π–RED$^+$
A Compiling Graph–Reduction System for a Full–Fledged λ–Calculus

D. Gärtner, A. Kimms and W. Kluge
Christian–Albrechts–Universität Kiel
Institut für Informatik
Preußerstraße 1–9
D–2300 Kiel 1
Email: {dg,ak,wk}@informatik.uni–kiel.dbp.de

Abstract

This paper describes a compiling graph reduction system which performs high–level program transformations governed by the reduction rules of an untyped applied λ–calculus. In contrast to other well–known graph reducers, our system completely implements the reduction semantics of a full–fledged λ–calculus. This renders it possible to support higher–order functions without any restrictions. Functions may be computed by applications of functions to other functions or by partial applications and are returned to the user as high–level programs. Also, programs may be executed in a step–by–step manner under interactive control. All intermediate results are presented to the user as high–level programs. Stepwise program execution provides a convenient high–level debugging facility.

In order to execute programs with competitive speed, we have developed an abstract stack machine ASP, which gives special support to some key techniques for the execution of untyped functional languages: function calls, parameter passing, dynamic type–checking and heap management.

The second part of the paper describes an extension to the original version of π–RED$^+$, which terminates reduction whenever weak normal forms (WNFs) are reached. The extension makes it possible to reduce programs to their normal forms by using the basic mechanisms of the abstract stack machine in conjunction with a novel indexing scheme which resolves naming–conflicts that may occur in the presence of free variables.

1 Introduction

Compiled graph reduction is known to be an efficient implementation technique for functional languages [Joh84, Fai86]. Function definitions are usually compiled to some flat set of supercombinator code which systematically constructs and reduces the graph of what eventually becomes the resulting program term. Compilation to efficient code generally requires that programs be well–typed and do not contain free variables. This approach does not give full support to higher–order functions: functions statically and irreversibly turned into code may be supplied to other functions as parameters, but they cannot be returned as results, at least not in intelligible form. What can be computed are merely (sequences of) basic values.

In order to exploit the full potential of the functional programming paradigm, implementing a full–fledged untyped λ–calculus appears to be essential. The reduction rules of the λ–calculus specify program transformations by context–free substitutions. High–level functional programs may be converted in a more or less straightforward manner into equivalent λ–terms for reduction.

By subsequently re–converting partially or completely reduced λ–terms into equivalent high–level programs, a reduction semantics for the high–level language is established. Due to the Church–Rosser property, reductions may be carried out in any part of a large program without causing side–effects in other parts. Thus, functional systems which strictly rely on the reduction rules of the λ–calculus are well suited for interactive program development, high–level debugging, and rapid prototyping.

We have in the past experimented with various versions of an applicative order graph reducer π–RED which achieves these ends by high–level interpretation of λ–terms [SKZ89, BSK92]. Though internally performing supercombinator reductions, π–RED employs a post–processor which, by means of full β–reductions, turns all top–level closures left over at the end of the processing phase into WNF's of λ–abstractions. In particular, all naming conflicts are correctly resolved using an indexing scheme for variables which avoids renaming. Moreover, since the system truly implements a reduction semantics, programs can be executed stepwise and intermediate λ–terms can be returned to the user in high–level notation. The performance of π–RED equals that of an SML interpreter and outperforms a Miranda interpreter by a factor of three [Zim90].

Without compromising any of the above features, π–RED$^+$ [Gär91] consequently employs compiled graph reduction, making use of the fact that interpretation steps are necessary only in those cases for which the course of program execution cannot be completely statically inferred. They essentially include partial applications and occurrences of free variables in function positions of applications. Compilation is made fully reversible in the sense that high–level program representations can be re–constructed from all intermediate states of computation which correspond to complete β– and δ–reduction steps.

Figure 1 gives an overview over the different program transformation levels in π–RED$^+$. At the user interface, high–level program transformations can be observed. High–level programs may be specified in KiR [KIR89], a purely functional language. The internal machinery is based on the reduction rules of an applied λ–calculus $\mathcal{M}\text{ini}\mathcal{R}\text{ed}$. Internally, reductions are performed by executing abstract machine code on the abstract stack machine ASP.

$$\begin{tikzcd}
\text{KiR:} & \text{program} \arrow[rr, "\beta\text{–}/\delta\text{–}red."] \arrow[d, "pre\text{–}"] & & \text{program'} \\
\mathcal{M}\text{ini}\mathcal{R}\text{ed:} & \lambda\text{–expr} \arrow[rr, "\beta\text{–}/\delta\text{–}red."] \arrow[d, "comp"] & & \lambda\text{–expr'} \arrow[u, "post\text{–}"'] \\
\text{ASP:} & \text{code} \arrow[urr, "processing"] & &
\end{tikzcd}$$

Figure 1: Program transformation levels in π–RED$^+$

As an improvement over π–RED, all terminating reduction sequences now return normal forms, i.e. reductions continue in the bodies of λ–abstractions, particularly of those resulting from partial applications. Note that this is accomplished by executing statically compiled code. The following expression gives a good example and underlines the relevance of computing normal forms. The application of the well–known twice–function to itself can be expressed in sugared λ–calculus notation as follows:

let twice = λf.λx.(f (f x)) in (twice twice)

Computing a weak normal form leads to:

λ x.let twice = λ f. λ x.(f (f x)) in (twice (twice x))

This result is not very satisfactory result in the sense that the λ–calculus would allow for more β–reductions. The normal form of this expression is what is really wanted:

λ x'. λ x.(x' (x' (x' (x' x)))),

i.e. the self–application of a binary function which applies its first argument twofold to its second argument reduces to a binary function which applies its first argument four times to its second argument.

π–RED$^+$ is based on a novel abstract machine ASP (for abstract stack processor) which features four runtime stacks. Two of them are alternatingly used as argument and workspace stacks for nested function calls in order to be able to release argument graphs as early as possible. In combination with reference counting, this feature renders it possible to perform a fairly high percentage of value–transforming operations on large data structures as simple updates. The other two stacks are used to store return addresses and efficiently support pattern matching and list comprehensions. Compilation to abstract machine code employs some special optimization techniques, e.g. for general tail–end recursions and early arguments release. Low–level interpretation of this code enhances the performance by up to an order of magnitude over high–level interpretation of λ–terms. The implementation of ASP on a tailor–made RISC–processor yields program run–times equal to, and in some cases less than those for equivalent C–programs [Aßm92].

We will outline the basic operating principles of ASP, basic compilation and de–compilation schemes, and particularly focus on the implementation of full β–reductions.

2 Program Execution in π–RED$^+$

Programs for π–RED$^+$ may be written in the functional languages KiR (Kiel Reduction Language) or OREL/2 [SP90]. It is also possible to support any other purely functional language which can be transliterated reversibly into the intermediate language $\mathcal{M}$ini$\mathcal{R}$ed, an applied λ–calculus which is described below.

KiR (or OREL/2) programs typically consist of a set of mutually recursive, nested function definitions, and an expression to be evaluated which is generally constructed from function applications.

π–RED$^+$ performs high–level program transformations governed by the reduction rules (β– and δ–reductions) of the underlying applied λ–calculus $\mathcal{M}$ini$\mathcal{R}$ed.

All intermediate states of program execution are (if desired) returned to the user as valid high–level programs which may be inspected, modified and re-submitted for further reductions.

Stepwise reduction is illustrated by the example program `Foldr` which is shown below.

```
def
  Foldr [Op,N,L] = def
                     Fldr [L] = if empty(L)
                                then N
                                else ( hd(L) Op Fldr[tl(L)] )
                   in Fldr [L]
in Foldr [+,0,<1,2,3>]
```

The execution of one reduction step (one step in this context means the execution of exactly one application of a user–defined function) returns the application of an instance of the new local function `Fldr` to the initial argument list.

```
def
  Fldr [L] = if empty(L)
             then 0
             else ( hd(L) + Fldr[tl(L)] )
in Fldr [<1,2,3>]
```

The relatively free variables `Op` and `N` in the body of the original `Fldr` have been substituted by the primitive operator `+` and its neutral element `0` respectively. The new function `Fldr` is thus a specialized version of the original one and specifies directly the computation of the sum of all elements of its argument list. The next reduction step starts unwinding the summation and delivers

```
( 1 + def
        Fldr [L] = if empty(L)
                   then 0
                   else ( hd(L) + Fldr[tl(L)] )
      in Fldr [<2,3>] )
```

which is reduced in three further reduction steps to 6.

π–RED$^+$ truly supports higher–order functions. Partial applications on the outermost program level are reduced by full β–reductions. For example the partial application `Foldr[+,0]` is reduced in one step to

```
lambda [L]
in def
     Fldr [L] = if empty(L)
                then 0
                else ( hd(L) + Fldr[tl(L)] )
   in Fldr [L].
```

This is also a variant of the specialized summation function mentioned above (in fact it is η–equivalent to the local function `Fldr`).[1]

Another important aspect of π–RED$^+$ is the way it handles free variables. Naming conflicts which may occur in the course of full β–reductions are correctly resolved by a protection mechanism proposed by Berkling [Ber76]. Free variables, when substituted into the scope of a variable with the same name, are protected against parasitic bindings by a so called λ–bar, denoted as `\`.

`ap lambda [X,Y] in <X,Y> to Y` $\rightarrow_\beta$ `lambda [Y] in <\Y,Y>`.

The λ–bar is removed, when the binding is removed against which the free variable is protected:

`ap lambda [Y] in <\Y,Y> to Z` $\rightarrow_\beta$ `<Y,Z>` .

The λ–bar mechanism dynamically renames (free) variables, if necessary by simply adding or deleting protection keys (λ–bars).

3 A Simple Functional Language

High–level functional programs are transliterated before execution into the intermediate language $\mathcal{M}$ini$\mathcal{R}$ed. Using an intermediate language renders it possible to change the syntax of the high–level language without affecting the internal machinery. Only the transliteration of high–level programs into $\mathcal{M}$ini$\mathcal{R}$ed and vice–versa must be re–specified. $\mathcal{M}$ini$\mathcal{R}$ed is a de–sugared applied λ–calculus defined by the constructor–syntax shown in figure 2.

A $\mathcal{M}$ini$\mathcal{R}$ed–expression may be either a constant (denoted by c), a variable, (denoted as x), a λ–abstraction, an application of a function expression e_1 to an argument–expressions e_2, a

[1] η–reductions are currently not executed by π–RED$^+$.

$$e = c \mid x \mid \lambda\, x\, .\, e \mid @\, e_1\, e_2 \mid \; < e_1 \ldots e_n > \; \mid \texttt{letrec}\ f_1{=}e_1\ \ldots\ f_n{=}e_n\ \texttt{in}\ e_0$$

Figure 2: The syntax of $\mathcal{M}$ini$\mathcal{R}$ed–expressions

n–ary list or a group of mutually recursive function definitions with goal–expression e_0. Other syntactic constructs, e.g. pattern matching, are not discussed here any further.

4 The Program Execution Phases of π–RED$^+$

Program execution in π–RED$^+$ is subdivided into four phases: pre–processing, compiling, processing and post–processing in order to keep the processing phase as simple and efficient as possible. Processing is based on compiled supercombinator graph–reduction. The burden of performing full β–reductions is shifted into the post–processing phase.

In the pre–processing–phase, the token string representation of the high–level program received from the programming environment is transformed into an equivalent $\mathcal{M}$ini$\mathcal{R}$ed expression which is internally represented as a graph. This expression is then transformed into a supercombinator expression by *lambda–lifting* [Joh85]. Bound variable occurrences are eliminated and replaced by de Bruijn–indices [dB72] (which are regarded as special variables). These can be used as offsets for indexed stack accesses as a means of efficient variable substitutions. To distinct between λ–terms and supercombinators, upper case lambdas with the original variable names as indices are used to specify the binders of the latter. For the execution of supercombinator–reductions the variable names are not needed any more. But in order to be able to transform supercombinators back into high–level programs, the original variable names are kept both in their text– and graph–representation.

The `twice`–program from above is transformed into a supercombinator graph which corresponds to the following $\mathcal{M}$ini$\mathcal{R}$ed expression:

$$\texttt{letrec}\ \mathrm{twice} = \Lambda_f\, .\, \Lambda_x\, .\, @\ \#1\ @\ \#1\ \#0\ \texttt{in}\ @\,\mathrm{twice}\,\mathrm{twice}.$$

The variables f and x are within the functions body replaced by the indices #1 and #0 respectively. Globally free variables are left in place as they are and treated as constants. As long as only supercombinator–reductions are performed, this cannot cause any naming conflicts.

The supercombinator graph produced by the pre–processing phase is compiled into code for the abstract stack processor ASP which is described in more detail in section 5.

Figure 3 shows the graph– and code–structure of the `twice`–program as generated in the pre–processing and compiling phases. The graph can be mapped one to one to the $\mathcal{M}$ini$\mathcal{R}$ed–expression given above. Groups of function definition are represented as LREC–nodes, pointing to the functions names and definitions, right hand sides of function definitions as LAMBDA–nodes, which contain pointers to the bound variable names and the functions body, and occurrences of letrec–bound variables as indirection nodes of type LREC_IND. For each function a descriptor of type COMB (for combinator) is allocated, which contains a reference to the functions code. The top level goal expression is regarded as a combinator of arity 0. The instruction set of ASP will be explained later.

Code execution constructs a result graph which is transformed by the post–processor into a high–level program. The post–processor also performs full β–reductions if there are some top–level partial applications (closures) left over by the processor. By the execution of partial applications, new functions are created. The post–processor also resolves naming conflicts which may arise in the course of reducing partial applications.

Let us take as an example the expression $@\,\lambda x\, .\, \lambda y\, .\, @\, x\ y\ \ y$. This is transformed in the pre–processing phase into a supercombinator–graph which corresponds to $@\,\Lambda_x\, .\, \Lambda_y\, .\, @\ \#1\ \#0\ y$.

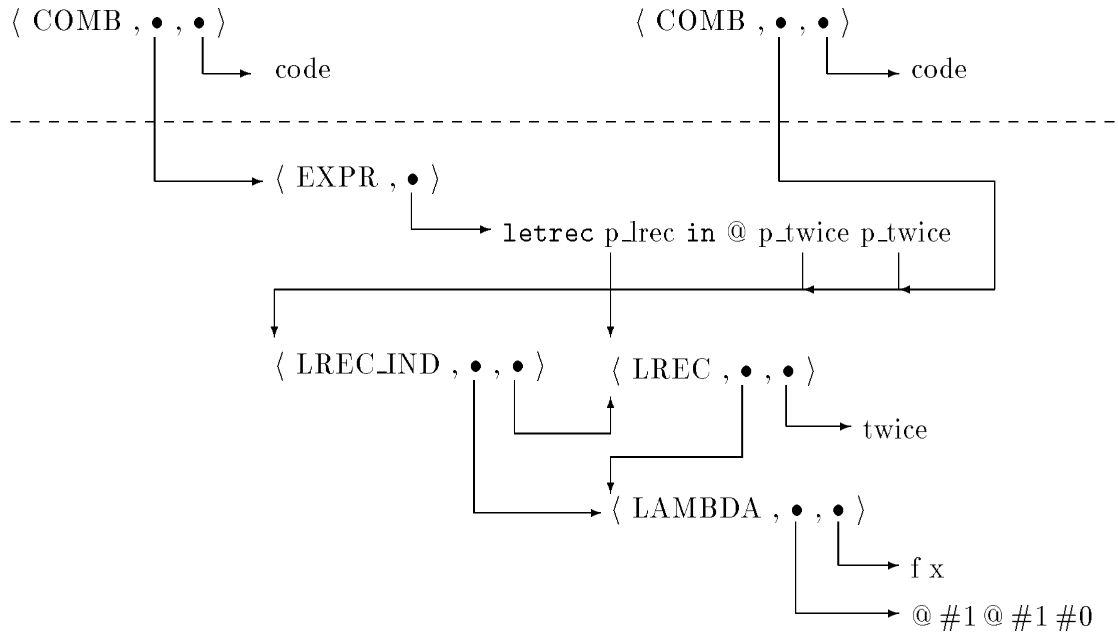

Figure 3: Graph– and code of the program `twice`

The partial application is not reduced in the processing phase and the graph is returned as it is for post–processing. The post–processor executes the partial application and produces in a first step $\Lambda_y \, . \, @ \, y \; \#0$. Next, the λ–binder $\lambda \, y$ is created. This causes a name clash with the free variable y, which is resolved by protecting it against binding: $\lambda \, y \, . \, @ \, \backslash y \; \#0$. Finally the de Bruijn–index #0 is substituted by the original variable name y yielding $\lambda \, y \, . \, @ \, \backslash y \; y$.

5 ASP: An Abstract Stack Processor

The abstract stack processor ASP has been designed to give direct support to some key techniques for the efficient execution of untyped functional languages. Efficient function call and parameter passing mechanisms are supported by a multiple–stack system. This stack system allows for an elegant optimization of general tail–recursion to jumps. It also makes it very easy to release unused parts of function argument frames from the stacks as early as possible during function evaluation in order to save stack and heap space an time spent on garbage collection.

Safe updating of (elements of) structured data, e.g arrays, requires (at least conceptually) to copy the data structure and perform the update on the copy. This makes updating of large data structures very expensive. However, some structure–transforming operations generate (completely) new result data structures, leaving the original arguments as garbage behind. Thus recursive functions which perform computations on structured data may produce large amounts of intermediate structures. When references to these intermediate structures are held in argument–frames of function invocations until termination, a garbage collector may have to copy all reachable structures. In most cases, large parts of the heap constitute garbage since the pointers held in the stacks are not used any more for further computations. Therefore it is desirable to release parts of the argument frames in the runtime stacks as early as possible during function evaluation in order to save heap space and time spent on garbage collection. Early releases of unused argument pointers, in a significant number of cases, renders it possible to perform updates of data structures in a destructive manner, i.e. without the need to copy entire structures, since most heap objects are referenced only once.

Direct support for function calls and parameter passing is provided by a system of four stacks called `A`, `W`, `R` and `H` respectively. The actual argument frame of the function under evaluation always resides on top of stack `A`, the argument stack, from where the arguments are accessed by indexed read operations. Stack `W`, the working stack, is used to allocate temporary registers for evaluating expressions and creating argument frames for further recursive function invocations. To enable fast parameter passing, the stacks `A` and `W` are interchangeable by a so–called stack–switch–operation. Stack–switching is done on function calls to pass the actual arguments, which have been computed by the calling function on stack `W`, to the called function on stack `A`. The called function itself reads its arguments form its actual stack `A` and computes its result on its actual stack `W`. On termination (return) the result is moved from stack `W` to stack `A`, and again a stack–switch is performed, which restores the old situation and returns the result on the working stack of the calling function. Stack `R` is used to store return–addresses on function calls, and stack `H` is used as special purpose stack for pattern matching.

6 Compilation and Code Execution

The basic instruction set of ASP is given below.

`PUSH c` pushes the item c onto the working stack `W`.

`PUSHARG i` reads the i–th item (the top element has index 0) of the argument stack `A` and pushes it onto the working stack `W`.

`MKLIST n` builds an n–ary list in the heap from the n top elements of the working stack `W`. The n elements are removed from the stack, and a pointer to the list is pushed.

`APPLY n` applies the top element of the working stack `W` (which is supposed to be a function symbol) to the n elements below. If the first element is a function, and there are enough arguments of appropriate type, the application is reduced, otherwise a closure is constructed in the heap. Function and arguments are replaced by the result. Supercombinator applications effect a stack–switch and a subroutine jump to the functions code.

`JFALSE l` continues code execution at label `l` if the top element of the working stack equals `false`. The top element of the stack is popped in any case.

`FREE n` pops n elements off the argument stack `A`. This instruction is used to explicitly release argument frames.

`RTF` performs a return from a function call. Also the result is moved from the working stack to the argument stack, and subsequently a stack–switch is performed. Thus the status of the stack system is the same as before the function call, and the result resides afterwards on the working stack of the calling function.

`EXIT` terminates the program execution.

These instructions are sufficient for the compilation of $\mathcal{M}$ini$\mathcal{R}$ed–programs. However it is possible to improve the code significantly with respect to runtime and stack/heap space consumption. The complex instruction `APPLY` can be replaced in many cases by simple `CALL`–instructions or primitive operations (`ADD, SUB` etc.) if this can be statically decided by the compiler. Stack and heap space consumption can be reduced significantly by placing the `FREE`–instructions as early as possible in the code–sequence. This can be done easily since the actual argument frames of function invocations always reside on top of the (actual) argument stack `A`.

It is also possible to detect general tail–end recursive calls and optimize them to jumps (`JTAIL`). This requires some slight modification of the semantics of the `RTF` instruction. Since optimized tail jumps also require stack–switching to pass parameters, but with every `JTAIL` instruction the execution of one RTF instruction is 'optimized away', some (but not all) RTF instructions must not execute a stack–switch operation. It cannot be decided during compilation when stack–switches are required and when not. However, there is a simple dynamic method to overcome this problem. A single bit in the return address is used as a 'tail–flag'. Normal CALL instructions leave this tail–flag unset. When a JTAIL is executed, the tail–flag of the topmost return address on the return stack R is complemented. A set tail–flag indicates that the stack–switch must be suppressed when executing a RTF–instruction.

Compilation into abstract machine code and code execution are explained by means of an application of the left–associative Fold–Function Foldl:

```
def
  Foldl [ L , Op , N ] = if empty( L )
                         then N
                         else Foldl [ tl( L ) , Op , ( N Op hd( L ) ) ]
in Foldl [ <1,2,3> , + , 0 ].
```

The abstract code for the function Foldl is shown in figure 4. The primitive Operations EMPTY, HEAD, TAIL and JFALSE are typed, i.e. if the parameters are not of appropriate type (this is checked dynamically) the operation cannot be performed and a closure, representing the application of the primitive function to its ill–typed arguments, is constructed and returned instead.

```
c_Foldl : PUSHARG 0     /* L */
          EMPTY         /* empty(L) */
          JFALSE Label  /* jump if not empty */
          FREE 2        /* drop L and Op from A */
          MOVE          /* move N from A to W (same as PUSHARG 0; FREE 1) */
          RTF           /* (switch stacks, move and) return */
Label :   PUSHARG 0     /* push L onto W */
          HEAD          /* compute hd(L) */
          PUSHARG 2     /* push N onto W */
          PUSHARG 1     /* push Op onto W */
          APPLY 2       /* compute (N Op hd(l)) */
          PUSHARG 1     /* push Op onto W */
          MOVE          /* move L from A to W */
          FREE 2        /* drop Op and N from A */
          TAIL          /* compute tl(L) */
          JTAIL c_Foldl /* compute Foldl[Op,(N Op hd(l)),tl(L)] */
          RTF           /* (switch stacks, move and) return */
```

Figure 4: The abstract machine code generated for Foldl

User–defined functions are untyped. In order to allow stepwise reduction, the execution of function calls (in our example JTAIL c_Foldl) is controlled by a so called reduction counter. When greater zero it is decremented by one on each call, otherwise the call is suppressed, and a closure is constructed instead. The representation of the function, including the original variable names, the graph representation of its body and the function code can be accessed via the label c_Foldl (this can be implemented e.g. by storing a pointer p_Foldl to the function representation at memory location c_Foldl-1).

The graph representation of a compiled function consists of a graph node (called descriptor), which contains a tag, the functions arity, and references to the functions graph and the functions code. The code is a closed thread terminated in every branch by a return–instruction and thus does not depend on the context of the function definition. The only pre–requisite for calling a function is that the right number of arguments reside on top of the working stack.

Nr.	Instruction	Stack1	Stack2	Stack3	redcnt
1	PUSH 0	W: 0	A:	R:	2
2	PUSH +	W: 0 +	A:	R:	2
3	PUSH <1,2,3>	W: 0 + <1,2,3>	A:	R:	2
4	CALL c_Foldl	A: 0 + <1,2,3>	W:	R: @EXIT	1
5	PUSHARG 0	A: 0 + <1,2,3>	W: <1,2,3>	R: @EXIT	1
6	EMPTY	A: 0 + <1,2,3>	W: false	R: @EXIT	1
7	JFALSE Label	A: 0 + <1,2,3>	W:	R: @EXIT	1
8	PUSHARG 0	A: 0 + <1,2,3>	W: <1,2,3>	R: @EXIT	1
9	HEAD	A: 0 + <1,2,3>	W: 1	R: @EXIT	1
10	PUSHARG 2	A: 0 + <1,2,3>	W: 1 0	R: @EXIT	1
11	PUSHARG 1	A: 0 + <1,2,3>	W: 1 0 +	R: @EXIT	1
12	APPLY 2	A: 0 + <1,2,3>	W: 1	R: @EXIT	1
13	PUSHARG 1	A: 0 + <1,2,3>	W: 1 +	R: @EXIT	1
14	MOVE	A: 0 +	W: 1 + <1,2,3>	R: @EXIT	1
15	FREE 2	A:	W: 1 + <1,2,3>	R: @EXIT	1
16	TAIL	A:	W: 1 + <2,3>	R: @EXIT	1
17	JTAIL c_Foldl	W:	A: 1 + <2,3>	R: @EXIT'	0
18	PUSHARG 0	W: <2,3>	A: 1 + <2,3>	R: @EXIT'	0
19	EMPTY	W: false	A: 1 + <2,3>	R: @EXIT'	0
20	JFALSE Label	W:	A: 1 + <2,3>	R: @EXIT'	0
21	PUSHARG 0	W: <2,3>	A: 1 + <2,3>	R: @EXIT'	0
22	HEAD	W: 2	A: 1 + <2,3>	R: @EXIT'	0
23	PUSHARG 2	W: 2 1	A: 1 + <2,3>	R: @EXIT'	0
24	PUSHARG 1	W: 2 1 +	A: 1 + <2,3>	R: @EXIT'	0
25	APPLY 2	W: 3	A: 1 + <2,3>	R: @EXIT'	0
26	PUSHARG 1	W: 3 +	A: 1 + <2,3>	R: @EXIT'	0
27	MOVE	W: 3 + <2,3>	A: 1 +	R: @EXIT'	0
28	FREE 2	W: 3 + <2,3>	A:	R: @EXIT'	0
29	TAIL	W: 3 + <3>	A:	R: @EXIT'	0
30	JTAIL c_Foldl	W: (@ p_Foldl <3> + 3)	A:	R: @EXIT'	0
31	RTF	W: (@ p_Foldl <3> + 3)	A:	R:	0
32	EXIT	W: (@ p_Foldl <3> + 3)	A:	R:	0

Figure 5: The execution of the code for Foldl

Figure 5 shows the execution of the code for the application `Foldl[<1,2,3>,+,0]`. The left hand side of the table lists the executed instructions and on the right hand side lists the contents of the stacks after instruction execution. Stack–switching occurs in lines 4 and 17. The `JTAIL`–instruction in line 17 sets the tail–flag in the return–address `@EXIT`, which is symbolized by a quote. The second `JTAIL`–instruction in line 30 is suppressed because the reduction counter `redcnt` is exhausted. Instead the application graph of `Foldl` to its arguments is constructed. The return instruction in line 31 does not execute a stack–switch since the tail–flag in the return–address is set. The resulting application is finally converted by the post–processor into the program:

```
def
  Foldl [ L , Op , N ] = if empty( L )
                         then N
                         else Foldl [ tl( L ) , Op , ( N Op hd( L ) ) ]
in Foldl [ <3> , + , 3 ].
```

7 Reducing to Normal Forms

As it is discussed so far, π–RED$^+$ performs during the processing phase top–level supercombinator reductions which terminate with graph representations of weak normal forms. They include closures left over from partial supercombinator applications which are reduced by the post–processor to weak normal forms of λ–abstractions, using full–fledged β–reductions. This is to say that

- what originally was a function of n formal parameters is turned into a new function of some $n-k$ parameters by the substitution of some $k < n$ argument terms,
- all potential situations of name clashes among free variable occurrences in the k argument terms that have been substituted and the remaining $n-k$ formal parameters of the new function are resolved in an orderly form,
- no further reductions are performed in the body of the new function.

However, since π–RED$^+$ makes full use of the referential transparency inherent in purely functional programs, the focus of control may be manually shifted, in the resulting high–level representation of the new abstraction, to the topmost application of the function body, and another reduction sequence may be started in this new context which, again, terminates with a WNF. This interactive process may be repeated until a normal form of the entire program is reached. Unfortunately, depending on the complexity of an application program, particularly on the nesting depths of function definitions, the number of user interactions to accomplish this may be fairly high, i.e. the entire process may become rather time–consuming and hence boring. Moreover, on each of these interactions π–RED$^+$ moves through the full cycle of pre–processing, compiling, processing and post–processing. This is particularly wasteful and annoying insofar as generally very little is changed in the internal program representation when shifting the focus of control to another subexpression.

We will therefore describe how shifting the focus of control recursively into the bodies of top–level abstractions can be mechanized so that normal forms are being computed in one pass and without repeatedly pre–processing, compiling and post–processing large parts of the program over and over again.

Reducing program expressions to normal forms rather than just to weak normal forms does not appear to be compatible with supercombinator reductions for it generally takes full β–reductions to deal with partial applications. However, with a little trick supercombinator reductions need not stop at top–level partial applications but may continue in instantiated body terms of what are to become the new functions. This can be accomplished without creating name clashes at run–time. All there is to do conceptually is to make a partial supercombinator application of the general form

$$\underbrace{@\ldots@}_{k}\Lambda_{x_1}\ldots\Lambda_{x_n}.\,body\ a_1\ldots a_k \text{ with } (k < n)$$

the subject of an $(n-k)$–fold η–extension:

$$\lambda x_{(k+1)}\ldots\lambda x_n.\,\underbrace{@\ldots@}_{n}\Lambda_{x_1}\ldots\Lambda_{x_n}.\,body\ a_1\ldots a_k\ x_{(k+1)}\ldots x_n$$

It creates a new function (abstraction) of the parameters $x_{(k+1)},\ldots,x_n$ in the body of which the partial application is turned into a full supercombinator application, with $x_{(k+1)},\ldots,x_n$ filling in for the missing argument terms. Thus, when reducing this application, all free occurrences of $x_{(k+1)},\ldots,x_n$ in *body* are naively substituted by the argument terms $a_1,\ldots,a_k$, respectively, and all free occurrences of $x_{(k+1)},\ldots,x_n$ are replaced by themselves (or left as they are).

Further partial applications that may be encountered in the course of reducing the instantiated body must be dealt with in the same way until all of them are done.

This scheme can be elegantly integrated into π–RED$^+$. Here supercombinator reductions are performed by executing compiled code which constructs a result graph. A partial application as above results in a closure which, in addition to (pointers to) the argument terms includes a pointer to a descriptor of the function (compiled to a supercombinator) which in turn, contains a pointer each to the function code, to the graph representation of its body term, and to the list of its formal parameters (see also figure 6).

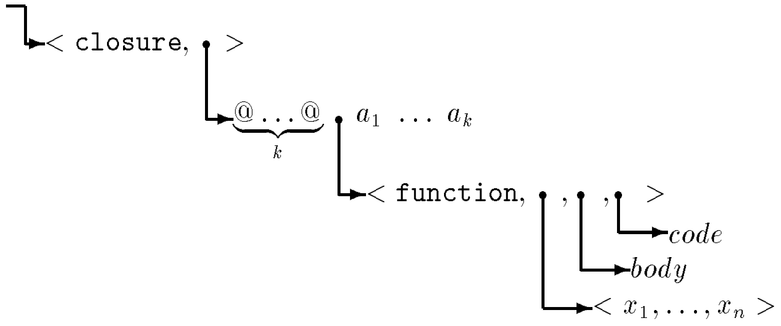

Figure 6: The graph of a partial application

Rather than terminating the processing phase in such a situation and handling the closure over to the post–processor (as it has been done so far), an interpreter takes over to

- create from the components of the closure an argument frame by pushing onto the actual argument stack first the available argument terms $a_1, \ldots, a_k$ and then the variables $x_{(k+1)}, \ldots, x_n$ in this order,
- invoke the processing phase again to execute the function–turned–supercombinator code in the environment set up on the argument stack, which reduces the instantiated body to its weak normal form,
- recursively invoke this scheme to reduce nested partial applications in order to compute a normal form of the body expression,
- construct an $(n-k)$–ary abstraction of the variables $x_{(k+1)}, \ldots, x_n$ with this normal form as its body term,
- pass this abstraction as the normal form of the entire program to the post–processor.

Thus, the path taken by a program through the various processing stages of the extended π–RED$^+$ may be depicted as shown in figure 7, of which the feedback cycle of the processing phase is used as many times as there are partial applications to be reduced.

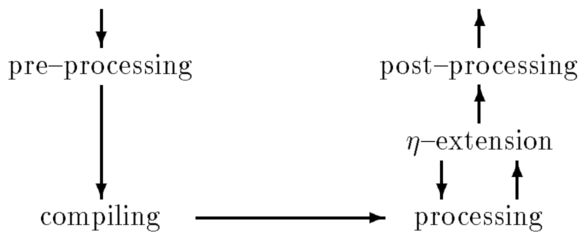

Figure 7: The extended π–RED$^+$ system

Though this evaluation scheme looks very simple, there are some strings attached: η–extending a partial supercombinator application, of course, may inflict name clashes between free variable occurrences in the argument terms and the bindings which are added. Resolving these name clashes the classical way, i.e. by renaming or by manipulating binding levels similar to de Bruijn indices, in fact means that full β–reduction need to be performed during the processing phase. However, this is exactly what is to be avoided for complexity reasons.

The way out of this dilemma consists in a new indexing scheme which, contrary to de Bruijn indices, counts binding levels from outermost to innermost, i.e. in the same direction in which binders for variables build up due to η–extensions. This scheme has the advantage that binding levels, once introduced, need not be changed at run–time. However, it should be emphasized that this indexing scheme can only be used in the context of η–extensions which are done strictly from outermost to innermost. We will therefore refer to them as η–binding levels or, more approximately, as η–nesting levels. The binding structures thus introduced must be distinguishable from those specified by de Bruijn indices since they must be treated differently by the post–processor.

A simple example may help to illustrate the idea: Suppose the processing phase terminates with a graph representation of a top–level partial application of the form

$$@\ \Lambda_x\Lambda_x.@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ \#0\ a$$

of which the argument term a may contain occurrences of globally free variables. The body of the supercombinator includes another partial supercombinator application.

Using Reduce and Eta as names for system functions which force reductions and η–extensions, respectively, of an expression, the above application is processed as follows:

$$
\begin{array}{lll}
 & \textsc{Reduce}(@\ \Lambda_x\Lambda_x.@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ \#0\ a) & \\
\rightarrow & \textsc{Eta}(@\ \Lambda_x\Lambda_x.@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ \#0\ a) & \\
\rightarrow & \lambda^{(1)}x.\textsc{Reduce}(@\ @\ \Lambda_x\Lambda_x.@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ \#0\ a\ x^{(1)}) & \text{, } \eta\text{–extension with } x \text{ (binding level 1)} \\
\rightarrow & \lambda^{(1)}x.\textsc{Reduce}(@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ x^{(1)}) & \text{, one supercombinator reduction step} \\
\rightarrow & \lambda^{(1)}x.\textsc{Eta}(@\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ x^{(1)}) & \\
\rightarrow & \lambda^{(1)}x.\lambda^{(2)}x.\textsc{Reduce}(@\ @\ \Lambda_x\Lambda_x.@\ \#1\ \#0\ x^{(1)}\ x^{(2)}) & \text{, } \eta\text{–extension with } x \text{ (binding level 2)} \\
\rightarrow & \lambda^{(1)}x.\lambda^{(2)}x.@\ x^{(1)}\ x^{(2)} &
\end{array}
$$

Here the η–nesting levels of both the variables and binders introduced by η–extensions are annotated as superscripts. The resulting expression is a new abstraction in which binding levels are represented as η–nesting indices which relate variable occurrences to the correct binder(s).

The abstraction is now transferred to the post–processor which, among other things, protects variable occurrences by means of λ–bars against all surrounding bindings of identically named variables and then deletes as much as λ–bars as the η–binding level determines. This is to say that identically named variable occurrences which belong to different binders are in the resulting program expression distinguished by indices which count binding levels from innermost to outermost.

Thus the resulting program term

$$\lambda^{(1)}x.\lambda^{(2)}x.@\ x^{(1)}\ x^{(2)}$$

of the above example is in a first step converted into

$$\lambda^{(1)}x.\lambda^{(2)}x.@\ \backslash\backslash x^{(1)}\ \backslash\backslash x^{(2)},$$

from which as many λ–bars have been consumed as are specified by η–nesting levels. Hence, the resulting program term is

$$\lambda x.\lambda x.@\ \backslash x\ x.$$

Top–level abstractions returned after a pass through the processing phase are merely special cases of partial applications: they are not applied to any arguments at all. η–extension applies them to the complete set of abstracted variables and sets up the full binding structure in front of this application. Thus reducing them to normal forms can accomplished in the same way.

In case of structured objects (e.g. lists or applications with no abstraction in function position) the reduction has to be recursively driven into each component.

As indicated in figure 7, reductions to normal forms essentially require that the processing phase which implements the function REDUCE be supplemented by an interpreter phase which does the η–extensions as formally specified by the function ETA. What this interpreter does is basically described on page 11: For every partial application (or abstraction) received from the processing phase, it creates on the argument stack an argument frame for a full application by filling the missing argument position with the respective bound variables (or formal parameters). These variables are represented as pointers to tuples which include the variable names and the η–nesting levels. These pointers are also pushed into a run–time stack which is supported by the interpreter in order to handle the recursive traversal of the expression. On each call of the interpreter, the η–nesting level for each formal parameter is incremented by one, starting with the value zero.

$$\bullet\!\!\longrightarrow < var\_name, \eta - nesting\ level >$$

Figure 8: The graph structure of a variable introduced by η–extension

In another pass through the processing phase, the reduction of the full supercombinator application which constitutes the body of the η–extended abstraction may now be resumed. This is done by pushing onto the return stack the address of an artificially created EXIT–instruction and then calling the supercombinator code. This code eventually terminates by executing a RETURN–instruction which pops the address of the EXIT–instruction.

When doing this repeatedly in case of nested partial applications, the formal parameters of several nested η–extensions may pile up on the run–time stack. If the body–expression returned from the processing phase is finally a normal form, i.e. no further η–extensions need to be done, the interpreter pops the variables held in its run–time stack to put the λ–binders in front of this expression in order to complete the η–extensions. This scheme handles nested partial applications as well as partial applications on the same nesting–level. The expression thus obtained is then passed on to the post–processor.

8 Conclusion

π–RED$^+$ constitutes a considerable improvement over the fully interpreting graph reducer π–RED both with respect to its performance and the semantics of the language it supports. Programs are compiled to code of an abstract machine ASP which uses multiple stacks for fast function calls, optimized general tail–end recursions and earliest possible release of argument graphs that are no longer needed. The latter feature drastically reduces the demand for heap space, increases run–time efficiency due to less frequent garbage collection phases, and, most importantly, in conjunction with reference counting allows for many in–place updates of large data structures which conceptually would require the creation of new objects. Depending on particularlies of the application programs, the performance of π–RED$^+$ is enhanced by factors ranging from 2 to 8 over the interpreter version π–RED. Another important difference to π–RED relates to a small extension of π–RED$^+$ by means of which all programs are now reduced to normal forms rather than to weak normal forms. The extension concerns an interpreter which η–extends partial supercombinator applications so that they can in fact be reduced as full applications, using the statically compiled ASP code without any modifications.

References

[Aßm92] C. Aßmann. *Eine Prozessor Architektur mit einem neuartigen Stacksystem.* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1992.

[Ber76] K.J. Berkling. A Symmetric Complement to the Lambda Calculus. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, September 1976. ISF-76-7.

[BSK92] H. Blödorn, C. Schmittgen, and W.E. Kluge. π–RED* a Graph–Reducer for a Full–Fledged λ–Calculus. to appear in New Generation Computing, OHMSHA Ltd/Springer, 1992.

[dB72] N. G. de Bruijn. Lambda–Calculus Notation with Nameless Dummies. A Tool for Automatic Formula Manipulation with Application to the Church–Rosser–Theorem. *Indigationes Matematicae*, 34:381–392, 1972.

[Fai86] J. Fairbairn. A simple abstract machine to execute supercombinators. In J. H. Fasel and R. M. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 49–52. Springer–Verlag, 1986.

[Gär91] D. Gärtner. *π–RED$^+$: ein interaktives codeausführendes Reduktionssystem zur vollständigen Realisierung eines angewandten λ-Kalküls.* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1991.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, 1984.

[Joh85] T. Johnsson. Lambda Lifting: Transforming Programs into Recursive Equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190 – 203. Springer–Verlag, 1985.

[KIR89] *The Kiel Reduction Language, User's Guide*, June 1989. Preliminary Version.

[SKZ89] C. Schmittgen, W.E. Kluge, and R. Zimmer. π–RED – An Interactive Reduction System Based on a Full–Fledged λ–Calculus. In D. DeGroot and P. Biswas, editors, *ISCA Workshop on Architectural Support for Declarative Programming*, May 1989.

[SP90] H. Schlütter and E. Pless. Die Reduktionssprache OREL/2. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, 1990.

[Zim90] R. Zimmer. *Zur Pragmatik eines operationalisierten λ-Kalküls als Basis für interaktive Reduktionssysteme.* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1990.