              then 'found`
                              else Search [ lcut [ 1 , Sequ ] , El ]
in Search [ < 5 , 7 , 9 , 4 , 6 , 3 , 6 , 2 , 1 > , 7 ]
```

a value substituted for `El` and, if this is the case, returns the string `'found``. Otherwise, it is recursively applied to the remainder of the list until either the next element matches or the list is exhausted (or empty), in which case the string `'not found`` is returned. So, when applied to arguments as above, the program terminates, after two recursive calls of `Search`, with `'found``; if the second argument is changed to `8`, it terminates, after 10 recursive calls of `Search`, with `'not found``.

At this point it should be explained why KiR does not yet support the primitive functions `head`, `tail` and `cons` which you can find in many other functional or function-based languages. The reason for this is that lists (sequences) generally are n-ary structures whereas these functions are usually defined for binary structures. The latter are merely special cases of the former which π–RED distinguishes neither in the external nor in the internal (graph) representations. So, implementing these functions could potentially lead to ambiguities. However, this feature is available in Orel/2 and will soon be implemented in KiR as well.

Two more things need to be mentioned: If you write programs that need to concatenate two lists, you may do so by using either `lunite` or alternatively the infix notation

$$(\texttt{sequ\_1} \; ++ \; \texttt{sequ\_2})$$

which is semantically the same. Likewise, applications of the function `select` may be alternatively specified as

$$(\texttt{index} \mid \texttt{list})$$

In fact, π–RED usually transforms occurrences of `lunite` or `select` into these infix notations when returning intermediate programs.

6.2 Strict Pattern Matching

The structuring functions just introduced are complete in the sense that they suffice to specify all application problems involving structured objects (in a strict sense some of them are even redundant). However, KiR also provides, in the form of pattern matches, elegant means to manipulate structures in a more general sense.

Pattern matching is available in all modern functional languages [Turn86, Lavi87, HuWa88] and primarily used as an elegant way of defining functions as ordered sets of alternative equations. In this setting, pattern matching may simply be seen as a generalized parameter-passing mechanism.

However, since pattern matching generally is an operation which extracts (sub-) structures of particular shapes and elements (or element types) from given structural contexts and substitutes them for placeholders in other structural contexts, its full potential definitely lies in complex rule-based modifications of structured objects. Exploiting this potential is a particularly difficult and hence challenging problem when it comes to performing structuring operations on lists (sequences) of heterogeneously typed components with varying nesting levels and arities. Here pattern specifications may have to include so-called wild cards which extract (sub-) structures of variable lengths from otherwise fitting contexts, e.g., in order to decompose a structure depending on the occurrence of specific patterns in varying syntactical (structural) positions. Such pattern matching generally requires backtracking and additional guards which test for certain properties of the matching components.

Pattern matches are in KiR defined in terms of special functions which have the form of so-called WHEN-clauses:

```
when (Pattern_1, Pattern_2, ... ,Pattern_n) guard Guard_expr do P_expr.
```

In this construct, `Pattern_i` denotes either a variable, a constant, a type specification, or a recursively nested sequence (list) of these items. The `when` constructor, similar to a `sub`, binds free occurrences of the pattern variables in the optional guard expression `Guard_expr` and in the body expression `P_expr`. The application of such a pattern matching function to n argument terms is denoted as[22]:

```
ap  case
    when (Pattern_1; ... ,Pattern_n) guard Guard_expr do P_expr
    end
to  [ Arg_1, ... Arg_n ]
```

and reduces as follows: If the syntactical structures of the patterns and of the corresponding (evaluated) argument terms match in that for all pairs `Pattern_i/Arg_i`

- for each (sub-) sequence of the pattern there exists a corresponding (sub-) sequence in the argument so that

[22] Note that the parentheses `(,)` should be dropped if there is just on pattern specified following the `when`, otherwise the editor may reject your program.

- each variable of the pattern can be associated with a (sub-) structure of the argument,
- each constant of the pattern matches literally with a constant in an equivalent syntactical position of the argument;
- each type specification of the pattern matches with an object of this type in an equivalent syntactical position of the argument,

then all free occurrences of the pattern variables in `Guard_expr` and `P_expr` are substituted by the respective argument (sub-) structures. If, in addition, the guard expression thus instantiated evaluates to `true`, then the entire application reduces to the weak normal form of the instantiated body expression `P_expr`. Otherwise, the arguments are considered not within the domain of the pattern matching function (or not of a compatible type), i.e., the application remains as it is. Following this interpretation, a pattern in fact defines a structured data type: it is the set of all objects which match the pattern and satisfy the guard.

When applying pattern matches, they must by convention be embedded in so-called CASE-constructs. To do so, you may simply type in the input line a fragment of it, `ap case when`, whereupon the editor will return in the FA field:

```
ap case
   when # guard # do #
   #
to [ # ]
```

Now you may proceed to replace the empty symbols as you like. Note that the editor inserts a `#` even in the place where you would expect the keyword `end`. This relates to a generalization of the `case` construct to which we will return a little later.

You should play with a few simple example programs to find out what pattern matching can do for you. The following may be quite instructive:

```
ap case
   when ( X , Y , 2 ) guard ( empty [ lcut [ 1 , Y ] ] eq false ) do X [ Y ]
   end
to [ reverse , < 1 , 2 , 3 , 4 > , 2 ]
```

Here you have under the `when` clause three simple patterns, of which two are variables and one is a constant. The guard tests whether the argument substituted for the second pattern, i.e., for the variable `Y`, is a list of at least two elements. This being the case, the argument substituted for `X` is applied to the argument substituted for `Y` in the body of the pattern matching function. Since the list of arguments comprises the primitive list function `reverse` in the first position, a list of four numbers in the second position, and the constant value 2 in the third position, the pattern match ought to succeed. So, let's see what happens step-by-step.

After one reduction step, π–RED will produce on your screen the application of a modified `case` construct to the same list of arguments (note that the layout mode is from now on changed to `vertical` in order to have all expressions fit onto the page):

```
ap case
   when ( . , . , . )
    guard ( empty( lcut( 1 , < 1 , 2 , 3 , 4 > ) ) eq false )
    do reverse( < 1 , 2 , 3 , 4 > )
   otherwise ap case
                when ( X , Y , 2 )
                 guard ( empty( lcut( 1 , Y ) ) eq false )
                 do X [ Y ]
                end
             to [ reverse , < 1 , 2 , 3 , 4 > , 2 ]
   end
to [ reverse , < 1 , 2 , 3 , 4 > , 2 ]
```

The patterns in the `when` clause are now replaced by dots, indicating that the patterns did match the arguments (a variable matches every legitimate KIR expression), as a consequence of which they have been substituted for all occurrences of the pattern variables in both the guard expression and the body expression, i.e., the function `reverse` for `X` and the list `<1,2,3,4>` for `Y`. In addition to this, π–RED has extended the `case` construct by an otherwise expression consisting 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, another three reduction steps will reduce 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 > .
```

Now you may try to to reduce the same program with just one element in the argument list:

```
ap case
   when ( X , Y , 2 )
    guard ( empty [ lcut [ 1 , Y ] ] eq false )
    do X [ Y ]
   end
to [ reverse , < 1 > , 2 ]
```

The guard term now reduces to `false` since `lcut[1,<1>]` yields the empty list `<>`. As a consequence of this, the copy of the original application held as `otherwise` expression is reproduced as result. π–RED suppresses further reductions on this application in order to avoid unending recursions.

The pattern matches themselves may be made to fail by the following variant of this program:

```
ap case
   when ( X , Y , 2 )
    guard ( empty( lcut( 1 , Y ) ) eq false )
    do X [ Y ]
   end
to [ reverse , < 1 , 2 , 3 , 4 > , 3 ]
```

Here you have in the third argument position the constant value `3` which does not match the `2` in the corresponding pattern position. So, the pattern match is not at all applicable to the argument list, i.e., π–RED reproduces the application without performing a reduction step (or leaves it as it is).

You may test for yourself other odd argument sets, e.g., a constant value, an application or a variable in the second argument position, where a list is expected. In all cases, the system reproduces the original application since neither of these arguments belongs to the domain of the pattern matching function.

Another example program which shows you how a nested sequence can be decomposed is the following:

```
ap case
   when  < < X , Y > , Z >  guard true do Z [ X , Y ]
   end
to [ < < 3 , 2 > , + > ]
```

Here you have a pattern which exactly matches the argument. With the guard trivially set to `true`, π–RED returns after one reduction step the instantiated body (in infix notation though) `( 3 + 2 )`, and after one more step the value `5`.

The match does also succeed if you have substructures rather than atomic components in your argument sequence since, again, a variable in the pattern matches all legitimate KiR expressions. Thus, an application of the same pattern matching function to a list of lists

```
ap case
   when  < < X , Y > , Z >  guard true do Z [ X , Y ]
   end
to [ < < < 1 , 2 > , < 3 , 4 > > , ++ > ]
```

reduces in one step to `( < 1 , 2 > ++ < 3 , 4 > )` and in another step to `< 1 , 2 , 3 , 4 >`.

There is a mismatch between the pattern and the argument if the latter contains fewer or more components on a nesting level than are specified in the former, e.g., as in

```
ap case
   when < < X , Y > , Z > guard true do Z [ X , Y ]
   end
to [ < < < 1 , 2 > , < 3 , 4 > , < 5 , 6 > > , ++ > ]
```

or in

```
ap case
   when < < X , Y > , Z > guard true do Z [ X , Y ]
   end
to [ < < < 1 , 2 > > , ++ > ] .
```

Neither of these two applications therefore constitutes an instance of reduction: they are constant expressions which remain unchanged.

As you may have expected, the CASE-construct can be expanded to include several pattern matches and an optional `otherwise` expression. Applications of such an alternative pattern match have the general syntactical form:

```
ap   case
       when (Pattern_11; ... ,Pattern_1n) guard Guard_expr_1 do P_expr_1
          ⋮
       when (Pattern_m1; ... ,Pattern_mn) guard Guard_expr_m do P_expr_m
       otherwise Esc_expr
     end
to   [ Arg_1, ... Arg_n ]
```

The `case` construct constitutes an n-ary function which, when applied to n argument terms, tries all pattern matches in the sequence from top to bottom. The first successful match, including a satisfiable guard, is reduced to its (weak) 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 as it is either explicitely specified or, if this specification is missing, to the `case` application itself which π–RED alternatively takes for it (compare also the first example program in this section).

6.3 An Example : Quicksorting

Let us now study a more elaborate program which uses alternative pattern matching in a rather tricky way: quicksorting a sequence of integer values.

The quicksort algorithm basically works as follows: If the sequence is empty or contains just one element, then it is already sorted, and you are done. If it contains two or more elements, then the second element is taken as a so-called pivot number against which all elements of the sequence are compared: those that are smaller are put into a first subsequence, those that equal the pivot number are put into a second subsequence, and those that are greater are sorted into a third subsequence. Subsequently, the first and the third subsequence are recursively quicksorted until they contain only one element or are empty and thus are sorted. The second subsequence is trivially sorted. Finally, all sorted subsequences are recursively concatenated to a flat sequence which has all elements of the original sequence sorted in ascending order.

A KiR program that realizes this algorithm is shown on the opposite page (note that the vertical layout mode is on, otherwise the program would not fit onto the page).

This program breaks down into the definition of a function

`Quicksort` which in its body uses a `case` construct to check whether the argument substituted for the parameter `Sequ` is an empty sequence or a sequence of one element, and otherwise applies to it another pattern match which constructs the recursive calls of `Quicksort` to the subsequences received from an application of the function

```
def
  Quicksort [ Sequ ] =
   ap case when <> guard true do <> ,
      when < X > guard true do < X >
      otherwise ap case
                 when < S_1 , S_2 , S_3 > guard true
                  do ( Quicksort [ S_1 ] ++ ( S_2 ++ Quicksort [ S_3 ] ) )
                 end
              to [ Sort [ lselect [ 2 , Sequ ] , Sequ , <> , <> , <> ] ]
      end
   to [ Sequ ] ,
  Sort [ Pivot , Sequ , U , V , W ] =
   let First = lselect [ 1 , Sequ ]
   in if empty [ Sequ ]
      then < U , V , W >
      else Sort [ Pivot ,
                   lcut [ 1 , Sequ ] ,
                                 if ( First lt Pivot )
                                 then ( U ++ < First > )
                                 else U , if ( First eq Pivot )
                                          then ( V ++ < First > )
                                          else V , if ( First gt Pivot )
                                                   then ( W ++ < First > )
                                                   else W ]
in Quicksort [ < 6 , 4 , 8 , 1 , 9 , 4 , 3 , 2 > ]
```

`Sort` which sorts, by means of the parameter `Pivot`, the sequence substituted for `Sequ` into three subsequences named `U,V,W`,

as just described. The goal expression of the `def` construct applies `Quicksort` to a sequence of integer values.

Now, let's see how this program reduces.

The first reduction step just expands the goal expression by substituting the argument sequence for all occurrences of `Sequ` and the entire `def` construct for all occurrences of `Quicksort` and `Sort` in the body of `Quicksort`. The next step reduces the application of the outermost `case` construct of this body, returning the `otherwise` expression since the sequence contains more than one element (see next page).

The important part of this expression is what is hiding under the question mark in the argument position of the new `case` application, which is the application of `Sort` to concrete argument terms. This application is supposed to

```
ap case
   when < S_1 , S_2 , S_3 > guard true
    do ( ap def
              Quicksort [ Sequ ] =
               ap case
                  when <> guard true do <> ,
                  when < X > guard true do < X >
                  otherwise ap case
                               when < S_1 , S_2 , S_3 > guard true
                                do ( Quicksort [ S_1 ] ++ ( S_2 ++ Quicksort [ S_3 ] ) )
                               end
                            to [ Sort [ ( 2 | Sequ ) , Sequ , <> , <> , <> ] ]
                  end
               to [ Sequ ] ,
              Sort [ Pivot , Sequ , U , V , W ] =
               let First = ( 1 | Sequ )
               in if empty( Sequ )
                  then < U , V , W >
                  else Sort [ Pivot ,
                                lcut( 1 , Sequ ) ,
                                  if ( First lt Pivot )
                                  then ( U ++ < First > )
                                  else U , if ( First eq Pivot )
                                  then ( V ++ < First > )
                                  else V , if ( First gt Pivot )
                                           then ( W ++ < First > )
                                           else W ]
            in Quicksort
         to [ S_1 ] ++ ( S_2 ++ ap def
                                     Quicksort [ Sequ ] =
                                         ap case
                                            when <>
                                             guard true
                                             do <>,
                                            ?
                                         to [ Sequ ] ,
                                        ?
                                  in Quicksort
                               to [ S_3 ] ) )
   end
to [ ? ]
```

reduce to a sequence of three subsequences which are sorted against the second element of the original argument sequence. When moving the cursor under this question mark and pressing the function key F9 (for 1000 reduction steps), you get in this position, as expected,

```
< < 1 , 3 , 2 > , < 4 , 4 > , < 6 , 8 , 9 > >
```

The next reduction step applies the pattern `< S_1 , S_2 , S_3 >` to this argument and creates recursive calls of `Quicksort` to the sequences associated with `S_1` and `S_3` which are concatenated with the sequence bound to `S_2` (in order to avoid unnecessary details, the layout modes `abb_def` and `dgoal` are turned on here):

```
( ap def*
     in Quicksort
  to [ < 1 , 3 , 2 > ] ++ ( < 4 , 4 > ++ ap def*
                                            in Quicksort
                                         to [ < 6 , 8 , 9 > ] ) )
```

If you now move the cursor to the innermost application of `Quicksort` and reduce it locally, you will get

```
( ap def*
     in Quicksort
  to [ < 1 , 3 , 2 > ] ++ ( < 4 , 4 > ++ ( ap def*
                                              in Quicksort
                                           to [ < 6 > ] ++ ( < 8 > ++ ap def*
                                                                         in Quicksort
                                                                      to [ < 9 > ] ) ) ) )
```

and when locally reducing the new applications of `Quicksort` on the right

```
( ap def*
     Quicksort
  to [ < 1 , 2 , 3 > ] ++ ( < 4 , 4 > ++ ( < 6 > ++ ( < 8 > ++ < 9 > ) ) ) )           ,
```

and so on, until you get as the final result the sorted list

```
< 1 , 2 , 3 , 4 , 4 , 6 , 8 , 9 >      .
```

If you have fun playing with this program, you may try a somewhat larger argument sequence, say,

```
< 54 , 87 , 5 , 2 , 98 , 45 , 23 , 45 , 56 , 12 , 343 , 87542 ,
  98 , 56 , 5 , 3 , 5 , 8 , 1 , 9 , 7 , 56 , 44 , 45 , 76 , 87 ,
  98 , 99 , 12 , 34 , 45 , 56 , 76 , 54 , 43 , 32 , 21 , 98 ,
  99 , 5 , 2 , 3 , 4 , 45 , 56 , 88 , 3 , 5 , 0 , 99 , 88 , 23 ,
  12 , 54 , 87 , 67 , 46 , 86 , 98 , 3 , 4 , 6 , 98 , 0 , 45 ,
  65 , 3 , 1 , 9 , 7 , 65 , 5 , 4 , 98 , 6532 , 34 , 76 , 56 ,
  2 , 3 , 98 , 6 , 5 , 4 , 3 , 8 , 7 , 1 , 19 , 12 , 32 , 19 ,
  5 , 6 , 3 , 9 , 7 , 12 , 14 , 16 , 82 , 1 , 2 , 54 , 5 , 4 ,
  3 , 2 , 99 , 9 , 88 , 8 , 99 , 5 , 3 , 6 , 6 , 7 , 5 , 4 , 3 ,
  99 , 88 , 77 , 66 , 55 , 44 , 88 , 99 , 44 , 66 , 77 , 88 ,
  11 , 12 , 13 , 77 , 55 , 5 , 6 >
```

and reduce it in one pass by setting the reduction counter to some generous value, say `9999999`, which by all means suffices to have the computation terminate. Almost instantly, π–RED will respond with the sorted list

```
< 0 , 0 , 1 , 1 , 1 , 1 , 2 , 2 , 2 , 2 , 2 , 3 , 3 , 3 , 3 , 3 ,
  3 , 3 , 3 , 3 , 3 , 3 , 4 , 4 , 4 , 4 , 4 , 4 , 5 , 5 , 5 , 5 ,
  5 , 5 , 5 , 5 , 5 , 5 , 5 , 5 , 6 , 6 , 6 , 6 , 6 , 6 , 7 , 7 ,
  7 , 7 , 7 , 8 , 8 , 8 , 9 , 9 , 9 , 9 , 11 , 12 , 12 , 12 , 12 ,
  12 , 12 , 13 , 14 , 16 , 19 , 19 , 21 , 23 , 23 , 32 , 32 , 34 ,
  34 , 43 , 44 , 44 , 44 , 45 , 45 , 45 , 45 , 45 , 45 , 46 , 54 ,
  54 , 54 , 54 , 55 , 55 , 56 , 56 , 56 , 56 , 56 , 56 , 65 , 65 ,
  66 , 66 , 67 , 76 , 76 , 76 , 77 , 77 , 77 , 82 , 86 , 87 , 87 ,
  87 , 88 , 88 , 88 , 88 , 88 , 88 , 98 , 98 , 98 , 98 , 98 , 98 ,
  98 , 98 , 99 , 99 , 99 , 99 , 99 , 99 , 99 , 343 , 6532 , 87542 > ,
```

after having actually performed `8782` reduction steps.

6.4 Pattern Matching with Wild Cards

Sofar you have only learned about pattern matches that are strict in the sense that the argument terms must exactly match the structures specified in the patterns, which is to say that the arities of all matching nodes (which are the list constructors) must be exactly the same. However, when dealing with n-ary lists (sequences), this requirement is often too restrictive to specify real-life applications in an elegant and concise way. For instance, you may wish to search through a sequence of expressions for the occurrence of a particular substructure but you neither know nor are interested in its exact position (it may not even occur at all), and you also don't care about the other components of the sequence. Or, you may wish to identify in a sequence of variable length one or several substructures, modify or remove them, and do something else with the remaining fragments.

KIR supports these applications by means of patterns that may include so-called wild cards. Syntactically, these are atomic pattern components (or leaf nodes) just like variables or constants. However, semantically they are to match against subsequences of varying lengths, depending on the pattern context in which they occur on the one hand, and on the particular shapes of the arguments on the other hand. Wild cards may or may not be combined with pattern variables, in which case matching subsequences of the arguments are, as before, substituted for occurrences of these variables in the respective body expressions.

The following wild cards are available or scheduled for a later release of KIR (the latter are marked by a ‡):

- `.` or `Var` which matches against a subsequence of exactly one item length;
- `...` or `as ... Var` which covers subsequences of minimal lengths, including empty sequences;
- `.+`‡ or `as .+ Var`‡ which covers subsequences of minimal lengths, excluding empty sequences;
- `.*`‡ or `.* Var`‡ which covers subsequences of maximal lengths, including empty sequences.

As you may have guessed, the construct `as wild_card Var` is the one that substitutes the matching subsequences for occurrences of the variable `Var`.

Some simple examples may help you to get an idea as to what can be done with these wild cards.

The first one is to demonstrate, in several variants, which parts of a sequence are matched by the various wild cards.

```
ap case
   when < ... , < U , V > , as ... P , < W , Z > , ... >
    guard true do < < U , W , Z , V > , P >
   end
to [ < 1 , 2 , < 4 , 5 > , 3 , < 6 , 6 > , < 8 , 8 , 8 > , < 7 , 9 > , 4 , 3 > ]
```

Here you can expect a match of the following kind:

- the ...   matches against the first two elements of the sequence which are simply ignored;
- the `<U,V>` matches against the first substructure of two elements, which is `<4,5>`;
- the `as ...  P` matches against the shortest intervening subsequence that is between the last and the next substructure of two elements, which is just the `3`;
- the `<w,z>` matches agains the `<6,6>`, and
- the remaining ... matches against the rest of the sequence which, again, is ignored.

Thus, after one reduction step you get:

```
< < 4 , 6 , 6 , 5 > , < 3 > >
```

If you now change the wild card `as ...  P` to `as .* P` (assuming it would be available), it covers a subsequence of maximal length to the next occurrence of a subsequence with two elements (which is `<7,9>`):

```
ap case
   when < ... , < U , V > , as .* P , < W , Z > , ... >
    guard true do < < U , W , Z , V > , P >
   end
to [ < 1 , 2 , < 4 , 5 > , 3 , < 6 , 6 > , < 8 , 8 , 8 > , < 7 , 9 > , 4 , 3 > ]
```

This would reduce in one step to:

```
< < 4 , 7 , 9 , 5 > , < 3 , < 6 , 6 > , < 8 , 8 , 8 > > >
```

Another interesting variant is this one:

```
ap case
   when < .* , < U , V > , as ... P , < W , Z > , ... >
    guard true do < < U , W , Z , V > , P >
   end
to [ < 1 , 2 , < 4 , 5 > , 3 , < 6 , 6 > , < 8 , 8 , 8 > , < 7 , 9 > , 4 , 3 > ]
```

Here we have in the first pattern position a wild card requiring a top-level subsequence of maximal length to a substructure of two elements which, however, must be followed by another subsequence after which there must a second substructure of two elements. This is generally not easy to figure out, requiring a lot of backtracking and retrying, until the correct solution is found. However, the pattern matching facilities of π–RED will soon be smart enough to handle this and return as the correct solution the one that matches `<U,V>` against `<6,6>` and `<W,Z>` against `<7,9>`:

```
< < 6 , 7 , 9 , 6 > , < < 8 , 8 , 8 > > >
```

You may play with other variants of this example for yourself, e.g., with more than just two strict sub-patterns and different wild cards in between, in order to develop a better understanding of how this form of pattern matching works.

A last example which shows how pattern matching with wild cards can be used in the context of recursive functions is a program which tests a sequence of integer values for the palindrome property, meaning that reading the sequence from left to right is the same as reading it from right to left. This problem can be specified as follows:

```
def
  Pal [ Sequ ] =  ap case
                     when <> guard true do 'palindrome' ,
                     when < . > guard true do 'palindrome' ,
                     when < U , as ... W , V > guard ( U eq V ) do Pal [ W ]
                     otherwise 'no palindrome'
                     end
                  to [ Sequ ]
in Pal [ < 1 , 2 , 3 , 4 , 4 , 3 , 2 , 1 > ]
```

The `case` construct in the body of the function `Pal` basically does the following: if the argument sequence substituted for `Sequ` is either empty or contains just one element, then the sequence is a palindrome and we are done. If it contains more than one element, the third `when` clause extracts the first and the last element, compares them for equality and, if this is the case, applies the function `Pal` recursively to the subsequence without the first and the last element. If none of the patterns matches, then the sequence is no palindrome.

As you reduce this program step-by-step, you will note that in cycles of four steps (which it takes to complete recursive calls of `Pal`) π–RED returns the initial program with the actual first and last elements cut off the argument sequence, until the sequence is empty and you get the string `'palindrome'` as result.

6.5 Pattern Matching with User-Defined Constructors

You may have askes yourself why pattern matching as introduced in the preceding section is confined to list structures, but is not applicable to other non-atomic KIR expressions, e.g., applications or user-defined functions (abstractions). Though one can immediately think of many interesting applications of such a feature, there is a very simple and good reason for not supporting it: it would render the results of computations dependent on execution orders and thus violate the Church-Rosser property and also referential transparency. As a consequence, we would end up having a rather versatile term rewriting system, but not anymore a purely functional reduction system.

In order to 'fake' at least syntactically the applicability of pattern matching to objects other than just lists (sequences), KIR supports user-specified constructors. They may be used to construct expressions of a meta-level language that sits on top of KIR and can be interpreted only by rewrite rules specified as pattern matches. This feature, among many other applications, greatly facilitates rapid prototyping and testing of abstract machines, compilation schemes, type checkers etc. for other programming language, imperative or declarative (including KIR itself).

A meta-level expression generally has the syntactical form:

`Constr{Expr_1 , Expr_2 , ... , Expr_n}`

where `Constr` is a character string beginning with a letter which, in conjunction with the curly brackets { and } that enclose the sequence of n KIR expressions `Expr1` , ..., `Expr_n`, is taken as an n-ary user-specified constructor. In particular, the KIR expressions may be recursively constructed from other meta-level expressions. Note that the constructor symbol, though the editor converts its first letter to upper case, is not treated as a variable, and that the sequence within the brackets may be empty, in which case you have a 0-ary constructor.

A pattern that matches against such an expression must feature the same constructor name and the same arity[23], but may only have legitimate pattern components in the places of the expressions. These are variables, constants, wild cards, type specifications, and recursively non-atomic patterns made up from ordinary list constructors or user-specified constructors.

The following is a simple example which shows how pattern matching with user-specified constructors works:

```
ap case
   when C_1{< U , V > , 'abc' , C_2{W}} guard true do C_2 [ U [ V , W ] ]
   end
to [ C_1{< + , 6 > , 'abc' , C_2{4}} ]
```

[23]except in those cases where the pattern contains wild card synbols

Here the pattern obviously matches against the argument term in all components that matter, i.e., the application reduces to the body of the `when` clause in which the variables `U, V, W` are replaced by `+, 6, 4`, respectively:

```
C_2 [ ( 6 + 4 ) ] .
```

One more reduction step yields `C_2 [ 10]` .

You may convince yourself that changing the names of the user-specified constructors or of their arities either in the pattern or in the argument (but not identically in both (!)) leads to a mismatch, i.e., the application itself is returned as its normal form.

The particular syntax for meta-level expressions and for the corresponding patterns, of course, is more or less syntactic sugar. The same ends may be achieved by expressions of the form

$$< \text{'}\mathtt{constr}\text{`} \, , \, \mathtt{Expr\_1} \, , \, \ldots \, , \, \mathtt{Expr\_n} >$$

and equivalent patterns, in which the user-introduced constructors are represented as ordinary list constructors in combination with dedicated character strings (i.e., constants) in the first (or any other) index positions of the respective sequences. In fact, π–RED uses internally this representation as it reduces exactly as desired. There is one significant difference to ordinary lists though: structures made up from user-defined constructors can only be re-structured by pattern matching; primitive structuring functions are not applicable to them.

6.6 A Case Study : the SECD Machine

Let us now look at an interpreter for Landin's SECD machine [Lan64] as a perfect example to demonstrate the expressive power of pattern matching with both ordinary lists and user-defined constructs. In its original form, the SECD machine was conceived as an abstract applicative order evaluator for expressions of a pure λ-calculus. It derives its name from the four data structures it uses as a run-time environment. These data structures are operated as push-down stacks, of which

stack S holds evaluated λ-terms in essentially the order in which they are encountered during the traversal of the original λ-expression;

stack E is an environment stack which holds name/value pairs representing instantiations of free occurrences of λ-bound variables;

stack C is a control structure which accommodates the expression that still needs to be processed;

stack D serves as a dump for actual machine states that need to be saved when entering into the evaluation of a new 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 a current into a next machine state. To do so, the function τ must distinguish among six basic constellations on the stack-tops of components of (partially or fully evaluated) λ-terms. These constellations can be readily expressed in terms of pattern matches collected in a `case` construct. They include the occurrence of

- an empty control structure C in conjunction with a non-empty dump, in which case a new machine state must be retrieved from the dump;
- a variable on top of the control structure, in which case the associated value must be looked up in the environment E and pushed into the value stack S;
- a λ-abstraction on top of the control structure, in which case a closure representing its value in the actual environment E must be created on top of stack S;
- an applicator on top of C in conjunction with a closure on top of S, in which case a new name/value pair must be added to the environment E;
- an applicator on C in conjunction with any other term on on top of S, in which case the value of applying the topmost to the second-to-top item on stack S must be computed and pushed into S;
- an application on top of the control structure which, in compliance with the applicative order regime, must be reconfigured on C so as to have its argument and function terms in this order precede an explicit applicator;

The λ-terms to be processed by the interpreter that we wish to specify for this machine may be defined in the following meta-syntax in order to distinguish them from the λ-terms of KIR proper:

$$\texttt{Term} \; \rightarrow \; \texttt{Var}\{\texttt{X}\} \mid \texttt{Lambda}\{\texttt{X} \, , \, \texttt{Term}\} \mid \texttt{Apply}\{\texttt{Term} \, , \, \texttt{Term}\} \; .$$

Thus, we consider only the terms of a pure λ-calculus, treating variables, abstractions and applications as special constructor expressions which in fact define a meta-language on top of KIR . In addition we have closures of the general form `Clos{ E, X, Expr}` which define `Expr` as the value to be substituted for occurrences of the variable `X` in the environment `E`.

The specification of the interpreter may be based on the representation of the run-time environment as a sequence of four subsequences which correspond to the four push-down stacks. The state transition function τ can then be implemented straightforwardly by means of the `case` construct shown below. It is composed of six pattern matches which realize the above rewrite rules.

As you can see, all rules straightforwardly map old into new states. The second rule is slightly more complicated than the others for it must call upon a recursive function `Search` to search the environment `E` for the value with which a variable appearing on top of the control structure is instantiated.

```
case
when < < X , S > , E , <> , < < Ss , Se , Sc , Sd > , Rest > >
 guard true
 do < < X , Ss > , Se , Sc , Sd > ,
when < S , E , < Var{X} , C > , D >
 guard true
 do let Val = def
                Search [ E ] =
                 let First = lselect( 1 , E )
                 in if empty( E )
                    then Var{X}
                    else if ( lselect( 1 , First ) eq X )
                         then lselect( 2 , First )
                         else Search [ lcut( 1 , E ) ]
              in Search [ E ]
    in < < Val , S > , E , C , D > ,
when < S , E , < Lam{X , Expr} , C > , D >
 guard true
 do < < Clos{E , X , Expr} , S > , E , C , D > ,
when < < Clos{Env , X , Expr} , < Hed , S > > , E , < Ap{ } , C > , D >
 guard true
 do < <> , ( < < X , Hed > > ++ Env ) , < Expr , <> > ,
          < < S , E , C , D > , D > > ,
when < < Hed , < Heta , Rest > > , E , < Ap{ } , C > , D >
 guard true
 do < < Eval [ Hed , Heta ] , Rest > , E , C , D > ,
when < S , E , < Apply{E_1 , E_2} , C > , D >
 guard true
 do < S , E , < E_2 , < E_1 , < Ap{ } , C > > > , D >
otherwise < Env , 'irreg termination' >
end
```

Since the function τ must be recursively applied to the run-time environment (or machine state) until both the control structure and the dump are empty, we simply define a recursive function `Secd` whose body includes another `case` construct composed of

- a `when` clause which handles the termination condition;
- an `otherwise` clause under which `Secd` is recursively applied to the new state (or environment) resulting from the application to the current state, represented by the variable `Env`, of the `case` construct that realizes τ.

We thus get the following complete program specification for the interpreter function `Secd`, in the body of which the abbreviated inner `case` construct is the one shown on the preceding page which realizes the state transition rules.

```
def
  Secd [ Env ] =
   ap case
      when < S , E , <> , <> >
       guard true
       do < S , E , <> , <> , Done >
      otherwise Secd [ ap case
                           ?
                           ?
                        to [ Env ] ]
      end
   to [ Env ]
in Secd
```

Let us now try to apply the interpreter function `Secd` to a concrete λ-expression which in KIR would look like this:

```
ap ap sub [ U ]
      in sub [ V ]
         in U [ V ]
to [ sub [ X ] in X , W ]
```

As can be easily verified, this application reduces in three steps to the free variable `W`.

In the special constructor syntax that can be understood by the interpreter, this application must be specified as:

```
Apply{Apply{Lam{'u`,Lam{'v`,Apply{Var{'u`},Var{'v`}}}} ,
            Lam{'x`,Var{'x`}}},Var{'w`}}                    .
```

It is important to note here that what are supposed to be concrete variable names in this syntax must be specified as constants, i.e., as character strings embedded in quotation marks. If you would not do this, they would become variables of KIR but not of the meta-level λ-terms that are to be interpreted and as such could get parasitically bound outside the intended context. Moreover, the interpreter must compare, as part of its second pattern match in the inner `case` construct, variable occurrences in the λ-terms of your special meta-language against matching entries in the environment structure E. This can be done only on the basis of instantiations with constant values of the pattern variables in terms of which this comparison is specified.

In order to interpret the above meta-language expression, it must be put into the position of the third subsequence of the argument sequence for the function `Secd` (which represents the control structure C), with all other subsequences initially empty:

```
< <> , <> ,
< Apply{Apply{Lam{'u`,Lam{'v`,Apply{Var{'u`},Var{'v`}}}} ,
              Lam{'x`,Var{'x`}}},Var{'w`}}, <> > ,     <> >
```

Thus, the complete program, ready for execution, is this (with the body of `Secd` again abbreviated):

```
ap def
     Secd [ Env ] =
      ap case
            ?
            ?
      to [ Env ]
   in Secd
to [ < <> , <> ,
     < Apply{Apply{Lam{'u`,Lam{'v`,Apply{Var{'u`},Var{'v`}}}} ,
                   Lam{'x`,Var{'x`}}},Var{'w`}}, <> > ,     <> > ]
```

Provided you didn't make any mistakes when editing this program, it returns after 96 reduction steps the correct solution in the form of the following environment structure:

```
< < Var{'w`} , <> > , <> , <> , <> , Done >
```

i.e., with the variable construct `Var{'w`}` as the sole entry of the value stack S, and with all other stacks empty.

You may alternatively try to work your way through a step-by-step execution of this program, but be warned that it may take quite a while for you to figure out what is going on in a particular situation and where the next transformation step is going to take place. Since intermediate programs cannot be completely visualized on the screen without many question marks, it is helpful to study the first few steps very carefully in order to be able to select as the actual cursor expressions the `case` applications which are to be reduced next and need to be inspected in detail.

As an interesting exercise, you could try to supplement this SECD machine interpreter, within the framework of an applied λ- calculus, by the data types `boolean`, `integer number` and `list`, including a complete set of primitive functions defined on these types (e.g., the ones you have available in KiR). This will enable you to write in this extended meta-language a few more interesting application programs and have them executed on your interpreter.

7 Vectors and Matrices as System-Supported Data Types

7.1 Problem Identification

All serious computational problems involve value-transforming and structuring operations on large structured objects, in scientific and engineering applications especially on arrays of numerical values. The proper representation of these objects and the efficient implementation of operations on them is a considerable problem in all functional languages and systems. Conceptually, all operations must create new objects rather than overwrite existing ones in order to keep the computations free of side effects. Copying repeatedly large data structures in which only a few entries have been modified is extremely costly in terms of both processing time and memory space expended.

The language KIR as you know it sofar is particularly ill equipped in this respect since it provides just the bare essentials for structuring operations on heterogeneously typed lists (sequences) of KIR expressions. The ensuing lack of expressive power often results in rather awkward programs even for simple problems which may inflict a considerable run-time complexity.

A typical example would be a KIR program for the multiplication of two matrices represented as lists of lists of numbers. With only the primitive (re-) structuring operations `lselect`, `lcut`, `lunite` or pattern matching available, this rather straightforward problem would turn out to be a rather sophisticated piece of artful programming.

Assuming that a first argument matrix of n rows and k columns is represented as a list of n sublists of k entries each, and a second argument matrix of k rows and m columns is represented as a list of k sublists of m entries each, this problem must be broken up into applications of the functions in two arguments

`Mm_prod` (for matrix-matrix product) which recursively pairs all inner lists (row vectors) of its first argument matrix with the entire second argument matrix, applies another function `Vm_prod` to them, and concatenates the resulting lists (which are supposed to be the rows of the product matrix);

`Vm_prod` (for vector-matrix product) which recursively pairs a list (vector) expected in its first argument position with all inner lists (column vectors) of the matrix representation expected in its second argument position,applies a third function `Vv_prod` to them, and concatenates the resulting values (which are to become the components of the row vectors of the product matrix);

`Vv_prod` (for vector-vector product) which recursively pairs the elements of the flat lists (vectors) expected in both of its argument positions, multiplies them and adds them up.

Thus, the entire matrix multiplication program would look like this:

```
def
  Mm_prod [ Ma_1 , Ma_2 ] =
   let X = lcut [ 1 , Ma_1 ] ,
       Y = Vm_prod [ lselect [ 1 , Ma_1 ] ,
                     Ma_2 ]
   in if empty [ X ]
      then < Y >
      else ( < Y > ++ Mm_prod [ X , Ma_2 ] ) ,
  Vm_prod [ Vec , Matr ] =
   let X = lcut [ 1 , Matr ] ,
       Y = Vv_prod [ Vec , lselect [ 1 , Matr ] ]
   in if empty [ X ]
      then < Y >
      else ( < Y > ++ Vm_prod [ Vec , X ] ) ,
  Vv_prod [ Vec1 , Vec2 ] =
   let X = lcut [ 1 , Vec1 ] ,
       Y = lcut [ 1 , Vec2 ] ,
       Z = ( lselect [ 1 , Vec1 ] * lselect [ 1 , Vec2 ] )
   in if and [ empty [ X ] ,
               empty [ Y ] ]
      then Z
      else ( Z + Vv_prod [ X , Y ] )
in Mm_prod [ < < 1 , 2 > ,
               < 3 , 4 > ,
               < 5 , 6 > > ,
             < < 5 , 6 > ,
               < 3 , 4 > ,
               < 1 , 2 > > ]
```

To transpose one of the argument matrices, if this should be necessary, requires another three function definitions of about the same complexity.

Executing this program in one shot returns, after 287 reductions, the correct product matrix

```
< < 17 , 11 , 5 > ,
  < 39 , 25 , 11 > ,
  < 61 , 39 , 17 > >          .
```

This rather formidable program primarily specifies the recursive decomposition of the argument matrices into their atomic components, embedded in the recursive construction of the resulting product matrix from other atomic components which are being generated by applications of the function `Vv_prod`. The multiplication of an $n * k$ matrix by a $k * m$ matrix thus calls n times the function `Mm_prod`, m times the function `Vm_prod` in each call of `Mm_prod`, and k times the function `Vv_prod` in each call of `Vm_prod`, i.e., we have altogether $n * m * k$ recursive function calls. On first sight, this computational complexity seems to be appropriate for the particular problem since it equals the number of elementary arithmetic operations (additions and multiplications) that need to be carried out. However, the inefficient and hence expensive part is due to the generation of these operations by means of recursive function applications, of which each involves the substitution of two formal by actual parameters, the evalution of an `if_then_else` clause, and applications of primitive function which decompose and assemble lists. Depending on implementation details, the complexities of the latter operations in both time and space consumption may be rather substantial.

Another problem with this program concerns the compatibility of both argument matrices w.r.t. their shapes and element types. If neither is the case, program execution may get stuck with the entire computation unfolded to the point where either the function `Vv_prod` has consumed one of its argument lists with further elements left in the other, in which case the shapes are not compatible[24], or any of the arithmetic operations is not applicable since the lists contain elements other than numbers (which syntactically is perfectly legitimate).

To see what happens in these cases, let's first replace the `1` in the first row of the first argument matrix by a character string `'abc‘`, in which case no arithmetic operations involving this element can be carried out due to incompatible types. Thus you get, after 281 reduction steps, the following structure[25]:

```
< < ( ( 'abc‘ * 5 ) + 12 ) ,
    ( ( 'abc‘ * 3 ) + 8 ) ,
    ( ( 'abc‘ * 1 ) + 4 ) > ,
  < 39 , 25 , 11 > ,
  < 61 , 39 , 17 > >
```

Now let's try incompatible shapes by just taking one element, the `1`, off the first row of the first matrix. Here is what you get as an incomplete result matrix after 266 reduction steps:

[24] The syntax of lists even renders it possible to have varying numbers of elements in each of the sublists (the row or column vectors) of a matrix representation

[25] It is easy to see why this reduction sequence took six steps less than the one that applied to the matrix which had in place of the string `'abc‘` a number: there are three multiplications and three additions left over that could not be done.

```
< < ( 10 + if ( empty( lcut( 1 , <> ) ) and true )
          then ( lselect( 1 , <> ) * 6 )
          else ( ( lselect( 1 , <> ) * 6 ) + ap def
                                                 ?
                                                 ?
                                            to [ .. ] ) ) ,
    ( 6 + if ( empty( lcut( 1 , <> ) ) and true )
         then ( lselect( 1 , <> ) * 4 )
         else ( ( lselect( 1 , <> ) * 4 ) + ap def
                                                ?
                                                ?
                                           to [ .. ] ) ) ,
    ( 2 + if ( empty( lcut( 1 , <> ) ) and true )
         then ( lselect( 1 , <> ) * 2 )
         else ( ( lselect( 1 , <> ) * 2 ) + ap def
                                                ?
                                                ?
                                           to [ .. ] ) ) > ,
  < 39 , 25 , 11 > ,
  < 61 , 39 , 17 > >
```

As you can see, the computation stops with a constant expression containing expanded function calls upon failing to extract a second element from the first row of the first argument matrix (by trying to cut off an element from a list that is already empty). Since the predicate terms of the **if_then_else** clauses thus cannot be reduced to Boolean constants, no further reductions take place in the respective alternative terms. Note that only the first row of the product matrix remains incomplete, whereas the other two rows are flawlessly computed as before.

The inconveniences of writing programs in this awkward form and of having them reduced to a large extent (but to no avail) in case of type incompatibilities of the argument objects led to the introduction in KiR of the composite (or array) data types `vector` and `matrix`. The basic ideas are more or less directly adopted from the programming language APL [Iver62] and the data type architectures proposed in [Abr70, GiGu82].

7.2 Re-Structuring Vectors and Matrices

Including in KiR composite data types greatly facilitates programming complex numerical application problems, making programs more concise and hence more comprehensible. It relieves the programmer of the tedious and error-prone burden of specifying operations on composite objects in terms of nested recursive functions (or iteration loops) which essentially prescribe their decomposition and (re-) construction, and it also does away, to a considerable degree, with the run-time complexity (particularly space consumption) thus inflicted . This is primarily due to the implementation of a variety of standardized (re-) structuring and value-transforming operations as system primitives of π–RED , applications of which are δ-reduced in one conceptual step. Most of the primitive functions are overloaded (or generic), i.e., they are applicable to any type- and shape- compatible combination of atomic values (scalars), vectors and matrices. Type- and shape-compatibility tests are done dynamically at run-time: the operations are carried out if and only if these tests succeed.

Before looking at anything else, you should make yourself familiar with the KiR representations of vectors and matrices and how they can be edited on your display screen. Though they are essentially based on lists (or sequences) of elements, there are things that are a little different. Whereas ordinary lists may have as components any of the legitimate KiR expressions (including lists) and thus are heterogeneously typed, vectors and matrices are composite objects whose elements are atomic and of the same basic (elementary) type such as decimal numbers, boolean values, and character strings. Moreover, matrices require that all its rows (or columns) have the same number of elements. Following these constraints, the editor rejects any attempts to introduce into certain positions of a partially specified structured object elements that either syntactically or because of their types don't belong there.

Vectors and matrices are distinguished from ordinary lists by special keywords which also define the types of the elements. These keywords are taken as the outermost constructors of the objects. For vectors they include

`vect` for a row vector of decimal numbers;

`bvect` for a row vector of boolean constants;

`svect` for a row vector of character strings;

and `tvect`, `btvect`, `stvect` for the respective transposed (or column) vectors. It is important to note that applications of dyadic value-transforming functions on vectors (and on matrices as well), as you will see a little later, can be reduced if and only if the constellation of row and column vectors (or matrices) complies with the respective rules of linear algebra.

You may edit a vector of numbers by typing in the input line `vect` which, upon hitting the enter key, will return to the FA field as `vect< # >`. The empty symbol may now be replaced with any sequence of numbers separated by commas, say `1,5,6,7`, yielding:

```
vect< 1 , 5 , 6 , 7 > .
```

Further elements may be inserted into this vector anywhere you like and in the same way you learned to do it with lists: you simply move the cursor under the element preceding the position of insertion and type `%,` followed by the sequence of numbers you wish to have sqeezed in there. For instance, with the cursor under the `1` and with `%, 2,3,4` as input you will get:

```
vect< 1 , 2 , 3 , 4 , 5 , 6 , 7 >  .
```

However, if you try to insert anything other than a number, say the string ’abc‘, it will not be accepted as a legitimate input, i.e., the vector in the FA field remains as before, and on the message line you will be notified that a number is expected instead.

The transposed version of this vector may be edited using the keyword `tvect`, which will give you in the FA field:

```
tvect< 1 ,
       2 ,
       3 ,
       4 ,
       5 ,
       6 ,
       7 >  .
```

The (re-) structuring functions that may be applied to vectors are essentially the same as those for lists. They are distinguished by a prefix letter `v` instead of `l`, i.e., we have `vselect`, `vcut`, `vrotate`, `vreplace`, `vunite`, with the same index parameters as for lists. In addition, there is a function `transpose` which returns the transpose of the actual vector.

Typical applications of these functions have the following effects:

- Selecting an element from a vector:

  ```
  vselect [ 3 , vect< 7 , 6 , 5 , 4 , 3 , 2 , 1 > ]
  ```

  returns the value `5`;

- Rotating a vector to the left:

  ```
  vrotate [ 3 , vect< 1 , 2 , 3 , 4 , 5 , 6 , 7 > ]
  ```

  returns

  ```
  vect< 4 , 5 , 6 , 7 , 1 , 2 , 3 >   ;
  ```

- Cutting elements off the trailing end of a vector:

```
vcut( ~2 , vect< 1 , 2 , 3 , 4 , 5 , 6 , 7 > )
```

returns[26]

```
vect< 1 , 2 , 3 , 4 , 5 >    ;
```

- Transposing a vector from column to row:

```
transpose [ tvect< 1 ,
                   2 ,
                   3 ,
                   4 ,
                   5 ,
                   6 ,
                   7 > ]
```

returns

```
vect< 1 , 2 , 3 , 4 , 5 , 6 , 7 >    ;
```

- Concatenating two vectors:

```
vunite [ tvect< 1 , 2 , 3 , 4 > , vect< 5 , 6 , 7 , 8 > ]
```

returns

```
vect< 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 >    ;
```

(Note that this function application does not work on vectors of which one is transposed but the other is not).

- Replacing an element:

```
vreplace [ 3 , 11 , vect< 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 > ]
```

returns

```
vect< 8 , 7 , 11 , 5 , 4 , 3 , 2 , 1 >    .
```

Legitimate keywords for (the construction of) two-dimensional matrices are

[26]Note that in the application of `vcut` the parameter value $\tilde{2}$ must be read as a negative number.

`mat` for matrices of decimal numbers;

`bmat` for matrices of boolean constants;

`smat` for matrices of character strings;

`mmat` for matrices of matrices.

A matrix of numbers may be edited by typing in the input line `mat`, which will be echoed in the FA field as:

```
mat
< # >.                .
```

The keyword `mat` is here taken as the outermost constructor of the matrix, and the `#` is the placeholder for its first row. So, if you type in the input line, say, `3,2,1` with the cursor underneath the `#`, you will get this:

```
mat
< 3 , 2 , 1 >.                .
```

As long as you have only one row in your matrix, you may add further entries as you like, e.g., if you type `5,4,%` with the cursor underneath the `3`, you get:

```
mat
< 5 , 4 , 3 , 2 , 1 >.                .
```

Likewise, you may delete any number of elements from this structure.

Now let's try to add more rows. You may do so by moving the cursor under the opening parenthesis `<` of the row that already exists and simply type as input something like `%,< 6,`. It will show up in the FA field as:

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< 6 , # , # , # , # >.                .
```

Something interesting has happened here: even though the second row was incompletely specified, the editor has laid out its complete structure, taking the number of elements of the first row as the row length for the entire matrix. This length is fixed from now on, i.e., if you try to add or delete elements to or from a row, you will not succeed. There are only two things that you can do: fill out the empty symbols of incompletely specified rows, and add or delete entire rows since only the number of columns remains variable. You are invited to check this out by adding another two empty rows, say following the first row, and by immediately deleting one of them so that, in the end, you get this:

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< # , # , # , # , # > ,
< 6 , # , # , # , # >.              .
```

Now you should complete the matrix by inserting numbers into the placeholder positions in any way you prefer it, say thus:

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 4 , 5 > ,
< 6 , 7 , 8 , 9 , 0 >.    ,
```

and store it away on an `edit` file so that you have available some vehicle to play around with.

The (re-) structuring operations that are applicable to matrices are essentially the same as those for lists and vectors. The function names have to be pre-fixed with an `m` instead of `l` or `v`, i.e., you may use `mselect`, `mcut`, `mrotate`, `mreplace` and `munite`. There is one significant difference though: applications of these functions usually require two index parameters to specifies how the operations are to be performed along both matrix coordinates. Thus, the general syntax of applications of these functions is:

`mfunc[index_1, index_2, matrix]`,

with `index_1` and `index_2` specifying row and column indices, respectively.

Here is a number of representative examples for what these functions do to the above matrix:

- Selecting an element from the matrix:

  ```
  select( 3 , 4 , mat
                  < 5 , 4 , 3 , 2 , 1 > ,
                  < 1 , 2 , 3 , 4 , 5 > ,
                  < 6 , 7 , 8 , 9 , 0 >. )
  ```

  returns the element `9` of the third row and the fourth column;

- Cutting off columns:

```
mcut [ 0 , 2 , mat
               < 5 , 4 , 3 , 2 , 1 > ,
               < 1 , 2 , 3 , 4 , 5 > ,
               < 6 , 7 , 8 , 9 , 0 >. ]
```

  returns

```
mat
< 3 , 2 , 1 > ,
< 3 , 4 , 5 > ,
< 8 , 9 , 0 >.       ;
```

- Rotating the rows of the matrix:

```
mrotate [ ~1 , 0 , mat
                   < 5 , 4 , 3 , 2 , 1 > ,
                   < 1 , 2 , 3 , 4 , 5 > ,
                   < 6 , 7 , 8 , 9 , 0 >. ]
```

  returns

```
mat
< 6 , 7 , 8 , 9 , 0 > ,
< 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 4 , 5 >.       ;
```

- Rotating the matrix in one step both along rows and columns:

```
mrotate [ ~1 , 2 , mat
                   < 5 , 4 , 3 , 2 , 1 > ,
                   < 1 , 2 , 3 , 4 , 5 > ,
                   < 6 , 7 , 8 , 9 , 0 >. ]
```

  returns

```
mat
< 8 , 9 , 0 , 6 , 7 > ,
< 3 , 2 , 1 , 5 , 4 > ,
< 3 , 4 , 5 , 1 , 2 >.       ;
```

- Replacing an element of the matrix:

```
mreplace [ 2 , 4 , 11 , mat
                        < 5 , 4 , 3 , 2 , 1 > ,
                        < 1 , 2 , 3 , 4 , 5 > ,
                        < 6 , 7 , 8 , 9 , 0 >. ]
```

returns

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 11 , 5 > ,
< 6 , 7 , 8 , 9 , 0 >.       ;
```

- Replacing entire columns and rows can be accomplished by two variants of `mreplace` which require different parameter specifications:

```
mreplace_c( 3 , tvect< 4 ,
                       4 ,
                       4 > , mat
                             < 5 , 4 , 3 , 2 , 1 > ,
                             < 1 , 2 , 3 , 4 , 5 > ,
                             < 6 , 7 , 8 , 9 , 0 >. )
```

replaces the third column of the matrix by the column vector given in the second argument position, returning:

```
mat
< 5 , 4 , 4 , 2 , 1 > ,
< 1 , 2 , 4 , 4 , 5 > ,
< 6 , 7 , 4 , 9 , 0 >.     ;
```

Likewise,

```
mreplace_r [ 2 ,
             vect< 6 ,  6 , 6 , 6 , 6 > ,
             mat
             < 5 , 4 , 3 , 2 , 1 > ,
             < 1 , 2 , 3 , 4 , 5 > ,
             < 6 , 7 , 8 , 9 , 0 >. ]
```

replaces the second row of the matrix with the row vector specified as second argument and returns:

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< 6 , 6 , 6 , 6 , 6 > ,
< 6 , 7 , 8 , 9 , 0 >..      .
```

It is important to note that the objects to be substituted must be specified as vectors of the same lengths, element types and orientations as the ones that are to be replaced. No other structures will be accepted as legitimate arguments, i.e., the applications could not be reduced in these cases.

- Concatenating two matrices (in this case just for convenience a matrix with itself) along rows must be specified as:

```
munite( 1 , mat
            < 5 , 4 , 3 , 2 , 1 > ,
            < 1 , 2 , 3 , 4 , 5 > ,
            < 6 , 7 , 8 , 9 , 0 >. , mat
                                     < 5 , 4 , 3 , 2 , 1 > ,
                                     < 1 , 2 , 3 , 4 , 5 > ,
                                     < 6 , 7 , 8 , 9 , 0 >. )
```

which returns

```
mat
< 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 4 , 5 > ,
< 6 , 7 , 8 , 9 , 0 > ,
< 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 4 , 5 > ,
< 6 , 7 , 8 , 9 , 0 >.      ;
```

and concatenating it with itself along columns must be specified as:

```
munite( 2 , mat
            < 5 , 4 , 3 , 2 , 1 > ,
            < 1 , 2 , 3 , 4 , 5 > ,
            < 6 , 7 , 8 , 9 , 0 >. , mat
                                     < 5 , 4 , 3 , 2 , 1 > ,
                                     < 1 , 2 , 3 , 4 , 5 > ,
                                     < 6 , 7 , 8 , 9 , 0 >. )
```

and returns

```
mat
< 5 , 4 , 3 , 2 , 1 , 5 , 4 , 3 , 2 , 1 > ,
< 1 , 2 , 3 , 4 , 5 , 1 , 2 , 3 , 4 , 5 > ,
< 6 , 7 , 8 , 9 , 0 , 6 , 7 , 8 , 9 , 0 >.    .
```

Note that the first parameter must have the value 1 for row concatenation, and the value 2 for column concatenation; no other value in this position is within the domain of `munite`. Of course, concatenation succeeds if and only if the two argument matrices have the same element types and the same dimension along the direction in which they are to be glued together.

- Transposing a matrix, as in the case of vectors, is denoted as:

```
transpose [ mat
            < 5 , 4 , 3 , 2 , 1 , 5 , 4 , 3 , 2 , 1 > ,
            < 1 , 2 , 3 , 4 , 5 , 1 , 2 , 3 , 4 , 5 > ,
            < 6 , 7 , 8 , 9 , 0 , 6 , 7 , 8 , 9 , 0 >. ]
```

and returns

```
mat
< 5 , 1 , 6 > ,
< 4 , 2 , 7 > ,
< 3 , 3 , 8 > ,
< 2 , 4 , 9 > ,
< 1 , 5 , 0 > ,
< 5 , 1 , 6 > ,
< 4 , 2 , 7 > ,
< 3 , 3 , 8 > ,
< 2 , 4 , 9 > ,
< 1 , 5 , 0 >.
```

.

7.3 Value-Transforming Operations

Most of the primitive value-transforming functions are generic: they may be applied to (any compatible combination of) scalars (atomic data), vectors and matrices.

Let us first consider dyadic arithmetic operations. Applications in infix notation have the general form

(`operand_2 arith_op operand_1`) ,

in which case you must use the symbols `+`, `-`, `*` and `/` as substitutes for the operator `arith_op`. Alternatively, you may specify the applications as

`arith_op[ operand_1, operand_2]` ,

in which case the operators must be denoted as `plus`, `minus`, `mult` or `div`.

The reduction rules for these applications are roughly as follows:

- if both operands are scalar values of the same type, then the operation is performed as usual, with another scalar of the same type as result;
- if `operand_2` is a matrix or a vector and `operand_1` is a scalar of the same type, then the result is another matrix or vector, respectively, of the same size whose elements are computed by pairing the scalar value of `operand_1` with all elements of `operand_2`;
- if `operand_1` is a matrix or a vector and `operand_2` is a scalar, then we have the complementary operation with the scalar value and the elements of the composite object in each pair interchanged;
- if both operands are either matrices or vectors of the same sizes and element types, then the result is another matrix or vector of that size whose elements are computed by pairing all elements with identical row and column indices from both operand objects.

The following examples may help to illustrate this:

- Dividing a scalar by all elements of a matrix:

  ```
  ( 2 / mat
        < 2 , 4 , 6 , 8 > ,
        < 1 , 3 , 5 , 7 >. )
  ```

  reduces to another matrix

  ```
  mat
  < 1 , 0.5 , 0.33333 , 0.25 > ,
  < 2 , 0.66666 , 0.4 , 0.28571 >.      ;
  ```

- Doing the same with both operands interchanged, i.e., dividing all elements of a matrix by a scalar:

  ```
  ( mat
    < 2 , 4 , 6 , 8 > ,
    < 1 , 3 , 5 , 7 >. / 2 )
  ```

  yields

  ```
  mat
  < 1 , 2 , 3 , 4 > ,
  < 0.5 , 1.5 , 2.5 , 3.5 >.      ;
  ```

- Dividing two matrices elementwise:

```
( mat
  < 4 , 8 , 12 , 16 > ,
  < 2 , 6 , 10 , 14 >. / mat
                        < 2 , 4 , 6 , 8 > ,
                        < 1 , 3 , 5 , 7 >. )
```

results in

```
mat
< 2 , 2 , 2 , 2 > ,
< 2 , 2 , 2 , 2 >.      .
```

There are three more dyadic functions available which operate in essentially the same way on scalars, vectors and matrices:

`mod` computes (elementwise) the remainder of the second operand after division by the first operand;
example:

```
( mat
  < 4 , 5 , 6 > ,
  < 9 , 8 , 7 >. mod mat
                    < 2 , 2 , 4 > ,
                    < 4 , 2 , 7 >. )
```

reduces in one step to

```
mat
< 0 , 1 , 2 > ,
< 1 , 0 , 0 >.      ;
```

`max` computes (elementwise) the maximum numbers of both operands;
example:

```
( mat
  < 4 , 5 , 6 > ,
  < 9 , 8 , 7 >. max mat
                    < 7 , 6 , 4 > ,
                    < 4 , 9 , 7 >. )
```

reduces in one step to

```
mat
< 7 , 6 , 6 > ,
< 9 , 9 , 7 >.      ;
```

`min` computes (elementwise) the minimum numbers of both operands; example:

```
( mat
  < 4 , 5 , 6 > ,
  < 9 , 8 , 7 >. min mat
                   < 7 , 6 , 4 > ,
                   < 4 , 9 , 7 >. )
```

reduces to

```
mat
< 4 , 5 , 4 > ,
< 4 , 8 , 7 >.        .
```

The most powerful primitive function is the one for the direct computation of inner products of composite objects with compatible shapes and element types. The syntax of its application is

`ip [ operand_1, operand_2]` .

Here are some typical applications:

- Vector-vector products:

```
ip( vect< 4 , 5 , 6 , 7 > , tvect< 1 ,
                                   2 ,
                                   3 ,
                                   4 > )
```

reduces in one step to `60`;

```
ip( tvect< 4 ,
           5 ,
           6 ,
           7 > , vect< 1 , 2 , 3 , 4 > )
```

reduces in one step to

```
mat
< 4 , 8 , 12 , 16 > ,
< 5 , 10 , 15 , 20 > ,
< 6 , 12 , 18 , 24 > ,
< 7 , 14 , 21 , 28 >.        .
```

Note that one of the operand vectors must be transposed according to the rule books, otherwise the application is irreducible.

- Vector-matrix products:

```
ip [ vect< 3 , 2 , 1 > , mat
                         < 4 , 8 , 12 , 16 > ,
                         < 5 , 10 , 15 , 20 > ,
                         < 6 , 12 , 18 , 24 >. ]
```

reduces to `vect< 28 , 56 , 84 , 112 >`.

```
ip( mat
    < 4 , 8 , 12 , 16 > ,
    < 5 , 10 , 15 , 20 > ,
    < 6 , 12 , 18 , 24 >. , tvect< 1 ,
                                   2 ,
                                   3 ,
                                   4 > )
```

reduces to

```
tvect< 120 ,
       150 ,
       180 >    .
```

- A matrix-matrix product:

```
ip( mat
    < 4 , 8 , 12 , 16 > ,
    < 5 , 10 , 15 , 20 > ,
    < 6 , 12 , 18 , 24 >. , transpose [ mat
                                        < 4 , 8 , 12 , 16 > ,
                                        < 5 , 10 , 15 , 20 > ,
                                        < 6 , 12 , 18 , 24 >. ] )
```

reduces in one step to

```
ip( mat
    < 4 , 8 , 12 , 16 > ,
    < 5 , 10 , 15 , 20 > ,
    < 6 , 12 , 18 , 24 >. , mat
                            < 4 , 5 , 6 > ,
                            < 8 , 10 , 12 > ,
                            < 12 , 15 , 18 > ,
                            < 16 , 20 , 24 >. )      ,
```

and in another step to

```
mat
< 480 , 600 , 720 > ,
< 600 , 750 , 900 > ,
< 720 , 900 , 1080 >.        .
```

There is another interesting set of primitive functions available in KIR which for arithmetic operations on vectors and matrices has the effect of the higher-order primitive `fold_right` that is available in many other functional languages: it drives an arithmetic operator from left to right into a vector or simultaneously into all row and column vectors of a matrix, applying it recursively to the first element and to the rest of the vector(s).

The syntax for applications of these functions is

$$\texttt{vc\_func[vect} < \ldots >]$$

for vectors, and

$$\texttt{c\_func[index}\,,\, \texttt{mat} < \ldots ,>, \ldots < \ldots > .]$$

for martrices, in which case `index = 1` specifies folding along columns and `index = 2` specifies folding along rows. `func` is a non-terminal which stands for either of the arithmetic function symbols `plus`, `minus`, `mult` or `div`.

The reduction of these applications follows the general scheme

$$\texttt{vc\_func[vect} < \texttt{a\_1}, \ldots , \texttt{a\_n} >] \rightarrow$$
$$(\texttt{a\_1 vc\_func (a\_2 vc\_func (} \ldots \texttt{(a\_(n} - \texttt{1) vc\_func a\_n))} \,.$$

Here are some typical examples which illustrate these operations:

- Addition along a vector:

  ```
  vc_plus( vect< 16 , 8 , 4 , 2 , 1 > )
  ```

  reduces in one step to `31` ;

- Substraction along the same vector:

  ```
  vc_minus( vect< 16 , 8 , 4 , 2 , 1 > )
  ```

  reduces in one step to `11` ;

- Division along the columns of a matrix:

```
c_div [ 1 , mat
            < 16 , 8 , 4 , 2 > ,
            < 8 , 4 , 2 , 1 > ,
            < 4 , 2 , 1 , 1 > ,
            < 1 , 1 , 1 , 1 >. ]
```

reduces in one step to `vect< 8 , 4 , 2 , 2 >`;

- Division along the rows of the same matrix:

```
c_div [ 2 , mat
            < 16 , 8 , 4 , 2 > ,
            < 8 , 4 , 2 , 1 > ,
            < 4 , 2 , 1 , 1 > ,
            < 1 , 1 , 1 , 1 >. ]
```

reduces to

```
tvect< 4 ,
       4 ,
       2 ,
       1 >    .
```

There are two more unary functions applicable to scalars, vectors and matrices of numbers which may be of interest:

`neg` multiplies all elements by `~1` (minus one);

`abs` returns all elements as absolute values.

The relational operators of KIR are defined over scalar values, vectors and matrices too. The element types may be decimal numbers or character strings (in which case all comparisons are made based on lexical orders). As far as composite objects are concerned, the applicability of these operators follows essentially the same rules as for dyadic arithmetic operators: a scalar value in one operand position is paired wih all elements of a composite object in the other argument position, and composite objects in both operand positions must have the same shapes so that elements of identical index positions can be paired.

Legitimate relational operators are

`eq | ne` which test both operands over all elements for equality or inequality, respectively;

`f_eq | f_ne` which test both operands elementwise for equality or inequality, respectively;

`lt | le` which test whether or not the (elements of the) first operand are less than or less/equal (the elements of) the second operand, respectively;

`gt | ge` which tests whether or not the (elements of the) first operand are greater or greater/equal (the elements of) the second operand, respectively,

and return as results (vectors or matrices of) boolean values.

Here are a few examples of how this works:

- Comparison of a decimal number against a vector of numbers:

```
le( 3 , vect< 5 , 4 , 3 , 2 , 1 > )
```

  reduces to

```
bvect< true , true , true , false , false >    ;
```

- Comparison of two vectors of numbers:

```
gt( vect< 1 , 2 , 3 , 4 , 5 > , vect< 5 , 4 , 3 , 2 , 1 > )
```

  reduces to

```
bvect< false , false , false , true , true >    ;
```

- Comparison of two vectors of character strings:

```
lt( stvect< 'abc` ,
            'cde` ,
            'uvw` ,
            'xyz` > , stvect< 'ab` ,
                              'cde` ,
                              'uvwx` ,
                              'zyx` > )
```

  reduces to

```
btvect< false ,
        false ,
        true ,
        true >    ;
```

- Comparison of two vectors of strings for equality over all elements:

```
eq( svect< 'abc' , 'cde' , 'uvw' , 'xyz' > ,
           svect< 'ab' , 'cde' , 'uvwx' , 'zyx' > )
```

reduces to `false` ;

- Elementwise comparison of the same vectors for inequality:

```
f_ne( svect< 'abc' , 'cde' , 'uvw' , 'xyz' > ,
             svect< 'ab' , 'cde' , 'uvwx' , 'zyx' > )
```

reduces to

```
bvect< true , false , true , true >     .
```

The logical operators **and**, **or**, **xor** and **not** may be used in the same way on atomic or composite objects of boolean element type.

7.4 Query Functions and Class Conversion

KIR also supports a number of query functions which are primarily intended to return structural and type information about composite objects, but one of them may also be used to classify other objects of the language. These functions include

`class` which, when applied to the respective KIR objects, returns as values `scalar`, `vector`, `matrix`, `list`, or `function`[27];

`type` which, when applied to scalars, vectors or matrices, returns their element types;

`ldim | vdim | mdim` which, when applied to lists, vectors, or matrices, respectively, returns their dimensions.

All other appplications of these query functions are irreducible.

Thus, when applying to the matrix

```
mat
< 4 , 8 , 12 , 16 > ,
< 5 , 10 , 15 , 20 > ,
< 6 , 12 , 18 , 24 >.
```

[27]It is important to note that there is no class `application` for this would mean that we have a test which, depending on reduction orders, could have two different outcomes and thus would violate the Church-Rosser property and referential transparency: if you have an application as argument of `class` then you may get the class `application` in response, if – for one reason or another – it cannot be reduced, or something else if it is being reduced before applying the function `class` to it.

the functions `class`, `type`, `mdim`, you should get as responses `matrix`, `number`, `(3,4)`, respectively.

Finally, KiR provides a set of cast functions by which composite objects of appropriate shape and element type may be converted into objects of other classes (or types). The following functions are available:

`to_scalar` converts a vector or a matrix of just one element into a scalar;

`to_vector` converts a matrix of just one row vector or a scalar into a vector;

`to_tvector` converts a matrix of just one column vector or a scalar into a transposed vector;

`to_matrix` converts a scalar, a vector or a transposed vector into a matrix of appropriate shape;

`transform` converts a matrix into a flat list of elements by concatenating its rows in ascending order;

`ltransform` , when applied as `ltransform[index_1, index_2, list]`, converts a flat list substituted for `list` into a matrix of `index_1` rows and `index_2` columns by putting consecutive subsequences of `index_2` elements from `list` in ascending order into rows.

Applications to objects other than those listed above are irreducible and therefore remain unchanged.

Here are some examples:

- Converting a vector into a matrix:

  ```
  to_mat [ tvect< 1 ,
                  2 ,
                  3 ,
                  4 ,
                  5 > ]
  ```

  reduces to

  ```
  mat
  < 1 > ,
  < 2 > ,
  < 3 > ,
  < 4 > ,
  < 5 >.       ;
  ```

- Converting a matrix into a flat list:

```
transform [ mat
            < 4 , 8 , 12 , 16 > ,
            < 5 , 10 , 15 , 20 > ,
            < 6 , 12 , 18 , 24 >. ]
```

reduces to

```
< 4 , 8 , 12 , 16 , 5 , 10 , 15 , 20 , 6 , 12 , 18 , 24 >     ;
```

- Converting a flat list into a matrix:

```
ltransform [ 6 , 2 , < 4 , 8 , 12 , 16 , 5 , 10 , 15 ,
                                20 , 6 , 12 , 18 , 24 > ]
```

reduces to

```
mat
< 4 , 8 > ,
< 12 , 16 > ,
< 5 , 10 > ,
< 15 , 20 > ,
< 6 , 12 > ,
< 18 , 24 >.        .
```

The `transform` and `ltransorm` functions in conjunction with the `to_...` functions basically allow you to convert any composite object into any other composite object, though not always very elegantly. Considerable difficulties arise with conversions into lists, and to some extent also from lists into objects of other classes. There is as yet no direct way of converting a vector into a list (you first have to convert it into a matrix), and – what is particularly annoying – you cannot directly convert a matrix into a nested list of the same shape. You can only convert it into a flat list, from which you must re-construct the original shape by means of a defined function which requires as additional parameters the dimensions extracted from the matrix you started with. Thus, what is needed are functions that convert vectors and matrices into lists (of lists) of the same shape, and convert lists of appropriate shapes and element types into vectors or matrices. They are not yet available in KiR , but we are working on it.

Appendix 1: The Syntax of KiR

This appendix gives a reasonably complete formal definition of the syntax of KiR in the usual Backus/Naur notation. However, some of the production rules which exhaustively break down everything into nuts and bolts details will just be explained verbally. To avoid confusion with the delimiters `<` and `>` of lists (sequences) of the syntax proper, we use the delimiters $\langle$: and :$\rangle$ to enclose non-terminal symbols. Likewise, to enclose material which is to be repeated zero or several times, we use the curly brackets (again with colons) {: and :}.

The basic syntactic entities of which all KiR programs are constructed are expressions (or terms). Thus we have a start symbol $\langle$:`expr`:$\rangle$ from which derive the following non-terminals:

$$\begin{array}{lcl} \langle:\texttt{expr}:\rangle & \Longrightarrow & \langle:\texttt{binding\_expr}:\rangle \quad | \\ & & \langle:\texttt{control\_expr}:\rangle \quad | \\ & & \langle:\texttt{ap\_expr}:\rangle \quad | \\ & & \langle:\texttt{struct\_expr}:\rangle \quad | \\ & & \langle:\texttt{const\_expr}:\rangle \end{array}$$

The binding expressions that derive from $\langle$:`binding_expr`:$\rangle$ are essentially all those which define functions and the scopes of their formal parameters, including one which directly substitutes non-recursively defined variables by the values of expressions:

$$\begin{array}{lcl} \langle:\texttt{binding\_expr}:\rangle & \Longrightarrow & \langle:\texttt{recfunc\_def}:\rangle \quad | \\ & & \langle:\texttt{letvar\_def}:\rangle \quad | \\ & & \langle:\texttt{func\_def}:\rangle \quad | \\ & & \langle:\texttt{pattern\_match}:\rangle \end{array}$$

The control expressions break down into `then_else` clauses[28] and `case` constructs:

$$\begin{array}{lcl} \langle:\texttt{control\_expr}:\rangle & \Longrightarrow & \texttt{then}\ \langle:\texttt{expr}:\rangle\ \texttt{else}\ \langle:\texttt{expr}:\rangle \quad | \\ & & \texttt{case}\ \langle:\texttt{pattern\_match}:\rangle\ \{:,\ \langle:\texttt{pattern\_match}:\rangle\ :\} \\ & & \{:\ \texttt{otherwise}\ \langle:\texttt{expr}:\rangle\ :\}\ \texttt{end} \end{array}$$

Note here that the `otherwise` expression, contrary to the general definition of the brackets {: and :}, may appear at most once in a `case` expression.

The applicative expressions come in four alternative forms which, depending on the number of arguments and the class (or type) of the expressions that are in function positions, are appropriately chosen by the π–RED editor when

[28]Note that complete `if_then_else` clauses are applications of `then_else` clauses to predicate expressions.

returning intermediate programs:

$$\begin{array}{rcll}
\langle:\texttt{ap_expr}:\rangle & \Longrightarrow & \texttt{ap}\ \langle:\texttt{expr}:\rangle\ \texttt{to}\ [\langle:\texttt{expr}:\rangle\ \{:,\ \langle:\texttt{expr}:\rangle\ :\}] & | \\
 & & \langle:\texttt{expr}:\rangle[\langle:\texttt{expr}:\rangle\ \{:,\ \langle:\texttt{expr}:\rangle\ :\}] & | \\
 & & (\ \langle:\texttt{expr}:\rangle\ \langle:\texttt{expr}:\rangle\ \langle:\texttt{expr}:\rangle\ ) & | \\
 & & \texttt{if}\ \langle:\texttt{expr}:\rangle\ \texttt{then}\ \langle:\texttt{expr}:\rangle\ \texttt{else}\ \langle:\texttt{expr}:\rangle &
\end{array}$$

The structured expressions partition thus (the item marked ‡ may not yet be implemented):

$$\langle:\texttt{struct\_expr}:\rangle \Longrightarrow \langle:\texttt{tree}:\rangle^{\ddagger} \mid \langle:\texttt{sequence}:\rangle \mid \langle:\texttt{vector}:\rangle \mid \langle:\texttt{matrix}:\rangle$$

and, finally, we have the following constant expressions:

$$\begin{array}{rcll}
\langle:\texttt{const_expr}:\rangle & \Longrightarrow & \langle:\texttt{var}:\rangle \mid \langle:\texttt{num}:\rangle \mid \langle:\texttt{bool}:\rangle & | \\
 & & \langle:\texttt{string}:\rangle \mid \langle:\texttt{prim_func}:\rangle &
\end{array}$$

One level down, we have the following production rules for the various binding expressions:

$$\begin{array}{rcl}
\langle:\texttt{recfunc_def}:\rangle & \Longrightarrow & \texttt{def}\ \langle:\texttt{func_declar}:\rangle\ \texttt{in}\ \langle:\texttt{expr}:\rangle \\
\langle:\texttt{letvar_def}:\rangle & \Longrightarrow & \texttt{let}\ \langle:\texttt{var_declar};\rangle\ \texttt{in}\ \langle:\texttt{expr}:\rangle \\
\langle:\texttt{func_def}:\rangle & \Longrightarrow & \texttt{sub}[\langle:\texttt{var_list}:\rangle]\ \texttt{in}\ \langle:\texttt{expr}:\rangle \\
\langle:\texttt{pattern_match}:\rangle & \Longrightarrow & \texttt{when}\ \langle:\texttt{patterns}:\rangle\ \texttt{guard}\ \langle:\texttt{expr}:\rangle\ \texttt{do}\langle:\texttt{expr}:\rangle
\end{array}$$

The non-terminals other than $\langle:\texttt{expr}:\rangle$ of these rules in turn produce the various function and variable declarations (or binding constructs):

$$\begin{array}{rcl}
\langle:\texttt{func_declar}:\rangle & \Longrightarrow & \langle:\texttt{func_def}:\rangle\ \{:,\ \langle:\texttt{func_def}:\rangle\ :\} \\
\langle:\texttt{func_def}:\rangle & \Longrightarrow & \langle:\texttt{ident}:\rangle[\ \langle:\texttt{id_list}:\rangle\ ]\ =\ \langle:\texttt{expr}:\rangle \\
\langle:\texttt{var_declar}:\rangle & \Longrightarrow & \langle:\texttt{ident}:\rangle\ =\ \langle:\texttt{expr}:\rangle\ \{:,\langle:\texttt{ident}:\rangle\ =\ \langle:\texttt{expr}:\rangle\ :\} \\
\langle:\texttt{id_list}:\rangle & \Longrightarrow & \langle:\texttt{ident}:\rangle\ \{:,\ \langle:\texttt{ident}:\rangle\ :\} \mid \epsilon \\
\langle:\texttt{patterns}:\rangle & \Longrightarrow & \langle:\texttt{pattern}:\rangle \mid (\ \langle:\texttt{pattern}:\rangle\ \{:,\langle:\texttt{pattern}:\rangle\ :\}) \\
\langle:\texttt{pattern}:\rangle & \Longrightarrow & \langle:\texttt{pat_el}:\rangle\ \{:,\langle:\texttt{pattern}:\rangle\ :\} \mid \\
 & & \langle:\texttt{user_constr}:\rangle\ \{\ \langle:\texttt{pattern}:\rangle\ \} \mid \epsilon \\
\langle:\texttt{pat_el}:\rangle & \Longrightarrow & .\ \mid\ \langle:\texttt{ident}:\rangle\ \mid\ \ldots\ \mid \texttt{as}\ \ldots\ \langle:\texttt{ident}:\rangle\ \mid \\
 & & .+\ \mid \texttt{as}\ .+\ \langle:\texttt{ident}:\rangle\ \mid\ .*\ \mid \texttt{as}\ .*\ \langle:\texttt{ident}:\rangle\ \mid \langle:\texttt{const}:\rangle
\end{array}$$

Note that in the various declarations parts we use the non-terminal $\langle:\texttt{ident}:\rangle$ to denote defining variable occurrences. They need to be distinguished syntactically from applied variable occurrences in the respective body expressions which may be preceded by one or several protection keys. Thus, for applied variable occurrences we have the following production rule:

$$\langle:\texttt{var}:\rangle \Longrightarrow \langle:\texttt{ident}:\rangle \mid \backslash\langle:\texttt{var}:\rangle$$

The structured expressions break down as follows (again: the item marked ‡ may not yet be implemented):

$$
\begin{array}{rcl}
\langle\!:\texttt{tree}:\!\rangle^{\ddagger} & \Longrightarrow & <> \ \mid\ <\ \langle\!:\texttt{expr}:\!\rangle\ ,\ \langle\!:\texttt{expr}:\!\rangle\ > \\
\langle\!:\texttt{sequence}:\!\rangle & \Longrightarrow & <> \ \mid\ <\ \langle\!:\texttt{expr}:\!\rangle\ \{:,\ \langle\!:\texttt{expr}:\!\rangle\ :\}\ > \\
\langle\!:\texttt{vector}:\!\rangle & \Longrightarrow & \texttt{vect}\ <> \ \mid\ \texttt{vect}\ < \langle\!:\texttt{num}:\!\rangle\ \{:,\ \langle\!:\texttt{num}:\!\rangle\ \ :\}\ > \mid \\
 & & \texttt{tvect}\ <> \ \mid\ \texttt{tvect}\ < \langle\!:\texttt{num}:\!\rangle\ \{:,\ \langle\!:\texttt{num}:\!\rangle\ \ :\}\ > \mid \\
 & & \texttt{bvect}\ <> \ \mid\ \texttt{bvect}\ < \langle\!:\texttt{bool}:\!\rangle\ \{:,\ \langle\!:\texttt{bool}:\!\rangle\ \ :\}\ > \mid \\
 & & \texttt{btvect}\ <> \ \mid\ \texttt{btvect}\ < \langle\!:\texttt{bool}:\!\rangle\ \{:,\ \langle\!:\texttt{bool}:\!\rangle\ \ :\}\ > \mid \\
 & & \texttt{svect}\ <> \ \mid\ \texttt{svect}\ < \langle\!:\texttt{string}:\!\rangle\ \{:,\ \langle\!:\texttt{string}:\!\rangle\ \ :\}\ > \\
 & & \texttt{stvect}\ <> \ \mid\ \texttt{stvect}\ < \langle\!:\texttt{string}:\!\rangle\ \{:,\ \langle\!:\texttt{string}:\!\rangle\ \ :\}\ > \\
\langle\!:\texttt{matrix}:\!\rangle & \Longrightarrow & \texttt{mat}\ <<>>\ .\ \mid\ \texttt{mat}\ < \{:<\ \langle\!:\texttt{num}:\!\rangle\ \{:,\ \langle\!:\texttt{num}:\!\rangle\ \ >:\}\ >\ .\ \mid \\
 & & \texttt{bmat}\ <<>>\ .\ \mid\ \texttt{bmat}\ < \{:<\ \langle\!:\texttt{bool}:\!\rangle\ \{:,\ \langle\!:\texttt{bool}:\!\rangle\ \ >:\}\ >\ .\ \mid \\
 & & \texttt{smat}\ <<>>\ .\ \mid\ \texttt{smat}\ < \{:<\ \langle\!:\texttt{string}:\!\rangle\ \{:,\ \langle\!:\texttt{string}:\!\rangle\ \ >:\}\ >\ .
\end{array}
$$

Note that the production rules for matrices are context-sensitive in the sense that the material inside the outermost curly brackets represents row vectors, all of which must have the same number of elements (which, of course, is not properly specified by the simple context-free rules given above).

The following atoms may be derived from the non-terminals for constant expressions:

$\langle\!:\texttt{ident}:\!\rangle$ is any finite sequence of letters, decimal digits and special characters which starts with an upper case letter;

$\langle\!:\texttt{num}:\!\rangle$ is any finite sequence of decimal digits with or without a decimal point;

$\langle\!:\texttt{bool}:\!\rangle$ is either of the Boolean constants `true` or `false`;

$\langle\!:\texttt{string}:\!\rangle$ is any finite sequence of letters, decimal digits and special characters which is enclosed in a pair of quotation marks `'` and `` ` ``;

$\langle\!:\texttt{user\_constr}:\!\rangle$ is a finite sequence of letters starting with an upper case letter.

The following is a complete list of primitive functions as they can be derived from the non-terminal `prim_func` (the symbols in brackets, if any, are those used when denoting applications as `func[arg_1,...,arg_n]`; a symbol followed by ‡ means that the function is not yet implemented):

- monadic value-transforming functions:

abs	neg		
exp	ln		
sin	cos	tan	
floor	ceil	frac	trunc
vc_+ (vc_plus)	vc_− (vc_minus)	vc_∗ (vc_mult)	vc_/ (vc_div)

- monadic query functions:

ldim	vdim	mdim
class	type	empty

- monadic structuring functions (the functions marked ‡ which apply to trees may not yet be implemented):

transpose	reverse		
to_scalar	to_vector	to_tvector	to_matrix
head‡	tail‡		

- dyadic value-transforming functions:

+ (plus)	− (minus)	* (mult)	/ (div)
mod	ip	max	min
and	or	xor	
eq	ne	f_eq	f_ne
ge	gt	le	lt

- dyadic structuring functions:

lselect	lcut	lrotate	++ (lunite)
vselect	vcut	vrotate	vunite

- triadic value-transforming functions:

c_+ (c_plus)	c_− (c_minus)	c_* (c_mult)	c_/ (c_div)

- triadic structuring functions:

mselect	mcut	mrotate	munite
mreplace_r	mreplace_c		
ltransform			

- quaternary structuring functions:

mreplace	repstr

With the exception of the production rule for matrices, the syntax of KiR is context-free. In particular, any legitimate KiR expression which derives from the start symbol ⟨: expr :⟩ may be inserted wherever this non-terminal occurs on the right hand side of a production rule. Moreover, KiR is a dynamically typed language. No types need to be (and can be) specified for the formal parameters of KiR programs, and all substitutions by actual parameters are type-free. This is to say that all syntactically correct KiR expressions are legitimate arguments of defined function applications. There is no static type checking that would reject programs which are not well typed. Type checks are dynamically performed at run-time upon potential instances of δ-reductions. Applications of primitive functions that are not type-compatible are treated as constant expressions rather than producing error messages, and reductions continue in the remaining parts of the program. Thus, you have ample opportunity to write programs which are rather meaningless in the sense that they either don't terminate at all or produce lots of irreducible applications.

This freedom of program design may be considered a disadvantage insofar as a rigid monomorphic or polymorphic typing discipline is often claimed to introduce more confidence in program correctness. This point is stressed in most other functional or function-based languages [Turn85, HaMiTo88, HuWa88]. Programs that are well typed at least will never produce type errors at run-time. However, they may nevertheless contain many other errors and may not terminate either. Thus, it is pretty safe to assume that the correctness argument to some extent also serves as a pretext for a simple technical reason: typing is essential when it comes to compiling programs to efficiently executable conventional machine code. Since contemporary computing machines include only rudimentary run-time type checking facilities at best, the type consistency of programs must be statically inferred at compile time in order to guarantee that the code is in this respect fail-safe.

Unfortunately, compilation to type-checked code is not compatible with a full-fledged λ-calculus. Typing itself outlaws many useful higher-order functions (including self-applications) since all type systems that are currently in use cannot deal with recursive types [Miln78, HiSe86]. Moreover, compiling to static code to some extent separates the world of functions from the world of other objects. Whereas functions can still be passed as parameters, there is no way of computing new functions from function applications. Also, compiled and type-checked code must usually run to completion in order to produce meaningful and intelligible results. Thus, stepwise reduction and inspection of intermediate programs in a high-level representation cannot be supported either.

In contrast to this, the design of π–RED was strictly guided by the objective of supporting the reduction semantics of a full-fledged λ-calculus. This meant primarily high-level interpretation of λ-terms whenever necessary (rather than executing just compiled code), and also the absence of a typing discipline. However, this does not rule out a static type-checker for KiR programs which infers polymorphic types as far as possible. Nevertheless, neither need program execution be made dependent on the outcome of the type checking process, nor need all KiR programs be well typed in order to reduce them to normal forms that are free of irreducible redices. In fact, such a type inference system exists for Orel/2 [PlSc90, Zim91] and will in the future be available for KiR as well.

Appendix 2: Summary of the π–RED Controls

For quick references, we will in this appendix give a brief description of all the control functions of π–RED as they can also be looked up under the **help** feature that may be called with the function key F4.

Calling and Terminating π–RED

π–RED , if properly installed on your UNIX system, should be available under your own or someone elses user directory in a file directory `red`. The executable file `reduma` may be found under the path name `red/red.o`. These are the commands to call and exit from π–RED :

call : `reduma [-options] [ file [, file ] ]`
(the material enclosed in the brackets [,] is optional)

exit : `CTRL-c, CTRL-z` or editor command `exit`

options :
t: reduces the specified file and displays the time required;
i: instead of `red.init` uses the specified file as initialization file;
c: reduces the first file and compares it with the second file (if specified);
p: prints all specified editor files in the prettyprint format, using the extension `.pp`.

With any of the options enabled, π–RED runs in a batch mode, otherwise it runs in the interactive mode.

The Line Editor

The line editor handles what you type in the input line displayed at your screen. There is a bound on the length of the input line which is shown in the right-hand corner of the message field. An input may exceed this length, in which case you can see only part of it in the input field. The functions that are available to manipulate the contents of the input line other than just typing it are the following:

`F3` : turns on the insert mode in which existing input may be modified;

`CTRL-e` : turns the insert mode off;

`DEL` : deletes the character to the left of the current cursor position;

`F4` : deletes the character at the current cursor position;

`CTRL-d` : deletes all characters from (and including) the current cursor position to the end of the input line;

`cursor-home` : positions the cursor at the first input character;

`cursor-up` : moves the part of the input line actually displayed half a line to the left;

`cursor-down` : moves the part of the input line actually displayed half a line to the right;

`tab` : moves the cursor to the next tab stop.

Function Keys

π–RED supports 16 logical function keys F1 .. F16 which may or may not be assigned to (some of) the physical function keys that are available on your keyboard. You may define these assignments by yourself with the help of a `defkeys` command as described under the command interpreter. Alternatively, you may type in the input line (with the message line cleared) `:n`, where `n` is the number of the logical function key, which will be echoed as `Fn`.

`F1` : performs at most one reduction step on the current cursor expression[29];

`F2` : enters the read mode for `.edit` files;

`F3` : displays in the FA field of your screen the actual cursor expression;

`F4` : calls the π–RED help file;

`F5` : reads an expression from a backup buffer[30];

`F6` : reads an expression from a first auxiliary buffer[31];

`F7` : reads an expression from a second auxiliary buffer;

`F8` : enters the command mode;

`F9` : performs at most 1000 reduction steps on the current cursor expression;

`F10` : enters the write mode for `.edit` files;

`F11` : displays in the FA field the father expression of the current cursor expression;

`F12` : unassigned;

`F13` : writes the current cursor expression into a backup buffer[29];

`F14` : writes the current cursor expression into a first auxiliary buffer;

`F15` : writes the current cursor expression into a second auxiliary buffer;

`F16` : terminates π–RED .

[29]This is a default value which can be changed using the editor command `redcnt`.

[30]Note that this buffer by default always contains the expression held under the current cursor position prior to the last shift of reductions.

[31]In conjunction with special editor commands, this buffer may be used for dedicated purposes.

The Command Interpreter

The command mode may be entered by means of the function key F8. Under this mode, only one command may be executed at a time. π–RED returns to the normal edit mode immediately afterwards.

The following commands are available:

The Editor Commands

`abb_def` : abbreviates all `def` constructs;

`abb_idef` : abbreviates only all inner `def` constructs;

`append file` : appends prettyprint to `file`;

`arity` : displays the arity of the constructor that is in cursor position;

`cat file` : displays the contents of the file `file`;

`cmd shell_cmd` : issues the shell command `shell_cmd` to a shell;

`comp` : compares the expression in the first auxiliary buffer with the current cursor expression;

`curmode[par]` enables the line-oriented cursor mode with `par = on` and the syntax-oriented cursor mode with `par = off`, the latter also being the default option;

`defkeys` : enters the mode under which logical function keys and cursor movements can be assigned to physical function keys on your keyboard;

`dfct` : shows the function headers of abbreviated `def` constructs;

`dgoal` : shows the goal expressions of abbreviated `def` constructs;

`dmode[ n ]` : defines the number of nesting levels up to which an expression is being displayed (with `n = -1` showing all levels);

`editparms` : shows all editor parameters;

`exit` : terminates the editor;

`ext[ i [extension]]` : defines the extensions of the various file types that can be used under the π–RED editor, with `i` being assigned as follows:

0: input/output file `.ed`;
1: prettyprint file `.pp`;
2: document file `.doc`;
3: protocol file `.prt`;
4: ASCII file (portable) `.asc`;
5: red expression file `.red`;
6: pre-processed expr `.pre`;

`find` : searches, from the current cursor position downwards, for the first occurrence of a sub-expression that matches the expression stored in the topmost position of the first auxiliary buffer, if there is one;

`findexpr[expr]` : searches, from the current cursor position downwards, for the first occurrence of of a sub-expression that matches `expr`;

`help[text]` : searches for help;

`initfile[file]` : changes the name of the initialization file to `file`;

`initparms` : reads the parameters held in the initialization file;

`load[file]` : loads into the current cursor position an expression from `file` (which must be of type `.red`);

`next` : tries to find under the current cursor expression the next occurrence of the expression held in the first auxiliary buffer;

`pp[file]` : prettyprints the current cursor expression to `file.pp`;

`print[file]` : prints the current cursor expression to `file.asc`;

`protocol file` : appends to the file `file.prt` the (intermediate) expressions obtained after every shift of reductions;

`protocol` : terminates protocolling;

`read[file]` : reads an ASCII text expression from `file.asc`;

`redcnt n` : sets the default reduction counter assigned to the function key `F1` to the value `n`;

`reduce n` : performs at most `n` reduction steps on the current cursor expression;

`redparms` : shows the reduction parameters;

`refresh` : refreshes the screen;

`replace[expr_1; expr_2; mode]` searches, from the current cursor position downwards, for occurrences of sub-expressions that match `expr_1`, and replaces them by `expr_2`, with `mode` specifying whether all replacements are to be made in one go `mode = all` under interactive control `mode = ask`;

`saveparms` : saves reduction and editor parameters;

`shell` : calls another UNIX shell;

`small on|off` : sets the display mode;

`stack` : displays the contents of the π–RED run-time stacks;

`store[file]` : saves an expression on `file.red`;

`time` : shows the time used up for the last shift of reductions;

For the most frequently used of these commands there exist abbreviations.

pagesize no_of_bytes : sets the size of the communication page to the π–RED interpreter;

heapdes no_of_entries : specifies the number of entries in the descriptor array;

heapsize no_of_bytes : specifies the size of the heap segment in numbers of bytes;

qstacksize no_of_el : specifies, in numbers of stack elements (of four bytes), the sizes of the stacks E,A,H of the run-time environment of π–RED ;

mstacksize no_of_el : specifies the size of stack M of the run-time environment;

istacksize no_of_el : specifies the size of the argument stack I of the run-time environment;

fixformat : converts all decimal numbers into binary coded integer or floating point formats of the host system;

varformat : performs all arithmetic operations on decimal number representations with unlimited precision (this is the defasult option of π–RED);

base : defines the base for the decimal number representation under the `varformat`;

trunc no_of_digits : specifies an upper limit on the number of digits displayed for any decimal number;

mult_prec no_of_digits : specifies, in numbers of digits after the decimal point, the precision for decimal multiplications;

div_prec no_of_digits : specifies, in numbers of digits after the decimal point, the precision for decimal divisions;

betacount on|off : when on, counts only reductions of defined function applications, otherwise all δ-reductions as well.

The File System

π–RED supports several types of files to store and communicate KIR expressions. It makes use of the underlying UNIX file system. All files are stored under the same directory under which the executable file `reduma` is being called. The file types are distinguished by appropriate default extensions which are being appended to the file names upon file creation by type-specific commands. The following types are available:

input/output files (type 0)

This file type carries the default extension (suffix) `.ed`. It is used to create libraries of KIR expressions in the internal editor format.

Commands:

Writing an `.ed` file: function key F10 followed by `file_name`;
Writes the current cursor expression into the file `file_name.ed`. If this file already exists, its actual contents are overwritten.

Reading an `.ed` file: function key F2 followed by `file_name`;
Overwrites the current cursor expression with the expression held in the file `file_name.ed` if it exists.

`.ed` files may also be referenced from within KIR programs by `$file_name` (which must start with the first letter typed as upper case), in which case the full expressions held in the files are literally substituted for the references before executing the programs.

prettyprint files (type 1)
: This file type carries the default extension `.pp`. KIR expressions are in these files stored in the same layout in which they appear on the screen. In this form, they may be either directly printed or included in other text files.

Commands:

Writing a new or overwriting an existing `.pp` file: `pp[file_name]`.

document files (type 2) :
: The default extension for this file type is `.doc`. The command `doc file_name` is an abbreviation for the UNIX system command `$EDITOR file_name.doc`.

protocol files (type 3)
: The default extension for this file type is `.prt`. It may be used to store a succession of intermediate KIR expressions while executing a program in a stepwise mode.

Commands:

Opening a protocol file: `prot file_name`
Closing a protocol file: `prot`

ASCII files (type 4)
: The default extension is `.asc`. This file type uses a special external format for KIR expressions which is portable to other (text) editors.

Commands:

Writing an ASCII file: `print[file_name]`
(Over)writes the file `file_name.asc` with the current cursor expression;

Reading an ASCII file: `read[file_name]`
Overwrites the current cursor expression with the expression held in the file `file_name.asc`, if it exists.

red files (type 5)

The default extension of this file type is `.red`. This is another input/output file type which, in difference to files of type 0, treats KiR expressions as macros: reducing them, e.g., in the context of a larger program into which they are being inserted, decrements the reduction counter by just one, irrespective of the actual number of reduction steps that are carried out.

Commands:

Writing a `.red` file: `store[file_name]`
(Over)writes the file `file_name.red` with the current cursor expression;

Reading a `.red` file: `load[file_name]`
Overwrites the current cursor expression with the expression in `file_name.red`.

Appendix 3: The X-Window Frontend for the Editor

For some UNIX-based systems there is also available an X-window frontend which may be called with `Xred`. It facilitates handling the editor in that it realizes many of the editor controls, parametrization of the run-time system, file handling etc. as menue functions, i.e., you can mainly work with a mouse rather than typing editor commands. There are also two important extensions that come with the X-windows implementation: it is now possible to load ASCII files and to switch between syntax-oriented and line-oriented editing. Since the user is guided by menues, there is also no need to learn the syntax of the editor commands.

The following system versions are required to support `Xred`:

- UNIX system V, release 3.2 with IPC message queues enabled;
- X window system X11R5 or higher;
- X-view 3.2 or higher toolkit;
- colour work station or X-terminal.

Figure 2 shows the main window of the X-window frontend. It consists of the following three parts (from top to bottom):

- the command panel which includes eight menue buttons and also displays the (file) name of the program that is actually in use;
- the usual alphanumerical interface of π–RED , as described in chapter 3;
- the window footer which displays various short help and editor messages.

The X-frontend is just another interface sitting on top of the editor program that controls the alphanumerical interface. All features and functions, other than those that have been added, are working in the same way as before.

Under one of the menues, you may also call a text editor object which opens up another window, in which you can use a simple line-oriented mode to edit (parts of) KiR programs. Provided they are syntactically correct, they may be inserted in any syntactical position of the expression that is under the control of the main window. The layout of the text editor object is shown in fig. 3. It too features a command panel with eight function buttons, a main field which holds the text to be edited, and a window footer for various messages.

All system functions of the main window are grouped into eight menues which may be accessed through the menue buttons of the command panel. Following standard conventions, the menue entries that show up when clicking one of the menue buttons may be followed by three dots or by arrows. Selecting one of the former menues opens up a user request box, selecting one of the latter opens up another sub-menue. All menues may be pinned on the screen in order to use its functions repeatedly.

Here is a brief explanation of the available menues:

- the menue FILE includes all the file handling functions. These are

  `clear` which clears the FA-field of the editor and the backup buffer;

  `load ...` which opens a file request box to load a file of one of the formats `.ed` (for input/output files) `.red` (for red expression files) or `.pp` (for pretty print (ASCII) files). If you try to load a `.pp` file which contains a syntax error, the frontend will switch to a text editor object for corrections, otherwise the frontend asks you whether you want the expression stored away as an `.ed` file.

  `include` which opens a file request box to insert into the actual cursor position an expression read from a file of one of the three legitimate formats;

  `save` which saves the actual cursor expression under the file name last used;

  `save as ...` which opens a file request box to specify a file name under which the complete actual expression is to be saved;

  `save expression ...` which opens a file request box to specify a file name under which the actual cursor expression is to be saved;

  `load setups ...` which loads the setup file for the system parameters for inspection and changes;

  `save setups ...` which stores the (updated) parameter set shown on the screen in the setup file;

  `exit` which quits the X-window frontend;

  All functions followed by ... bring up a file request box which displays the files or directories (folders) that are available under the directory in which you called `Xred`. You may select one of the files or directories, preferably `red.o` in the latter case, to which you wish to store or from which you wish to fetch the actual cursor expression. Clicking one of the directories brings up the next lower level of files or directories, clicking the name of a file of type `.ed` actually does the copying of the file contents into the actual cursor position (or vice versa).

- the menue EDIT includes some functions which support syntax-oriented editing:

  `undo` undoes the last modification in that it copies the contents of the back-up buffer into the actual cursor position;

  `copy` copies the actual cursor expression into the backup buffer;

`paste` inserts the expression held in the backup buffer into the actual cursor position;

`cut` moves the actual cursor expression into the backup buffer, leaving an empty symbol `#` in its syntactical position;

`texteditor ...` calls the text editor object which shows up on your screen as another window with essentially the same layout. In its expression field you will find the cursor expression of the main window set up for line-oriented editing, while the cursor position in the main window is now replaced by a system-generated identifier for the text editor object. You may now directly use the keyboard of your workstation to modify or edit your program text. You may insert into this text all expressions stored under dollar variables. When clicking the `done` button in the command field, the text editor window disappears and the edited expression replaces the identifier in the expression field of the main window. Note that returning in this way to the main window is only possible if the expression is syntactically correct, otherwise the text editor window stays in place and displays an error message at the bottom. Thus it is now your responsibility to construct syntactically correct expressions.

- the menue BUFFERS includes the functions by which the actual cursor expression is written to or read from one of the buffers (using self-explaining inscriptions of the buttons);

- the menue REDUCE includes the functions which control the reduction process:

`undo last reduction` re-installs the cursor expression as before the last sequence of reductions;

`base expression` sets the cursor to the top of the entire expression;

`one step` performs at most one reduction step on the cursor expression;

`set reduction counter ...` opens a request box for the specification of an initial reduction counter value (which may be done by clicking the up/down arrows to the right of the counter field or by directly typing it into this field);

`use reduction counter` performs on the cursor expression at most as many reduction steps as specified by the last counter setting;

`reduce expression` performs an unlimited number of reduction steps and should therefore be used only if it is absolutely certain that the cursor expression terminates eventually;

- the OPTIONS menue allows you to set up certain system parameters such as the parameters of the run-time system, some properties of the frontend, and the layout of the expressions under the syntax-oriented editor mode;

- the EXTRA menue contains some special functions like tool boxes for frequently used commands;
- the HELP menue includes the help files, with the various help items classified in the same way as under the alphanumerical frontend.

Figure 2: Screen layout of the Xred main window

Figure 3: Screen layout of the text editor object

Appendix 4: Installing π–RED on a Host System

The following primarily concerns the installation of π–RED $^+$, i.e., of the compiled graph reducer version of π–RED , which is our standard version for release to the public. The installation of the interpreter version π–RED * is very similar.

π–RED $^+$ will be made available to you on a floppy disc or on our ftp-server as a compressed `tar` file. In order to get it loaded into your system, you must execute, under the directory under which you wish to install it, the UNIX commands

```
tar  xvf  /dev/rfd0c
uncompress  red.tar.Z
tar  xvf  red.tar
```

in this sequence. This will open up a new subdirectory `red/` under which you can find all the (executable) files and further subdirectories necessary to compile and assemble the complete reduction system.

If you have under the same directory another version of π–RED configured for a different target machine, first execute

```
make  clean
```

to delete all files that may be invalid. Also, on a UNIX system V/4 make sure that `/usr/bin` precedes `usr/ucb` in the directory path since the `cc` command in the latter uses libraries which have bugs.

If you wish to install π–RED on either a SUNSPARC or an Apollo system, you simply need to run next the executable file

```
configure  sun|apollo          .
```

in order to prepare both the editor and the run-time system for compilation to executable code. For all other systems, you may have to set 'by hand' several flags in two `makefiles` named `fed/Makefile` and `src/Makefile`. These compiler flags are as follows:

Editor flags contained in `fed/Makefile` (with default setting):

`-DUNIX=1`	for UN*X OS systems;
`-DAPOLLO=0`	for Apollo systems set this to 1;
`-DTRICK1_AP=1`	for internal use only (don't change);
`-DKEDIT=0`	for use with a distributed version of π–RED (don't change);
`-DM_PATT=1`	for multiple pattern matches;
`-DTRACE=0`	for internal use only (don't change);
`-DClausR=0`	for extended system version with logical variables (don't change);
`-DB_DEBUG=0`	for internal use only (don't change);

Run-time flags contained in `src/Makefile` (with default setting):

`-DUNIX=1`	for UN*X OS systems;
`-DODDSEX=0`	for Littleendian systems set this to 1;
`-DAPOLLO=0`	for Apollo systems set this to 1;
`-DSYS5=0`	for System5 systems set this to 1;
`-DPI_RED_PLUS=1`	for internal use only (don't change);
`-DRED_TO_NF=1`	enables reductions to normal forms;
`-DDEBUG=0`	disables custom-made debug system;
`-DDBUG=0`	disables fred-fish's debug package;

Having executed the `configure*` file or set the compiler flags directly, you may now call

```
make
```

which finally produces the executable file `reduma*`. Calling it brings π–RED into existence on your host system, i.e., the user interface shows up on the screen (window) you are working with.

Before calling `make`, it is advisable to look into the initialization file `red.init` in order to make sure that the system is set up with the parameters as you need them for most of your applications. This concerns primarily the sizes of the various run-time structures (heap size, stack sizes, etc.). This is good practice insofar as any changes of these parameters by the respective system commands do not survive a session, i.e., whenever you call the system with the command `reduma`, it is initialized with the old parameters taken from `red.init` at compile time.

The installation of the system version XRED which features an X-window frontend pre-supposes the existence of the system π–RED , preferably under the same directory. It also requires the following system environment:

- a UNIX system V, release 3.2 with IPC message queues enabled;
- an X-window system X11R5 or higher;
- X-view 3.2 or higher;
- a colour work station or X-terminal;

As with π–RED , XRED will be made available either on a floppy disc or on our ftp-server as a compressed `tar` file. To get it installed, you must execute, under this directory, the UNIX commands

```
tar  xvf  /dev/rfd0c
uncompress  udi.tar.Z
tar  xvf  udi.tar
```

in this sequence, which creates a new subdirectory `udi/` of source files. In order to get everything compiled, first you have to climb down into this directory with `cd udi`, under which you will find a file `README` (which should be read before proceeding) and a file `Install`. In `Install` you will have to set a number of paths according to your particular installation. These are

`REDLIB`	= path to the π–RED library;
`REDSRC`	= path to the source code of π–RED
`SFEDSRC`	= path to the source code of the syntax-drected editor;
`XFEDLIB`	= path to the X-window version of the library of the syntax-directed editor;
`TOPPATH`	= the complete path to the directory under which the `Install` file is held;
`SFED_PATH`	= search path to the help file `creduma.h` of the syntax-directed editor.

Next you have to call this installation script with `./Install`. Upon termination, it shows how the following two environment variables are set up:

`$PATH`	which should contain the directory `./bin`;
`$FHELPFILE`	which must point to the help file `creduma.h` for the syntax-directed editor.

If not, you must set it according to your particular installation.

Finally you may compile your source files using the following sequence of commands:

```
cd src
make depend
make
```

This will create in your directory

- `udi/bin` the executable files

```
Xred
kir
marco
sxred
```

- `udi/lib` the libraries and example make files

```
Example.Makefile.Compiler-Edoitor
Example.Makefile.CompilerGen
editor.o
libkir.o
libmarco.o
marco-main.o
```

A brief explanation of these files may be found in the `README` file.

References

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

[Abr70] Abrams, P.S.: *An APL Machine*, Technical Report No.3, Stanford University,Cal., Feb. 1970

[ApMcQu87] Appel, A.W.; MacQueen, D.: *A Standard ML Compiler*, Proceedings of the Conference on Functional Programming and Computer Architecture, Portland, Oregon, Lecture Notes in Computer Science, No. 274, 1987, pp. 301–324

[Aug84] Augustsson, L.: *A Compiler for Lazy ML*, Proceedings of the ACM Conference on Functional Programming and LISP, Austin, Texas, August 1984, pp. 218–227

[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

[Bar81] Barendregt, H.P.: *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, Studies in Logic and the Foundations of Mathematics, Vol. 103, 1981

[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

[Berk78] Berkling, K.J.: *Computer Architecture for Correct Programming*, Proceedings of the 5th Annual Symposium on Computer Architecture, 1978, ACM/IEEE, pp. 78–84

[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

[BiWa88] Bird, R.S; Wadler, P.L.: *Introduction to Functional Programming*, Prentice Hall, Englewood Cliffs, NJ, 1988

[BuMQSa80] Burstall, R.M.; MacQueen, D.; Sannella, D.T.: *HOPE : An Experimental Applicative Language*, CS Report CSR-62-80, University of Edinburgh, May 1980

[Chur41] Church, A.: *The Calculi of Lambda Conversion*, Princeton University Press, 1941

[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

[Gaer91] Gaertner, D.: π–RED $^+$*: ein interaktives codeausfuehrendes Reduktionssystem zur vollstaendigen Realisierung eines angewandten* λ*-Kalkuels*, Institut fuer Informatik und Praktische Mathematik, CAU Kiel, Bericht Nr. 9201, 1992

[GiGu82] Giloi, W.K.; Gueth, R.: *Concepts and Realization of a HIgh-Performance Data Type Architecture*, Int. Journal of Comp. and Inf. Science, Vol. 11, No. 1, 1982, pp. 25 – 44

[HaMcQMi86] Harper, R.; MacQueen, D.; Milner, R.: *Standard ML*, Laboratory for Foundations of Computer Science, University of Edinburgh, March 1986

[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

[HiSe86] Hindley, J.R.; Seldin, J.P.: *Introduction to Combinators and λ-calculus*, Cambridge University Press, London Mathematical Society Student Texts 1, 1986

[Hom77] Hommes, F.: *The Internal Structure of the Reduction Machine*, GMD-ISF-77-3, D–5205 Sankt Augustin 1, March 1977

[Hom80] Hommes, F.: *An Expression Oriented Editor for Languages with a Constructor Syntax*, Proceedings of the International Workshop on High-Level Language Computer Architecture, Fort Lauderdale, Florida, May 1980, pp. 181–189

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

[Iver62] Iverson, K.E.: *A Programming Language*, John Wiley, 1962

[John87] Johnsson, T.: *Compiling Lazy Functional Languages*, PhD Thesis, Chalmers University of Technology, Goeteborg, 1987

[Klu79] Kluge, W.E.: *The Architecture of the Reduction Machine Hardware Model*, GMD-ISF-79-3, D–5205 Sankt Augustin 1, August 1979

[KlSc80] Kluge, W.E.; Schluetter, H.: *An Architecture for Direct Execution of Reduction Languages*, Proceedings of the International Workshop on High-Level Language Computer Architecture, Fort Lauderdale, Florida, May 1980, pp. 174–180

[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

[McQMi87] MacQueen, D.; Milner, R.; Mitchell, K.; Sannella, D.: *Functional Programming in ML*, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987

[Miln78] Milner, R.: *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, Vol. 17, 1978, pp. 348–375

[Nikh88] Nikhil, R.S.: *ID Version 88.1, Reference Manual*, MIT Laboratory for Computer Science, CSG Memo 284, 1988

[PlSc90] Pless, E.; Schluetter, H.: *The Reduction Language OREL/2*, Arbeitspapiere der GMD, 1990

[Schm86] Schmittgen, C.: *A Data Type Architecture for Reduction Machines*, Proceedings of the 19th Hawaii International Conference on System Sciences, Vol. I, 1986, pp. 78–87

[SmBlKl91a] Schmittgen, C.; Bloedorn, H.; Kluge, W.: *Structured Data Types in the Reduction System* π-RED, in: Arrays, Functional Languages and Parallel Systems, Kluwer Academic Publishers, 1991, pp. 171–183

[SmBlKl91b] Schmittgen, C.; Bloedorn, H.; Kluge, W.: π-RED* – *A Graph Reducer for a Full-Fledged* λ*-Calculus*, New Generation Computing, OHMSHA Ltd. and Springer, Vol. 10, No. 2, 1992, pp. 173–195

[Turn85] Turner, D.A.: *Miranda : A Non-Strict Functional Language with Polymorphic Types*, Proceedings of the Conference on Functional Programming and Computer Architecture, Nancy, Lecture Notes in Computer Science, No. 201, Springer, 1985, pp. 1–16

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

[Zim91] Zimmer, R.: *Zur Pragmatik eines Operationalisierten* λ*-Kalkuels als Basis fuer Interaktive Reduktionssysteme*, GMD-Bericht Nr. 192, Oldenbourg, 1991

Index