J. Functional Programming **1** (1): 1–000, January 1993

π-RED$^+$
An Interactive Compiling Graph Reduction System for an Applied λ–Calculus

Dietmar Gärtner and Werner E. Kluge
Christian-Albrechts-Universität Kiel, Institut für Informatik,
D–24105 Kiel, Germany,
Email: wk@informatik.uni–kiel.d400.de

Abstract

This paper describes a compiling graph reduction system which realizes the reduction semantics of a fully-fledged applied λ-calculus. High-level functional programs are conceptually executed as sequences of program transformations governed by full β-reductions. They may be carried out step by step, and intermediate programs may be displayed in high-level notation, rendering the system suitable for interactive program design, high-level debugging, and also for teaching basic programming language concepts and language interpretation.

Run-time efficiency for production runs is achieved by means of an abstract stack machine ASM which serves as an intermediate level of code generation. It employs multiple stacks for reasonably fast function calls, optimized tail-end recursions, and earliest possible releases of subgraphs that are no longer needed. The ASM involves an interpreter if and only if potential naming conflicts need to be resolved when reducing partial function applications.

1 Introduction

Reduction languages are a subclass of functional languages whose semantics are directly and completely defined by the rewrite (reduction) rules of the λ-calculus or of a combinatory calculus (Church, 1941; Curry, 1934; Hindley and Seldin, 1986; Backus, 1978). Conceptually, program execution takes place entirely within the domain of programs. It is a process of meaning-preserving program transformations which for all semantically meaningful programs eventually terminates with a result which is itself a program. All rewrite rules may be applied in any part of a program without causing side-effects in other parts (referential transparency).

Reduction systems which implement this execution model can be made to reduce, under interactive control, programs step by step, and to return as output sequences of high-level intermediate programs. These programs may be inspected and modified, and the focus of control may be freely moved about them to evaluate, in any chosen order, other than top-level redices. Apart from termination problems, the resulting programs remain invariant against the order in which reduction steps are carried out.

The virtues of this concept are well recognized. It may be used to validate program correctness and correct execution, and also to teach, in a clean setting, basic programming language concepts and language interpretation.

In a recent paper, a strong case has been made for symbolic calculators as tools for computer-based learning (Goldson, 1994). It describes a system called MiraCalc which allows step-by-step transformations of MIRANDA programs. It is argued that students can be expected to benefit considerably from viewing functional computations as rule-based evaluation processes. This may help to develop a better understanding of basic programming principles and generally improve their ability to specify, design and reason about programs.

Stepwise program execution and inspection of intermediate program terms in high-level notation is also an established technique of debugging LISP programs (Sun, 1990; Allegro, 1992). Debuggers for COMMONLISP usually include tools for tracing function calls, for displaying the actual parameters passed on and the values returned by each call, for stepping through the evaluation of program terms, and for the inspection and modification of the elements of data structures. However, owing to the state transition semantics of LISP, this cannot be entirely done within the space of programs. There must also be means to display (parts of) the actual environment.

Almost all functional systems proposed or implemented to date employ some form of compiled graph reduction (Johnsson, 1984; Cardelli and McQueen, 1983; Peyton Jones, 1987; Fairbairn and Wray, 1987; Plasmeijer and van Eekelen, 1993; Peyton Jones, 1992). To measure up against conventional systems, they are tuned to compute with competitive speed (sequences or arrays of) basic values. The languages are usually typed, globally free variables are outlawed, and user-defined functions are converted into super-combinator terms (or other, semantically equivalent representations) to avoid at run-time scoping problems due to name clashes, and to simplify compilation to fast code. No provisions are made to allow for a stepwise execution mode. Executing compiled code which has undergone various optimizations usually cannot be directly and unambiguously related to reduction steps of the underlying calculus. It renders de-compiling intermediate states of the computation into equivalent high-level programs a difficult problem. The code must run to completion to produce meaningful and intelligible results.

Irrespective of a prior conversion into supercombinator terms, compilation to conventional machine code to some extent also separates the world of functions from the world of objects (the graphs) they operate on. Functions may be applied to functions as arguments, but there is no direct way of computing new functions, say, as results of partial applications, let alone getting them reduced to normal forms and returned in high-level notation as output. Functions compiled to code are static objects which must be supplied with full sets of arguments to execute correctly. They change whatever they are applied to, but do not modify themselves. Partial function applications need to be packed into closures and in this form passed along until eventually the missing arguments can be picked up. If the computation terminates with a closure, the user is simply notified of the fact that the result is a function, but it cannot be made visible in high-level notation (e.g., as a λ-abstraction) (Turner, 1986; Harper and MacQueen and Milner, 1986; Hudak et. al., 1988).

Further restrictions come with the polymorphic or monomorphic type systems. They rule out certain higher-order functions, especially self-applications, as they cannot unify recursive types.

In SK-combinator reduction systems the problems are of a different nature (Turner, 1979). Source programs of an applied λ-calculus are here compiled, by abstraction of bound variables, to combinator terms which are directly taken as executable codes. Combinator terms may be freely applied to others and reduced to (weak) normal forms. Since they are in fact curried, partial applications of what in high-level notation were n-ary functions can be reduced to new combinator terms equivalent to functions of lesser arities. However, combinator terms other than basic values are hardly intelligible, and they cannot generally be converted back into comprehensible high-level programs either. Major difficulties arise from the granularity of SK-combinator reduction steps, many of which cannot be related to high-level program transformations as they would be effected, say, by function calls carried out in one conceptual step. Also, with all user-defined variables gone, unique new variables would have to be introduced upon de-compilation, which may alienate the resulting high-level programs even further. Restoring the old variables, which for this purpose would have to be carried along with the combinators, say, by means of descriptors, may inflict name clashes which cannot be resolved within the framework of combinatory logic.

Essentially the same problems arise with categorial combinator reduction systems as well (Cousineau and Curien and Mauny, 1987).

Computing normal forms of functions and returning them as output in the same high-level notation in which the original program was specified generally requires a fully-fledged λ-calculus which treats both functions and variables as first class objects. When freely applying functions (λ-abstractions) to other functions or to themselves, and specifically when doing reductions within function body terms, name clashes may have to be resolved by (the equivalent of) full β-reductions. These clashes primarily come about when reducing partial applications of λ-abstractions, and generally when substituting terms with free variables into other terms which contain open abstractions as subterms.

The demand for a full λ-calculus also arises when selecting, under a stepwise execution mode, other than top-level redices for evaluation. In the chosen subterms, there may be variables that are locally free but bound in a larger context, and (some of) these variables may not yet be instantiated. Dealing with them as they are again requires full β-reductions to resolve potential naming conflicts. Alternatively, the system could simply refuse to perform reductions in program terms which are not fully instantiated, or ask the user to artificially instantiate free variables out of context (as is common practice in LISP systems), neither of which is a very satisfactory solution.

These considerations led us to develop a reduction sytem π-RED$^+$ which, as far as its appearance to the user is concerned, truly realizes the reduction semantics of a fully-fledged λ-calculus. The basic concepts of this system have been adopted from Berkling's λ-calculus machine developed as early as 1975, which used string reduction mechanisms (Berkling, 1975).

π-RED$^+$ may be interactively controlled to perform on a program (or on any selected subterm of it) some pre-specified number of reductions, after which an intermediate program term is being returned in high-level notation. Computing in one conceptual step the normal form of the entire program is merely a matter of allowing for a sufficiently large number of reductions. Programs which have been debugged and validated step by step are thus guaranteed to behave exactly the same when used for production runs on one and the same machinery.

An earlier version called π-RED* achieves these ends by high-level interpretation of λ-terms (Schmittgen and Bloedorn and Kluge, 1992). To avoid whenever possible the complexity of full β-reductions, this system resorts to some loose form of closing λ-terms and to reducing at run-time only full function applications. β-reductions are called upon to reduce to new functions top-level partial applications if and only if everything else is done, i.e., the new functions are not subject to further applications.

π-RED$^+$ shares with π-RED* the concept of closing λ-terms and (as a consequence) some pre- and post-processing functions, but rigorously employs compiled graph reduction at run-time to improve the performance of the system by about an order of magnitude, as compared with interpretation. All programs may be reduced to normal forms (provided they exist). Partial applications of λ-abstractions that pop to top level are η-extended to full applications for further code-controlled reductions. All potential name clashes are correctly resolved by means of an indexing scheme for identically named variables that are bound in different scopes.

A tailor-made abstract stack machine ASM serves as an intermediate level of code generation. It employs four stacks to accommodate the run-time environment. One is used for return addresses, another two hold argument and workspace frames for function calls, and the last one holds the frames for instantiations of non-local variables. As compared with a one-stack abstract machine, this approach simplifies compilation to intermediate code and the de-compilation of machine states resulting from partially executed code into high-level programs. It also leaves several options open for compilation to target machine code. The four stacks may be either implemented as they are or merged into a single stack, (parts of) the actual topmost stack frames may be placed into processor-internal register files, etc..

In the sequel, we will proceed as follows: in the next section, we will introduce a high-level kernel language MINIRED and briefly explain, by means of an example, how programs of this language may be stepwise reduced by π-RED$^+$. Section 3 describes the program execution phases of π-RED$^+$, and section 4 specifies the underlying abstract stack machine ASM. In sections 5 and 6 we define the compilation of MINIRED terms into ASM code, including a brief overview of code optimizations. Code execution by the ASM will be explained in section 7, particularly the problem of re-constructing high-level programs from intermediate machine states, using the compiled program of section 2. In section 8 we will describe in some detail how

π-RED$^+$ reduces partial applications of λ-abstractions and, in doing this, resolves potential name clashes, and how normal forms can be computed eventually (provided they exist). In section 9 we will give some comparative performance figures for π-RED$^+$ and other implementations of functional languages.

2 Program Transformation in π-RED$^+$

π-RED$^+$ is primarily designed to support untyped, statically scoped and strict functional languages. The choice of a strict regime is mainly motivated by pragmatic considerations. It is easier to implement, particularly with respect to stepwise program execution, generally consumes less memory space, executes faster than a lazy regime, and is quite appropriate for the majority of real-life applications.

For the purpose of this paper it suffices to introduce a simple high-level language MINIRED with the following syntax:

$$e = c \mid x \mid \lambda\, x_1 \ldots x_n.e \mid (\, e_0\, e_1\, \ldots\, e_n\,)$$
$$\mid \text{IF}\, e_0\, \text{THEN}\, e_1\, \text{ELSE}\, e_2 \mid \text{LETREC}\, f_1 = e_1 \ldots f_m = e_m\, \text{IN}\, e_o \quad .$$

It is an applied λ-calculus whose terms may be either constants or primitive function symbols (denoted as c), variables (denoted as x), n-ary λ-abstractions [1], applications of terms e_0 in function position to n argument terms $e_1\ ,\ldots\ ,\ e_n$, IF_THEN_ELSE clauses, or sets of some m mutually recursive function definitions followed by body terms e_0, respectively. Note that all applications are given in prefix notation, and that the parentheses may be dropped if no ambiguities can occur.

The semantics of MINIRED demands that all programs be reduced, under a strict (applicative order) regime, to normal forms (provided they exist), i.e., to MINIRED terms which contain no more redices. To do so, n-ary applications $(\, e_0\, e_1\, \ldots\, e_n\,)$ are by π-RED$^+$ reduced as follows:

First the component terms $e_0, e_1, \ldots, e_n$ are reduced to their normal forms in the order from right to left. This order has been chosen with respect to a convenient compilation, otherwise it is of no relevance: the terms could be reduced in any order. As long as there are other redices within the program, λ-abstractions emerging in any of the components remain in weak normal form, i.e., no reductions are being performed in their body terms. In either case, the resulting terms are denoted as $e_0^N, e_1^N, \ldots, e_n^N$, respectively.

If e_0 is, or evaluates to, an abstraction $\lambda\, x_1 \ldots x_r.e$, then the application $(\, \lambda x_1 \ldots\ x_r.e\, e_1^N\, \ldots\, e_n^N\,)$ reduces in one more conceptual step comprising, in general, the equivalent of several β-reductions to

- the body e of the abstraction, instantiated with $e_1^N, \ldots, e_n^N$ for free occurrences of the variables $x_1, \ldots, x_r$, respectively, if $r = n$;
- a new abstraction $\lambda\, x_{(n+1)} \ldots x_r.e'$, with e' emerging from e by instantiation

[1] Whenever appropriate, we will also use the equivalent curried version $\lambda x_1 \lambda x_2\ \ldots\ \lambda x_n.e$

of free occurrences of the variables $x_1, \ldots, x_n$ with the terms $e_1^N, \ldots e_n^N$, respectively, if $r > n$;

- an application $(e^c \, e_{(r+1)}^N \ldots e_n^N)$, where e^c is the result of reducing $(\lambda x_1 \ldots x_r.e \; e_1^N \ldots e_r^N)$ to (weak) normal form, if $r < n$;

If e_0 is, or evaluates to, a LETREC-bound identifier, it must simply be replaced by a copy of the respective λ-abstraction, in the body of which all free occurrences of the identifier are substituted by copies of the entire LETREC-construct under which it is defined.

Applications of primitive functions are handled in a similar way: full applications reduce to basic values, partial applications reduce to partially instantiated functions of lesser arities, and applications whose arities exceed those of the functions reduce to applications of lesser arities.

Following these reduction rules, an n-ary application specified in curried form, say as $(\ldots((\lambda x_1 \ldots x_r.e \, e_1) \, e_2) \ldots \; e_n)$, first gets its argument terms reduced to (weak) normal forms from outermost to innermost (or again from right to left), and then gets the nested applications reduced from innermost to outermost by (the equivalent of) up to n individual β-reduction steps. The resulting (weak) normal form is invariant against nesting levels; what differs is merely the number of reduction steps to reach it.

λ-abstractions are reduced to normal forms if and only if nothing else is left to do. This is primarily the case whenever a program reduces to an abstraction, but also whenever abstractions occur in argument positions of applications which have terms other than abstractions in function positions. These terms could be primitive functions which are not type-compatible with, and therefore are not applicable to, abstractions, or constant terms other than functions which cannot be applied to anything (such applications are perfectly legitimate if the language is untyped). π-RED$^+$ leaves these applications as they are, except that it reduces to normal forms all left-over abstractions in argument positions (which up to this point are only in weak normal form).

MINIRED program transformations as they can be made visible step by step at the user interface of π-RED$^+$ follow exactly these reduction rules.

To explain some basic operating principles of π-RED$^+$, compilation to code of the underlying abstract machine, and code execution, the following small MINIRED program will be used as a running example throughout the remainder of the paper.

```
LETREC
  f = λuv.LETREC
             g = λwz.IF GT u w THEN g ( − 1 u ) z ELSE f v ( + 1 w )
          IN g v u
IN f 1 2
```

This program consists of two mutually recursive functions f and g, of which the latter is local to the former, using u and v as non-local parameters. It is free of naming conflicts as all formal parameters are unique and all function calls are supplied with full argument sets [2]. Since f calls g, and g calls either f or g, depending on the actual values substituted for w and u, the program never terminates and thus is semantically meaningless. However, it allows us to expose in a nutshell some essential implementation features of π-RED$^+$.

One of these features concerns the treatment of non-terminating recursions. Rather than simply aborting them by time-out conditions, π-RED$^+$ deals with this problem in a more orderly form. It provides a special counting mechanism for reduction steps. Prior to starting the execution of a program, this mechanism must be initialized with some integer value which defines an upper bound on the number of reductions to be performed. Each reduction step decrements this value by one. Program execution is halted either after the count value is down to zero or after having reached a normal form, whichever occurs first. The program term computed at the halting point is returned as the result. If the program is prematurely terminated by exhaustion of the count value, it is left to the user to decide whether or not to re-submit it for another such sequence of reduction steps, to which we will also refer as a shift of reductions. If the program is guaranteed to terminate, the count value must for production runs simply be chosen large enough to allow for reductions to proceed to normal forms.

The very same mechanism may also be employed to set breakpoints, say, for debugging or validation purposes. The programs returned at such breakpoints may not only be inspected but also modified, say, to introduce specific argument terms for function calls, and the focus of control may be moved to other than top-level redices.

Fig. 1 shows the first three reduction steps of the above program, as it can be followed up on the user interface of π-RED$^+$. The counting mechanism is set up to count (or set breakpoints at) calls of LETREC-defined functions, and is each time initialized with the value one.

Each reduction step yields a new program which results from expanding the function application in the body term of the outermost LETREC by the instantiated right-hand side of the respective function definition, and by reducing this term to the point where another LETREC becomes the top-level term. As said before, instantiating a function body includes both the substitution of formal by actual parameters and the expansion of free occurrences of the LETREC-bound function identifier by the entire LETREC-construct under which the function is actually defined, which is equivalent to reducing applications of Y-combinators.

Note that this program has the interesting property of reproducing itself after three function calls with the parameters of the outermost application of f interchanged. Another two reduction steps restore the original program.

Stepwise execution generally creates a practical problem with recursive programs. Intermediate program terms may easily expand to the extent that they cannot be fully displayed on a screen. To deal with this problem, π-RED$^+$ provides various

[2] Another example program will be introduced later on to illustrate how partial applications and name clashes are handled by π-RED$^+$.

```
LETREC
  f = λuv.LETREC
            g = λwz.IF  GT u w  THEN  g ( − 1 u ) z
                                ELSE  f v ( + 1 w )
          IN  g v u
IN  f 1 2

                        ⇓

    LETREC
      g = λwz.IF  GT 1 w  THEN  g ( − 1 1 ) z
                          ELSE LETREC
                                 f = ...
                               IN  f 2 ( + 1 w )
    IN  g 2 1

                        ⇓

    LETREC
      g = λwz.IF  GT 1 w  THEN  g ( − 1 1 ) z
                          ELSE LETREC
                                 f = ...
                               IN  f 2 ( + 1 w )
    IN  g 0 1

                        ⇓

LETREC
  f = λuv.LETREC
            g = λwz.IF  GT u w  THEN  g ( − 1 u ) z
                                ELSE  f v ( + 1 w )
          IN  g v u
IN  f 2 1
```

Fig. 1. Stepwise program execution in π-RED$^+$

means to abbreviate or suppress program parts which are either not of interest or do not belong to the actual focus of control (Kluge, 1994). We forgo discussing these display techniques as they are outside the scope of this paper.

3 The Program Execution Phases of π-RED$^+$

π-RED$^+$ performs high-level program transformations as described in the preceding section by

- compiling MINIRED programs to code of an abstract stack machine ASM which constructs and reduces graph representations of program terms;
- de-compiling the graphs returned by the abstract machine after some pre-specified number of reduction steps (or after having evaluated all redices) into high-level programs.

Compiled graph reduction is subject to two constraints which call for a supercombinator-based parameter-passing mechanism. On the one hand, it derives its efficiency in large parts from substituting formal function parameters (λ-bound variables) by graph pointers rather than by the full argument terms they represent. As these substitutions are in fact naive, they must be kept free of name clashes. On the other hand, compiled function code must be supplied with full sets of arguments to execute correctly. Both constraints can only be satisfied by systematically closing all λ-abstractions and by treating all partial applications as irreducible, i.e., by turning them into closures.

It takes only a fairly simple extension of a supercombinator-based reduction machine to make it appear as a fully-fledged λ-calculus machine. All there is to be done is to add an interpreter stage which, whenever necessary, η-extends to full applications left-over partial supercombinator applications wrapped up in closures, and returns them to the machine for another sequence of code-controlled reductions [3]. The rules for η-extensions in fact coincide with those given in the preceding section for reducing weak normal forms of λ-abstractions to normal forms: η-extensions are called for if and only if programs reduce to (closures made up from) partial supercombinator applications or to (closures made up from) other irreducible applications which include argument terms in weak normal form. These rules ensure that partial applications are η-extended if and only if everything else is done, i.e., no other applications are left to reduce in the particular state of program execution. They also ensure that the resulting new abstractions, in the program term or subterm in which reductions are actually taking place, do never occur in function positions of applications, i.e., they are never applied to anything and need therefore not be compiled, during the actual shift of reductions, to executable machine code. The overhead inflicted by η-extensions is thus kept at the minimum of what is absolutely essential to compute normal forms.

Sequences of code-controlled reductions followed by η-extensions may have to be repeated several times until program terms are reduced to normal forms (provided they exist). Name clashes among η-extended variables can be resolved by a simple enumeration scheme which avoids the complexity of full β-reductions and maintains the original variable names.

Conversion to supercombinators or alternatively the allocation of closures for open functions instantiated in given contexts (environments) are standard techniques of closing λ-terms for compiled graph reduction (Hughes, 1982; Johnsson, 1984; Johnson, 1985; Hudak and Goldberg, 1985; Peyton Jones, 1987; Peyton Jones, 1992). However, both approaches inflict some degree of redundancy, though much of it can be eliminated by subsequent compiler optimizations. Supercombinators repeatedly copy the same instantiations of what originally were free variables into function calls. Closures must be formed individually for all functions defined in local contexts even if they share the same (sub)sets of free variables, i.e., the same variable instantiations may have to be copied several times.

π-RED⁺ uses a less rigorous concept of closing λ-terms which avoids this redundancy altogether. Free variables are abstracted (lifted) out of the larger contexts of LETREC-terms which define sets of mutually recursive functions. Exceptions are

[3] As a matter of convenience, we assume here that partial supercombinator applications include as special cases 0-ary applications, i.e., supercombinators as they are.

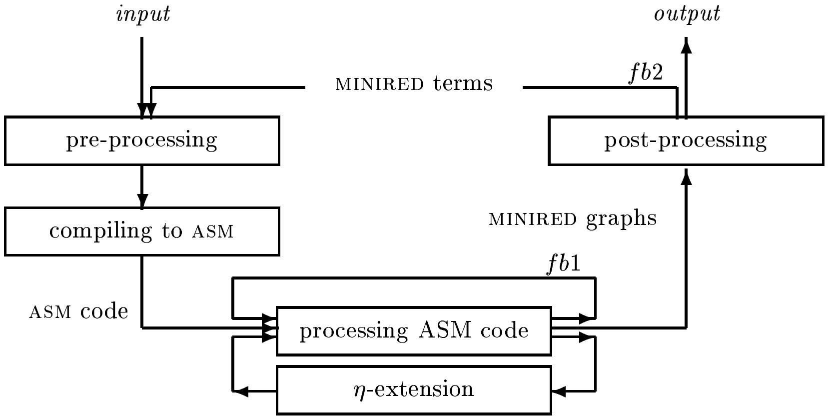

Fig. 2. The program execution cycle of π-RED$^+$

anonymous functions (λ-abstractions) which may occur anywhere within a program term and must therefore be closed individually. Lifted variables are in either case distinguished by special tags from those that are λ-bound in (or local to) the respective (sets of) functions.

These tags relate to two different run-time stacks which the underlying abstract machine ASM employs to accommodate instantiations of either kind of variables. Frames for variables that are locally bound in individual functions are, upon each function call, created on (and eventually released from) one of the stacks, whereas frames for the lifted variables are, in the other stack, set up only once when entering closed contexts (and released upon leaving them), but are never copied when (recursively) calling functions that are locally defined in these contexts.

To accept from and return to the user interface programs in high-level notation, and to run compiled code in between, program execution in π-RED$^+$ is a four phase process, as depicted in fig. 2 : pre-processing MINIRED terms, compiling them to ASM code, processing the compiled code, and post-processing the resulting graphs to obtain new MINIRED terms as output. The ASM processor may repeatedly call for η-extensions to set up partial applications for further reductions.

The pre-processor prepares MINIRED programs for compilation to ASM code by first λ-lifting free variables out of LETREC-terms and out of anonymous abstractions to form closed contexts.

If e denotes the term to be closed, and $\{ w_1, \ldots , w_q \}$ denotes the set of free variables in e, then λ-lifting turns e into:

$$(\sim \tilde{\lambda}\, w_1 \ \ldots\ w_q.e\ w_1 \ \ldots\ w_q\ ) \ .$$

The tilde annotations $\sim$ distinguish the applications and abstractions introduced due to λ-lifting from those specified in the original MINIRED program. They will henceforth be referred to as tilde-applications and tilde-abstractions, respectively.

It is important to note that tilde annotations do not belong to the syntax in which MINIRED programs are written, and consequently must also not occur in programs displayed at the user interface of π-RED$^+$after shifts of reductions. They can only be introduced by the pre-processor as specified above, and left-over annotations must be removed by the post-processor before returning programs as output. λ-lifting implies that tilde-abstractions can only occur in function positions of tilde applications, and that no other terms can occur in these positions.

This property is essential for the correct usage of two different stacks for variable instantiations. It enables the ASM compiler to generate code which is guaranteed to access argument terms to be substituted for occurrences of $\tilde{\lambda}$-bound variables on the one hand and for occurrences of λ-bound variables on the other hand from the same stacks into which they are being pushed upon reducing the respective applications.

When applying λ-lifting to the example program introduced in section 2, we get

$$
\begin{array}{l}
\text{LETREC} \\
\quad f \ = \ \lambda uv.(\sim \tilde{\lambda}\, uv.\text{LETREC} \\
\qquad\qquad\qquad\qquad g = \lambda wz.\text{IF GT } u\ w \text{ THEN } g\ (\ -\ 1\ u\ )\ z \text{ ELSE } f\ v\ (\ +\ 1\ w\ ) \\
\qquad\qquad\qquad\quad \text{IN } g\ v\ u \\
\qquad\qquad\qquad\qquad\qquad\qquad u\ v\ ) \\
\text{IN } f\ 1\ 2 \quad .
\end{array}
$$

This program includes an obvious opportunity for optimization which can be directly taken care of by the pre-processor. Since the inner LETREC in this particular case forms the entire body term of the function f, and f is defined at top level, the formal parameters of f are the same as those that need to be lifted. This does nothing but introduce an additional parameter passing step with no action in between. It can be immediately eliminated, yielding:

$$
\begin{array}{l}
\text{LETREC} \\
\quad f \ = \tilde{\lambda}\, uv.\text{LETREC} \\
\qquad\qquad\quad g = \lambda wz.\text{IF GT } u\ w \text{ THEN } g\ (\ -\ 1\ u\ )\ z \text{ ELSE } (\sim f\ v\ (\ +\ 1\ w\ )) \\
\qquad\qquad \text{IN } g\ v\ u \\
\text{IN } (\sim f\ 1\ 2) \quad .
\end{array}
$$

Since f has thus changed from a λ-abstraction to a $\tilde{\lambda}$-abstraction, applications of f must be changed to tilde-applications as well. This is to remain consistent with the constraints on the syntactical positions in which tilde-abstractions may legitimately occur to ensure correct ASM code generation. Note that this optimization is possible if and only if, as in this particular case, all occurrences of f are in function positions of full applications. However, if f occurs in function position of a partial application or is passed along as a function parameter, the non-optimized version of f must be used since it cannot generally be decided statically whether and where f will be applied to a full set of argument terms.

Another pre-processing step converts all λ- and $\tilde{\lambda}$-bound variable occurrences into differently tagged reversed de Bruijn indices (or de Bruijn levels) (deBruijn, 1972). This conversion is defined on λ-abstractions as

$$\lambda x_1 \lambda x_2 \ \ldots \ \lambda x_n.e \Longrightarrow \Lambda_{x_1} \Lambda_{x_2} \ \ldots \ \Lambda_{x_n}.e^{\#}$$

where $\Lambda_{x_1}\Lambda_{x_2}\ \ldots\ \Lambda_{x_n}$ denotes a sequence of n nameless binders [4], and $e^{\#}$ derives from e by replacing, for all $i \in \{1\ \ldots\ n\}$, free occurrences of x_i by the index $\#(i-1)$, i.e., the variable bound by the outermost λ receives the lowest index, and the variable bound by the innermost λ receives the highest index. We will refer to indices prefixed (or tagged) with a # as A-indices for short.

An equivalent conversion applies to $\tilde{\lambda}$-abstractions. It replaces binders $\tilde{\lambda}\ x_i$ by nameless binders $\tilde{\Lambda}_{x_i}$ and occurrences of x_i by indices prefixed (tagged) as $\sim (i-1)$, to which we will refer as T-indices. These conversions simplify the subsequent compilation to ASM code insofar as the indices directly translate into offsets relative to the respective stack frame bases.

When applying these conversions to our example program, we get:

$$
\begin{array}{l}
\text{LETREC} \\
\quad f\ = \tilde{\Lambda}_u\tilde{\Lambda}_v\ .\text{LETREC} \\
\qquad\qquad\qquad g = \Lambda_w\Lambda_z.\text{IF GT} \sim\!0\ \#0\ \text{THEN}\ g\ (\ -\ 1 \sim\!0\ )\ \#1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ (^{\sim}\ f \sim\!1\ (\ +\ 1\ \#0\ )) \\
\qquad\qquad\quad \text{IN}\ g \sim\!1 \sim\!0 \\
\text{IN}\ (^{\sim}\ f\ 1\ 2)
\end{array}
$$

All occurrences of the $\tilde{\lambda}$-bound (lifted) variables u and v are now replaced by the T-indices $\sim\!0$ and $\sim\!1$, and all occurences of the λ-bound variables w and z are now replaced by the A-indices #0 and #1, respectively.

Suffice it to say that these pre-processing steps can actually be performed in one pass through the original MINIRED program.

The ASM compiler translates pre-processed MINIRED terms into abstract machine code. This code controls the graph reductions performed by the ASM processor. It always reduces the actual top-level application of a program term (or of the subterm that constitutes the actual focus of control). Top-level partial applications are η-extended to full applications for further code-controlled reductions.

The post-processor transforms the graphs obtained from the processor into equivalent high-level MINIRED programs, with all function definitions still referenced and all variable names restored as in the original program, and with all left-over tilde-applications undone.

When reducing programs step by step and without modifying intermediate programs or changing the focus of control, further shifts of reductions may be resumed by saving and directly restoring the respective intermediate machine states, as is indicated by the arrow labeled $fb1$ in fig.2. Otherwise the programs must pass through a full execution cycle, as depicted by the path labeled $fb2$.

[4] The variable names attached to the $\tilde{\lambda}$-binders as subscripts are to enable the compiler to prepare a graph node from which the post-processor can re-construct the original function in the output program where needed.

4 The Abstract Stack Machine

The abstract stack machine ASM, of which an earlier version based on supercombinator reduction is described in (Gaertner and Kimms and Kluge, 1992), defines an intermediate level of code generation. It allows for a simple compilation scheme which produces fairly efficient code and facilitates the re-conversion of intermediate states of program execution back into high-level programs. An ASM implementation on a particular hardware platform, either by interpretation or by compilation to host machine code, need not necessarily correspond one-to-one to its abstract structure.

The ASM is defined by a quadruple $(C, (A, W, T, R), H, r)$ whose components denote (from left to right)

- the code C to be executed;
- a system of four run-time stacks A, W, T, R;
- a heap H which accommodates graph structures;
- a counter value r which specifies an upper bound on the number of reduction steps to be performed on the current program.

Using four stacks rather than just one allows for some rather straightforward high-level optimizations, including some house-keeping operations which minimize the demand for stack and heap space. It also liberates the ASM compiler from calculating addresses relative to changing stack tops which may have to be re-done when generating target machine code.

The stacks serve the following purposes [5] :

- stack W accommodates the workspace frames for function calls in which temporaries are held and the argument frames for further function calls are set up;
- stack A holds the argument frames for instantiations of local function parameters (occurrences of A-indices) which result from reducing full applications of λ-abstractions;
- stack T holds the frames for instantiations of non-local parameters (occurrences of T-indices) which result from the reduction of tilde-applications;
- stack R holds the return addresses of function calls, of branches to conditionals, and of tilde-applications; it is also used to set up argument frames for tilde-applications;

The argument stack A and the workspace stack W are interchanged upon each function call. Thus, the arguments set up by the calling function on its workspace stack become directly available on the argument stack of the called function. The function value is always computed on the actual workspace stack. When returning from a function call, (the pointer to) this value is moved from stack W to A, whereupon another stack switch restores the original stack configuration. The value

[5] In the sequel, the term function generally refers to user-defined functions (or λ-abstractions), not to primitive functions, conditionals etc..

computed by the called function thus ends up on top of the workspace stack of the calling function. Stack switches are also performed between the stacks T and R when reducing tilde-applications, i.e., when calling and returning from closed contexts.

The important point here is that the complete environment for executing a piece of code is thus available in what are actually the topmost stack frames. Accesses to the frames on stacks A and T can be specified as fixed displacements relative to the frame bases, which directly derive from the A- and T-indices generated by the pre-processor; accesses to stacks W and R affect at most as many of the topmost consecutive entries as are actually in the frames.

Switching the stacks also enables the ASM compiler to generate code which releases as early as possible, and right from the top of either stack A or T, (pointers to) argument graphs that are no longer needed, thus minimizing the consumption of stack and heap space.

Of course, the same ends can also be achieved with one stack on which all frames build up on top of each other. However, this solution generally requires a compilation scheme which keeps track of dynamically changing offsets relative to actual stack-top positions. Moreover, releasing arguments as soon as possible then entails re-arranging the stack by costly `MOVE` instructions: argument entries usually have to be pulled out from underneath workspace entries and return addresses, and the ensuing gaps must be closed unless stack space is to be wasted (Johnsson, 1984; Peyton Jones, 1987).

The very basic instructions of ASM are the following:

`PUSH_W c` pushes the item `c` (which may be a constant value, a primitive function symbol or a graph pointer) into the workspace stack W.

`PUSH_AW i` reads the i-th entry relative to the top of the argument stack A and pushes it into the workspace stack W.

`PUSH_TW i` reads the i-th entry relative to the top of stack T and pushes it into the workspace stack W.

Variants of these instructions are `PUSH_R c`, `PUSH_AR i`, `PUSH_TR i`, which use the return stack `R` rather than the workspace stack `W` as a destination.

`MOVE_WR` moves the topmost entry of stack `W` to the top of stack `R`.

`AP n` is the most complex instruction of the machine. It attempts to apply the top element of the workspace stack W to the n elements underneath. If the top element is a primitive function or a pointer to function code whose arity is less than or equal to n, then the application is actually reduced; otherwise a closure is constructed in the heap. In either case, the function and at most as many arguments as are required by it are eventually popped off the stack, and (a pointer to) the result is pushed instead. If the function consumes less than n arguments, then another attempt is made to apply the resulting term to the remaining arguments. Reducible applications of user-defined functions (λ-abstractions), before branching to the respective codes, flip the stacks A and W and push the actual instruction counter values as return addresses into stack R.

`BRA_T p_f` branches to the code of a tilde-abstraction the arguments for which are set up on the return stack R. Upon entry into the code at label `p_f`, the stacks R and T are flipped, and the actual instruction counter value is pushed into the new stack R. Since tilde-applications, by definition, are full applications, the code always finds the correct number of arguments on what after the stack switch has become stack T.

`BRA_C p_c` branches to code for a conditional and pushes the actual instruction counter value into stack R.

`JFALSE p_m` is generated as the first instruction of a conditional. It continues code execution at label `p_m` if the top element of the workspace stack is `FALSE`, and at the next instruction in sequence if it is `TRUE`. In either case, the top element is subsequently popped. If the top element is not a boolean value (which may happen with untyped languages), then a closure including this element and the pointer `p_m` is constructed in the heap.

`FREE_A n` pops n elements off the argument stack A.

`FREE_T n` pops n elements off the tilde stack T.

`RTF` is to return, by means of the address held on top of stack R, from a function call. Before doing so, the instruction moves the result from stack `W` to stack `A`, pops the return address off stack `R`, and subsequently flips both stacks to restore the situation as before the function call.

`RTT` is to return from the reduction of a tilde-application. It works as `RTF`, except that it flips the stacks T and R and that nothing is moved between stacks.

`RTC` is to return from a conditional, again by means of an address held on top of stack R, but without switching any stacks.

`EXIT` terminates the program execution.

5 The Basic Compilation Scheme

When compiling MINIRED programs to ASM code, information must be preserved about the nesting of function definitions and about formal parameter names (λ-bound variables). This information may be required by the post-processor to re-construct high-level programs from the graphs that emerge from partial or complete code execution. To include this information into the code, the ASM-compiler generates for each LETREC-construct of the form LETREC $\ldots\ f_i\ =\ \Lambda_{u_1} \ldots \Lambda_{u_r}.e_i\ \ldots$ IN e_0 a graph structure as shown in fig. 3. It is composed of a LETREC-descriptor node which for each function f_i contains a pointer `p_f`$_{\mathtt{i}}$ to a function descriptor node. This descriptor, in turn, includes a pointer each to the code that reduces the instantiated function body e_i, to a list of the formal parameter names of f_i, and back to the LETREC-descriptor node. The parameter list is preceded by two integer values r and s which respectively specify the number of local and non-local (lifted) parameters of f_i. The code for f_i may be referenced from somewhere else by the pointer `p_f`$_{\mathtt{i}}$. Anonymous λ-abstractions compile to just the lower part of this structure, with a nil pointer as the last entry of the function descriptor.

The basic compilation scheme for MINIRED programs may be defined by means

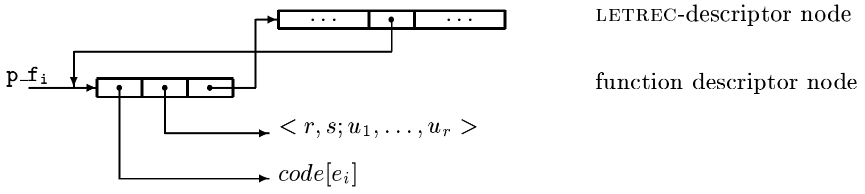

Fig. 3. The compiled graph for a function defined under a LETREC

of a function $\mathcal{C}$ as follows:

$$\mathcal{C}[e : es, (m,n)] \;=> \; code[e] \; ; \; \mathcal{C}[es, (m,n)] \; ; \quad .$$

It partitions the actual compiler input $e : es$ into some syntactically complete head term e, followed by a tail es which represents the remainder of the program to be compiled. The tail may be empty, denoted as ϵ [6]. The parameters m and n respectively denote the sizes of the frames on stacks A and T which make up the environment in which $e : es$ must be reduced. The symbol ";" catenates two pieces of code, with $code[e]$ denoting the code for the head term e.

ASM code uses only branch instructions, e.g. to function code or to code for (the components of) conditionals, which are complemented by return instructions. Code referenced by branch labels from within the code for e can thus be placed into the free space immediately following the code for es (and possibly other pieces of code that have already been placed there).

A typical example is the compilation of a LETREC-construct embedded in other program parts. It splits up into first compiling its body term, then compiling the remainder of the program text that follows, and finally compiling the function definitions. While compiling the LETREC body and the body terms of the individual functions, occurrences of LETREC-bound identifiers f_i are replaced by symbolic labels `p_f_i`. They remain undefined until the respective functions are compiled as well.

To abstract from unnecessary details, we specify the compilation of a function definition $f_i \;=\; \Lambda_{u_1} \ldots \Lambda_{u_r}.\, e_i$ simply as

$$\texttt{p\_f\_i} \;\longrightarrow\; code[e_i] \quad .$$

It defines the symbolic label `p_f_i` as a pointer to the heap location into which the code $code[e_i]$ is being placed. However, this notation implies that the compilation

[6] Splitting the compiler input recursively into head and tail merely reflects the serialization of the entire compilation process.

(1) $\mathcal{C}[c : es, (m,n)]$ => `PUSH_W c` ; $\mathcal{C}[es, (m,n)]$;
(2) $\mathcal{C}[\#i : es, (m,n)]$ => `PUSH_AW i`; $\mathcal{C}[es, (m,n)]$;
(3) $\mathcal{C}[\sim i : es, (m,n)]$ => `PUSH_TW i`; $\mathcal{C}[es, (m,n)]$;
(4) $\mathcal{C}[\text{IF } e_0 \text{ THEN } e_1 \text{ ELSE } e_2 : es, (m,n)]$ =>
$\mathcal{C}[e_0, (m,n)]$; `BRA_C p_c` ; $\mathcal{C}[es, (m,n)]$;
`p_c` $\rightarrow$ `JFALSE p_m` ; $\mathcal{C}[e_1, (m,n)]$; `RTC` ; `p_m` $\rightarrow \mathcal{C}[e_2, (m,n)]$; `RTC` ;
(5) $\mathcal{C}[(\ e_0 \cdots e_{(r-1)}\ e_r\ ) : es, (m,n)]$ =>
$\mathcal{C}[e_r, (m,n)]$; $\mathcal{C}[e_{(r-1)} : \cdots : e_0 : ap^r : es, (m,n)]$;
(6) $\mathcal{C}[ap^r : es, (m,n)]$ => `AP r` ; $\mathcal{C}[es, (m,n)]$;
(7) $\mathcal{C}[\Lambda_{u_1} \ldots \Lambda_{u_r}.e : es, (m,n)]$ => `PUSH_W p_f` ; $\mathcal{C}[es, (m,n)]$;
`p_f` $\rightarrow \mathcal{C}[e, (r,0)]$; `FREE_A r` ; `RTF` ;
(8) $\mathcal{C}[\text{LETREC } f_1 = e_1 \cdots f_r = e_r \text{ IN } e_0 : es, (m,n)]$ =>
$\mathcal{C}[e_0, (m,n)]$; $\mathcal{C}[es, (m,n)]$;
`p_f₁` $\rightarrow \mathcal{F}[e_1, (0,0)]$; $\cdots$; `p_fᵣ` $\rightarrow \mathcal{F}[e_r, (0,0)]$;
(9) $\mathcal{F}[\Lambda_{u_1} \ldots \Lambda_{u_r}.e, (0,s)]$ => $\mathcal{C}[e, (r,s)]$; `FREE_A r` ; `RTF` ;
(10) $\mathcal{F}[\tilde{\Lambda}_{v_1} \ldots \tilde{\Lambda}_{v_s} .\text{LETREC } f_1 = e_1 \cdots f_r = e_r \text{ IN } e_0 : es, (m,n)]$ =>
$\mathcal{C}[e_0, (0,s)]$; `FREE_T s` ; `RTT` ;
`p_f₁` $\rightarrow \mathcal{F}[e_1, (0,s)]$; $\cdots$; `p_fᵣ` $\rightarrow \mathcal{F}[e_r, (0,s)]$;
(11) $\mathcal{C}[(^\sim\ e_0 \cdots e_{(r-1)}\ e_r\ ) : es, (m,n)]$ =>
$\mathcal{S}[e_r, (m,n)]; \mathcal{S}[e_{(r-1)} : \cdots : e_0 : tap^r : es, (m,n)]$;
(12) $\mathcal{S}[c : es, (m,n)]$ => `PUSH_R c` ; $\mathcal{S}[es, (m,n)]$;
(13) $\mathcal{S}[\#i : es, (m,n)]$ => `PUSH_AR i`; $\mathcal{S}[es, (m,n)]$;
(14) $\mathcal{S}[\sim i : es, (m,n)]$ => `PUSH_TR i`; $\mathcal{S}[es, (m,n)]$;
(15) $\mathcal{S}[\tilde{\Lambda}_{v_1} \ldots \tilde{\Lambda}_{v_s}\ \Lambda_{u_1} \ldots \Lambda_{u_r}.e : tap^s : es, (m,n)]$ =>
`BRA_T p_f` ; $\mathcal{C}[es, (m,n)]$;
`p_f` $\rightarrow \mathcal{C}[e, (r,s)]$; `FREE_A r` ; `FREE_T s` ; `RTF` ;
(16) $\mathcal{S}[\tilde{\Lambda}_{v_1} \ldots \tilde{\Lambda}_{v_s}\ \text{LETREC } f_1 = e_1 \cdots f_r = e_r \text{ IN } e_0 : tap^s : es, (m,n)]$ =>
`BRA_T p_f` ; $\mathcal{C}[es, (m,n)]$;
`p_f` $\rightarrow \mathcal{C}[e_0, (0,s)]$; `FREE_T s` ; `RTT` ;
`p_f₁` $\rightarrow \mathcal{F}[e_1, (0,s)]$; $\cdots$; `p_fᵣ` $\rightarrow \mathcal{F}[e_r, (0,s)]$;
(17) $\mathcal{S}[f : tap^s : es, (m,n)]$ => `BRA_T p_f` ; $\mathcal{C}[es, (m,n)]$;
(18) $\mathcal{S}[e : es, (m,n)]$ => $\mathcal{C}[e, (m,n)]$; `MOVE_WR` ; $\mathcal{S}[es, (m,n)]$;
(19) $\mathcal{C}[\epsilon, (m,n)]$ => ϵ

Fig. 4. The ASM compilation rules

includes the creation of a function descriptor, as depicted in fig. 3, to which `p_fᵢ` is pointing, and through which the function code can only be accessed indirectly.

The complete compiler function $\mathcal{C}$, which employs two subfunctions $\mathcal{F}$ and $\mathcal{S}$ to compile abstractions and tilde-applications, respectively, is specified in fig. 4.

The following explanations may help to understand the various compilation rules and the codes they generate:

- A constant value or a primitive function symbol is directly pushed into the workspace stack W. For every occurrence of a LETREC-bound variable f_i a symbolic label `p_fᵢ` is pushed ($\mathcal{C}$-rule 1).
- An occurrence of an A-index $\#i$ requires that an argument (pointer) be copied from the position `i` relative to the top of stack A and be pushed into stack W ($\mathcal{C}$-rule 2); likewise, an occurrence of a T-index $\sim i$ requires that an argument

be copied from the position `i` relative to the top of stack T and, again, be pushed into stack W ($\mathcal{C}$-rule 3).

- A conditional compiles to code which first computes the predicate term e_0, and then branches, via the label `p_c`, to code for the component terms e_1 and e_2 which are compiled separately into the heap locations referenced by `p_c` ($\mathcal{C}$-rule 4).
- An application is compiled to code which first reduces the argument terms $e_r, \cdots, e_1$ in this sequence and then the term e_0 in function position. This code pushes $r + 1$ entries into stack W. The instruction `AP r` that follows attempts to interpret the topmost entry as a function and to apply it to the r entries underneath ($\mathcal{C}$-rules 5 and 6).
- An anonymous λ-abstraction without λ-lifted variables compiles to code which pushes a label `p_f` to a heap location into which the code for the body term is subsequently placed ($\mathcal{C}$-rule 7).
- A LETREC-construct compiles to code for its body term e_0, followed by the code for the rest of the program. The codes for the individual function definitions are compiled separately into the heap locations labeled $\texttt{p\_f}_1, \ldots, \texttt{p\_f}_\texttt{r}$. If no variables need to be lifted, the compilation of the function definitions by the rule $\mathcal{F}$ takes place with the parameters r and s set to zero since no accesses need to be performed on stack T, and the frame sizes r are determined in the course of compiling the functions by means of the $\mathcal{F}$-rule ($\mathcal{C}$-rule 8).
- A function defined under a LETREC is compiled to code for its body term, with the parameter r stepped up from zero to its arity as it needs to access an argument frame of that size on stack A, with the parameter s specifying the size of the frame that may have to be accessed on stack T ($\mathcal{F}$-rule 9). If, due to the optimization described in section 3, such a function is turned into a tilde-abstraction (whose body term is another LETREC), then $\mathcal{F}$-rule 10 applies.
- A tilde-application requires a slightly different compilation scheme $\mathcal{S}$ as its components need to be pushed into the return stack R rather than into the workspace stack W. Moreover, since a tilde-application, by definition, is always a full application, $\mathcal{S}$ generates a branch instruction `BRA_T p_f` which transfers control directly to the code of the respective tilde-abstraction. Anonymous λ-abstractions and LETRECs transformed into tilde-abstractions compile to codes which differ from those generated by the respective $\mathcal{C}$-rules in that the instructions `FREE_T s`; `RTT` are added at the end ($\mathcal{S}$-rules 11 to 17).
- The $\mathcal{S}$-rule 18 covers all cases to which none of the $\mathcal{S}$-rules 11 to 17 applies.
- An empty MINIRED term (denoted as ϵ) trivially compiles to empty code ($\mathcal{C}$-rule 19).

Yet another compilation rule generates an entry label `p_e` for and an `EXIT` instruction following the topmost level of program code (which usually is the code for the body term of some outermost LETREC construct).

6 Code Optimizations

When adding a few special instructions, the ASM codes generated by the compilation scheme $\mathcal{C}$ may be improved by some rather straightforward optimizations. However, in order to maintain the one-to-one correspondence between machine-level graph reductions and high-level program transformations that is necessary to support stepwise program execution in π-RED$^+$, the opportunities for code optimizations are limited.

Prime targets for optimizations are instruction sequences of the form

$$\ldots \texttt{PUSH\_W c ; AP n ;} \ldots \quad .$$

Whenever `c` is a primitive function symbol `sf` of arity k or a pointer `p_f` to an abstraction of arity k, and $k = n$, the following replacements may be made:

- ... `PUSH_W sf; AP n ;` ... $\Longrightarrow$ ... `DELTAsf ;` ...
- ... `PUSH_W p_f; AP n ;` ... $\Longrightarrow$ ... `BRA_F p_f ;` ...

These dedicated instructions execute directly, without first pushing a function symbol or a code pointer into stack W:

- `DELTAsf` stands for one of the primitive instruction symbols `ADD*`, `MULT*`,..., `GT*`, etc. [7]. It is directly applied to the appropriate number of entries in stack W, pops them, and pushes the result instead;
- `BRA_F p_f` flips the stacks A and W, pushes the instruction counter into stack R, and then branches to the code referenced by `p_f`;

Another important target for code optimizations are occurrences of the instructions `FREE_A n` and `FREE_T n`. They may be moved ahead of all instructions that neither change the stack configuration nor access the stacks A and T, respectively.

Good examples in kind are tail-recursive functions as in:

$$\text{LETREC } \texttt{f} = \Lambda_{u_1} \ldots \Lambda_{u_n} . \, g \; a_1 \; \ldots \; a_m \;\; , \ldots , \texttt{g} \; = \; \Lambda_{v_1} \ldots \Lambda_{v_m} . \, f \; b_1 \; \ldots \; b_n \text{ IN } e_o \quad .$$

By application of this optimization they would compile to:

$$\texttt{p\_f} \;\rightarrow\; \ldots \; \texttt{FREE\_A n; BRA\_F p\_g ; RTF ;} \quad \texttt{p\_g} \;\rightarrow\; \ldots \; \texttt{FREE\_A m; BRA\_F p\_f ; RTF ;}$$

with the `FREE_A` instructions possibly being moved even further to the left. This code executes in constant stack space as argument frames are being released before the subsequent tail calls are executed.

Tail recursions allow for yet another optimization which replaces tail calls by tail jumps, yielding the instruction sequences

$$\texttt{p\_f} \;\rightarrow\; \ldots \; \texttt{FREE\_A n; JTAIL\_A p\_g} \;\; ; \;\; \texttt{p\_g} \;\rightarrow\; \ldots \; \texttt{FREE\_A m; JTAIL\_A p\_f} \;\; ; \, .$$

The instruction `JTAIL_A p_f` switches the stacks A and W and then jumps directly back to the code for the function f, without storing a return address on stack R (and likewise for tail jumps to the code for g).

[7] The * following the instruction symbols are to indicate that these instructions are different from those that must be used in conjunction with `AP` instructions.

```
p_e  →  PUSH_R 2; PUSH_R 1; BRA_T p_f; EXIT

p_f  →  PUSH_TW 0; PUSH_TW 1; BRA_F p_g; FREE_T 2; RTT;

p_g  →  PUSH_AW 0; PUSH_TW 0; GT*;
        JFALSE p_m; PUSH_AW 1; FREE_A 2;
        PUSH_TW 0; PUSH_W 1; MINUS*; JTAIL_A p_g;

p_m  →  PUSH_AW 0; FREE_A 2; PUSH_W 1; PLUS*;
        MOVE_WR; PUSH_TR 1; BRA_T p_f; RTF;
```

Fig. 5. Optimized ASM code for the program shown in fig.1

Tail jumps cause a small problem though since the stack switches effected by these instructions are not complemented by equivalent numbers of `RTF` instructions. To restore the correct stack configuration upon returning from a sequence of tail jumps, each return address stacked up on R must include a single tail flag which is flipped upon each tail jump, starting with the tail flag set to zero when pushing the address by a `BRA_F p_f` instruction. When set to one upon executing the complementary `RTF` instruction, the stacks remain as they are, otherwise they must be flipped to restore the original constellation [8].

Whenever a conditional makes up the entire body term of an abstraction, as in

$$h \;=\; \Lambda_{u_1} \ldots \Lambda_{u_m}.\ \text{IF } e_0 \text{ THEN } e_1 \text{ ELSE } e_2 \quad ,$$

the code can be flattened to:

$$\mathcal{C}[e_0,(m,n)]\ ;\texttt{JFALSE p\_q}\ ;\mathcal{C}[e_1,(m,n)]\ ;\texttt{FREE\_A m}\ ;\texttt{RTF};$$
$$\texttt{p\_q} \;\rightarrow\; \mathcal{C}[e_2,(m,n)]\ ;\ \texttt{FREE\_A m}\ ;\texttt{RTF}; \quad ,$$

saving the branch instruction and the complementary return instruction. This rule can be recursively applied to nested conditionals.

Compilation of the pre-processed version of our example program as given in section 3 produces the ASM-code shown in fig.5, which forms a cyclic graph. The first line to which `p_e` is pointing computes the body term of the entire program, which is the application ($\sim f\ 1\ 2$). Its third instruction branches, via the pointer `p_f`, to the code of the function f which is shown in the second line. This code, in turn, branches, via the pointer `p_g`, to the code for the function g which follows in the third line. It first computes the predicate of the conditional and, depending upon its value, either branches, via the pointer `p_m`, to the code for the ELSE-term or continues with to the code for the THEN-term.

[8] We forgo optimizing in a similar way tail recursions that involve closed contexts (or tilde-abstractions) since replacing `FREE_T n ; BRA_T p_c ; RTT` with `FREE_T n ; JTAIL_T p_c` causes problems with the flag bit. `JTAIL_T` would have to flip the stacks `T` and `R`, as a consequence of which the return address into which the flag bit is included would be temporarily buried under T-frames and thus could not be easily accessed.

7 Code Execution

Code execution starts with empty ASM stacks and with nothing but the initial program graph in the heap. The orderly termination of the code upon executing the `EXIT`-instruction leaves the stacks A, T, R empty and a single entry representing the normal form of the program on the workspace stack W. This entry is either a pointer to a coherent program graph or a basic value.

Intermediate states of code execution generally have all stacks filled to the extend to which closed contexts and function calls therein are actually nested. Some of the entries on stacks A, T, W may be basic values, others may be pointers to graph fragments set up in the heap, say closures, which as yet need not necessarily form a coherent structure, i.e., they are not fully contained in each other. Entries on stack R are primarily return addresses, but also (pointers to) arguments of tilde-applications.

If the reduction counter decrements to zero upon arriving at some intermediate state of code execution, no more applications must be reduced. Instead, the ASM must assemble, from the actual stack contents and from the structures referenced in the heap, a coherent graph, which the post-processor may subsequently convert into high-level output. This can be accomplished in an orderly way by means of the code not yet executed. It specifies, in the form of `RTx` and `FREE_x` instructions, the complete return path to the terminal state, in which a pointer to the resulting graph is left on stack W and all other stacks are cleared. All there is to be done when executing the remaining code is to treat all function applications encountered along this path as irreducible and, following standard procedure in such cases, to convert them into closures.

The most general of the instructions which must create closures upon exhaustion of the reduction counter is `AP r`. It first inspects the topmost entry on stack W. If it is a pointer `p_f` to a function (λ-abstraction), then it retrieves from the function descriptor the frame parameters r and s to create a closure of the general form

$$\texttt{p\_clos} \ \rightarrow \ (\sim \underbrace{\tilde{\Lambda} \ldots \tilde{\Lambda}}_{s}.(\ \texttt{p\_f}\ a_1 \ldots a_r\ )\ b_1 \ldots\ b_s\ ) \quad ,$$

where `p_f` and $a_1, \ldots a_r$ are the topmost $r+1$ entries of stack W, which must be popped, and $b_1, \ldots, b_s$ are the topmost s entries of stack T, which must be copied without destruction.

If the topmost entry on W is some item `h` other than a pointer to function code, e.g. a primitive function symbol or a constant, then the closure simply takes the form

$$\texttt{p\_clos} \ \rightarrow \ (\ \texttt{h}\ a_1 \ldots a_r) \ ,$$

where `h` and $a_1, \ldots, a_r$ are the entries that need to be cleared off stack W.

The instruction `BRA_F p_f` creates closures of the same form as `AP r`, except that only r items must be popped off stack W.

The instruction `BRA_T p_f` inspects the descriptor of the closed context referenced by `p_f` for the value of the parameter s, and creates a closure

$$\texttt{p\_clos} \rightarrow (\, \texttt{p\_f}\, b_1 \, \ldots \, b_s \,) \quad ,$$

where $b_1, \ldots, b_s$ are the topmost s entries on stack R, which must be popped.

The pointers `p_clos` must in all cases be pushed into stack W after the components of the respective closures have been popped. As with all other graph structures set up in the heap, the construction of closures includes the construction of descriptors which are directly referenced by the pointers `p_clos` and, in turn, contain references to the closures themselves. As before, these descriptors have been omitted in the above specifications.

All other ASM instructions must be executed as defined in section 4 to push stack entries (which subsequently become components of closures), to return from closed contexts and from function calls, and to remove the respective stack frames.

ASM code execution is illustrated in fig.6. It shows a sequence of machine states as it develops when running the code given in fig.5. Since this program does not terminate by itself, termination is enforced by initializing the reduction counter with the value 4, which sets a breakpoint after as many function calls. Upon completing these calls, the machine is left in some intermediate state from where it must execute the remaining code as just described.

The first column of fig. 6 enumerates the sequence of instructions shown in the second column. The next four columns depict memory segments $S1$, $S2$, $S3$ and $S4$ about which the ASM stacks A, W, T and R are permuted, as is indicated by the respective annotations. The rows depict stack permutations and stack entries after having executed the instructions in the second column.

Code execution sets out with the instruction sequence that computes the outermost function application ($\sim f$ 1 2), and continues with two calls of the function g, followed by another call of f.

The instructions that are of interest in this sequence are those in steps `3`, `6`, `16`, `27`. They realize the function calls and thus effect stack switches, push return addresses into stack R (with the exception of the `JTAIL_A` instruction in step 16), and also decrement the reduction counter r [9].

Once this counter is down to zero, which in the particular example happens in step 27, the ASM completes the remaining code as described above. The creation of closures for left-over applications is exemplified by the instruction `BRA_F p_g` of step 30. It inspects the descriptor referenced by the pointer `p_g` to determine the number of local and non-local parameters of the function g, and accordingly removes the topmost two entries from stack W (the argument values 2 and 1), copies the topmost two entries of stack T (the values 1 and 2), and creates from

[9] The return addresses are given as references to the instructions immediately following the respective call instructions, and are denoted as `&EX`^{0} (as abbreviation for `&EXIT`), `&FT`^{0} (as abbreviation for `&FREE_T`), etc., with the superscripts denoting the flag bits that distinguish even and odd numbers of stack switches.

STEP	INSTR	S1	S2	S3	S4	r
1	PUSH_R 2	W:	A:	T:	R: 2	4
2	PUSH_R 1	W:	A:	T:	R: 2 1	4
3	BRA_T p_f	W:	A:	R: &EX^0	T: 2 1	3
4	PUSH_TW 0	W: 1	A:	R: &EX^0	T: 2 1	3
5	PUSH_TW 1	W: 1 2	A:	R: &EX^0	T: 2 1	3
6	BRA_F p_g	A: 1 2	W:	R: &EX^0 &FT^0	T: 2 1	2
7	PUSH_AW 0	A: 1 2	W: 2	R: &EX^0 &FT^0	T: 2 1	2
8	PUSH_TW 0	A: 1 2	W: 2 1	R: &EX^0 &FT^0	T: 2 1	2
9	GT*	A: 1 2	W: TRUE	R: &EX^0 &FT^0	T: 2 1	2
10	JFALSE p_m	A: 1 2	W:	R: &EX^0 &FT^0	T: 2 1	2
11	PUSH_AW 1	A: 1 2	W: 1	R: &EX^0 &FT^0	T: 2 1	2
12	FREE_A 2	A:	W: 1	R: &EX^0 &FT^0	T: 2 1	2
13	PUSH_TW 0	A:	W: 1 1	R: &EX^0 &FT^0	T: 2 1	2
14	PUSH_W 1	A:	W: 1 1 1	R: &EX^0 &FT^0	T: 2 1	2
15	MINUS*	A:	W: 1 0	R: &EX^0 &FT^0	T: 2 1	2
16	JTAIL_A p_g	W:	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
17	PUSH_AW 0	W: 0	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
18	PUSH_TW 0	W: 0 1	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
19	GT*	W: FALSE	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
20	JFALSE p_m	W:	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
21	PUSH_AW 0	W: 0	A: 1 0	R: &EX^0 &FT^1	T: 2 1	1
22	FREE_A 2	W: 0	A:	R: &EX^0 &FT^1	T: 2 1	1
23	PUSH_W 1	W: 0 1	A:	R: &EX^0 &FT^1	T: 2 1	1
24	PLUS*	W: 1	A:	R: &EX^0 &FT^1	T: 2 1	1
25	MOVE_WR	W:	A:	R: &EX^0 &FT$^!$ 1	T: 2 1	1
26	PUSH_TR 1	W:	A:	R: &EX^0 &FT^1 1 2	T: 2 1	1
27	BRA_T p_f	W:	A:	T: &EX^0 &FT^1 1 2	R: 2 1 &RFT^0	0
28	PUSH_TW 0	W: 2	A:	T: &EX^0 &FT^1 1 2	R: 2 1 &RTF^0	0
29	PUSH_TW 1	W: 2 1	A:	T: &EX^0 &FT^1 1 2	R: 2 1 &RTF^0	0
30	BRA_F p_g	W: p_clos	A:	T: &EX^0 &FT^1 1 2	R: 2 1 &RTF^0	0
31	FREE_T 2	W: p_clos	A:	T: &EX^0 &FT^1	R: 2 1 &RTF^0	0
32	RTT	W: p_clos	A:	R: &EX^0 &FT^1	T: 2 1	0
33	RTF	W: p_clos	A:	R: &EX^0	T: 2 1	0
34	FREE_T 2	W: p_clos	A:	R: &EX^0	T:	0
35	RTT	W: p_clos	A:	T:	R:	0
36	EXIT	W: p_clos	A:	T:	R:	0

Fig. 6. ASM code execution of the example program

these components a closure

$$\texttt{p\_clos} \ \rightarrow \ (\sim \widetilde{\Lambda}\widetilde{\Lambda} \, . \, (\ \text{p\_g}\ 2\ 1\ )1\ 2\ )\ ,$$

the pointer `p_clos` to which is pushed into stack W.

The remaining instructions simply clear the stacks, complete the function calls performed so far, and restore the initial stack configuration, with the pointer to the closure returned on stack W as the result of the computation. The return instructions RTT in lines 32 and 35 find the flag bits in the topmost addresses on stack R set to zero and thus switch the stacks T and R, whereas the return instruction RTF in line 33 finds its flag bit set to one and therefore leaves the stacks as they are.

The post-processor transforms the above closure into the high-level program

$$
\begin{array}{l}
\text{LETREC} \\
\quad g = \lambda w z.\text{IF}\ \ gt\ 2\ w\ \ \text{THEN}\ \ g\ (\ -\ 1\ 2\ )\ z \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE LETREC} \\
\qquad\qquad\qquad\qquad\qquad\qquad f = \ldots \\
\qquad\qquad\qquad\qquad\qquad\quad \text{IN}\ \ f\ 1\ (\ +\ 1\ w\ ) \\
\text{IN}\ \ g\ 1\ 2
\end{array}
$$

by undoing the tilde-application and by re-constructing, from the descriptor to which `p_g` is pointing the nesting of LETRECs as it defines the function g [10].

8 Reduction to Normal Forms

ASM code reduces only full function applications. Partial applications are treated as irreducible, or as being in weak normal form, and converted into closures. In subsequent reduction steps, these closures may or may not pick up the missing arguments to become full applications and thus reducible. Other closures may originate from applications of primitive functions to type-incompatible arguments or from applications which have terms other than functions in function positions. Both are irreducible and remain so irrespective of the contexts in which they may occur and of reduction steps which may substitute them somewhere else.

Partial applications of λ-abstractions which, in the form of closures, survive code execution could simply be left as they are and, in the post-processed program, show up in high-level notation, possibly marked as being irreducible. As most other compiled graph reducers, the system would then realize a supercombinator-based reduction semantics, i.e., it would compute weak normal forms.

However, it takes only a simple extension of the code execution (processing) phase to implement a fully-fledged λ-calculus under which partial applications of λ-abstractions, and thus the programs in which they occur, can be reduced to normal forms (if they exist). All there is to do conceptually is to η-extend a partial application of the general form

$$(\ \lambda\, x_1 \ldots x_n.\ e\ \ a_1\ \ \ldots\ \ a_k\ )\quad (k < n)$$

to a new $(n-k)$-ary function (λ-abstraction)

$$\lambda x_{(k+1)} \ldots x_n.\ (\ \lambda\, x_1 \ldots x_n.\ e\ \ a_1\ \ \ldots\ \ a_k\ x_{(k+1)}\ \ \ldots\ \ x_n\ )\quad ,$$

in the body of which the partial is turned into a full application, with $x_{(k+1)}, \ldots, x_n$ filling in for the missing argument terms. Reducing this application yields the abstraction $\lambda x_{(k+1)} \ldots x_n.e'$, where e' emerges from e by naive substitution of all free occurrences of $x_1, \ldots, x_k$ by the argument terms $a_1, \ldots, a_k$, respectively, and of all free occurrences of $x_{(k+1)}, \ldots, x_n$ by themselves. Subsequent reduction steps performed in e' may involve more η-extensions until eventually its normal form e^N is reached.

[10] Note that this program is the same as the second one in fig. 1, except that the instantiations of the parameters u and v are interchanged.

π-RED$^+$can be made to accomplish this largely by executing ASM code as it is. A partial application as above compiles to the code

$$code[\ a_k\ ]; \ldots code[\ a_1\ ]; \texttt{PUSH\_W p\_f}; \texttt{AP k} \quad .$$

which first computes and subsequently pushes into stack W (pointers to) the (weak) normal forms $a_1^N, \ldots, a_k^N$ of the arguments and then pushes the function pointer. As depicted in fig. 3, `p_f` points to a descriptor which, in turn, contains a reference each to the list of formal parameters $< n,\ s; x_1, \ldots, x_n >$ of the function and to the function code. The instruction `AP k` accesses this descriptor to inspect the arity n and, failing to match it with its own arity parameter k, removes the components of the application from the stack and creates a closure

$$\texttt{p\_clos} \rightarrow (\ \texttt{p\_f}\ a_1^N \ldots a_k^N\ ) \ .$$

in its place.

Whenever such a closure must be reduced to normal form, it is for further processing turned over to an η-extension unit, as indicated in fig. 2. This unit requires another stack P which keeps track of nested η-extensions. Both the ASM processor and the η-extension unit have access to the ASM stacks and to stack P.

By interpretation of the components of a closure as above, the η-extension unit

- sets up a full argument frame by pushing into the actual workspace stack W the variables $x_n, \ldots, x_{(k+1)}$ retrieved from the formal parameter list hung up under the function descriptor, followed by the argument terms $a_k, \ldots, a_1$, in this order;
- pushes the variables $x_{(k+1)}, \ldots, x_n$ in this order into stack P, from where the binders that need to be inserted into the resulting program are constructed later on (which requires another brief interception of code execution by the η-extension unit);
- returns control back to the ASM processor by executing a `BRA_F p_f` instruction to enter and execute the function code in the environment held in what, after the stack switch effected by this instruction, has become the topmost frame of the argument stack A (and is still present as the topmost frame on stack T).

These η-extensions may be recursively called for to reduce nested partial applications until a term in normal form is reached.

The variables which in a particular η-extension step are pushed into stack P must be popped again upon returning from the respective function call. They are, in the order in which they are popped, used to construct binders in front of the term returned as the normal form (if it exists) of the η-extended function application. The resulting λ-abstraction is subsequently placed into the heap.

Controlling stack P accordingly necessitates another execution mode for `BRA_F` and `RTF` instructions iff reductions are being performed in the body terms of η-extended λ-abstractions, i.e., iff stack P contains at least one set of variables. A branch instructions `BRA_F`, whether executed by the ASM processor or under the control of the η-extension unit, must push a separator symbol $|$ into stack P. Conversely, an `RTF` instruction (which may only be executed by the ASM processor) must pop a separator $|$. If variables are stacked up underneath, the η-extension unit must take over to pop them and to construct binders. Upon arriving at another separator $|$ or at an empty stack, control returns to code execution by the ASM processor.

Pushing and popping separator symbols as described ensures that correct nesting levels of η-extensions are maintained on the stack. Variables pushed by the η-extension unit prior to executing a `BRA_F` instruction are popped again immediately following the complementary `RTF` instruction, i.e., upon completing the computation of the term in front of which binders for these variables need to be constructed.

To avoid name clashes when η–extending partial applications in this naive form, the η-extension unit routinely assigns in ascending order so-called η-levels. All variables added in the same η-extension step receive the same η-level. Thus, while repeatedly cycling through η-extensions and subsequent code execution phases, the system need never engage in full β-reductions to maintain correct binding levels among identically named variable occurrences that are bound in different contexts (scopes).

As a simple example, we consider again a non-terminating program,

$$\text{LETREC } f \;=\; \lambda uv.\,(\,+\,(\;f\;u\;)\;v\;)\text{ IN } f\;v \quad ,$$

which contains two partial applications of the binary function f. After one β-reduction step (which in compliance with the classical definition renames to v^1 the bound variable v that may get involved in a naming conflict), it transforms to

$$\lambda v^1.\,(\,+\,(\text{ LETREC } f \;=\; \lambda uv.(\,+\,(\;f\;u\;)\;v\;)\text{ IN } f\;v\;)\;v^1\;) \quad ,$$

and, after another two steps, to

$$\lambda v^1.(\,+\,\lambda v^2.(\,+\,(\;\lambda v^3.(\,+\,(\text{LETREC } f \;=\; \ldots \text{ IN } f\;v\;)\;v^3\;)\;v^2\;)\;v^1\;) \quad .$$

π-RED$^+$computes this λ-term by executing the compiled ASM code of the original program, which is

```
p_e → PUSH_W v ; PUSH_W p_f ; AP 1 ; EXIT
p_f → PUSH_AW 1 ; PUSH_AW 0 ; FREE_A 2 ; PUSH_W p_f ; AP 1 ; ADD* ; RTF .
```

When initializing the reduction counter with the value $r = 3$, which sets a breakpoint after three function calls, this code produces the sequence of machine states shown in fig. 7. The layout of the figure is essentially the same as in fig. 6, except that the stacks R and P remain permanently assigned to the memory segments $S3$ and $S4$, respectively. Stack T is not shown at all since the particular program does not contain nested function definitions; hence there are no free variables that need to be λ-lifted.

Code execution includes three η-extensions to deal with the partial applications encountered by the instructions `AP 1` in steps 3, 9, and 15. They need to be com-

STEP	INSTR	S1	S2	S3	S4	r
1	PUSH_W v	W: v	A:	R:	P:	3
2	PUSH_W p_f	W: v p_f	A:	R:	P:	3
3	AP 1	W: p_clos	A:	R:	P:	3
	η-extension of the closure p_clos $\to$ (p_f v)					
	new stack configuration :					
		W: v^1 v	A:	R:	P: v^1	2
	branching to the function code:					
4	(BRA_F p_f)	A: v^1 v	W:	R: EX^0	P: v^1|	2
5	PUSH_AW 1	A: v^1 v	W: v^1	R: EX^0	P: v^1|	2
6	PUSH_AW 0	A: v^1 v	W: v^1 v	R: EX^0	P: v^1|	2
7	FREE_A 2	A:	W: v^1 v	R: EX^0	P: v^1|	2
8	PUSH_W p_f	A:	W: v^1 v p_f	R: EX^0	P: v^1|	2
9	AP 1	A:	W: v^1 p_clos	R: EX^0	P: v^1|	2
	η-extension of the closure p_clos $\to$ (p_f v)					
	new stack configuration :					
		A:	W: v^1 v^2 v	R: EX^0	P: v^1| v^2	1
	branching to the function code:					
10	(BRA_F p_f)	W:	A: v^1 v^2 v	R: EX^0 ADD*^{0}	P: v^1| v^2|	1
11	PUSH_AW 1	W: v^2	A: v^1 v^2 v	R: EX^0 ADD*^{0}	P: v^1| v^2|	1
			...			
15	AP 1	W: v^2 p_clos	A: v^1	R: EX^0 ADD*^{0}	P: v^1| v^2|	1
	η-extension of the closure p_clos $\to$ (p_f v)					
	new stack configuration :					
		W: v^2 v^3 v	A: v^1	R: EX^0 ADD*^{0}	P: v^1| v^2| v^3	0
	branching to the function code:					
16	(BRA_F p_f)	A: v^2 v^3 v	W: v^1	R: EX^0 ADD*^{0} ADD*^{0}	P: v^1| v^2| v^3|	0
			...			
20	PUSH_W p_f	A: v^2	W: v^1 v^3 v p_f	R: EX^0 ADD*^{0} ADD*^{0}	P: v^1| v^2| v^3|	0
21	AP 1	A: v^2	W: v^1 v^3 p_c^1	R: EX^0 ADD*^{0} ADD*^{0}	P: v^1| v^2| v^3	0
22	ADD*	A: v^2	W: v^1 p_c^2	R: EX^0 ADD*^{0} ADD*^{0}	P: v^1| v^2| v^3|	0
23	RTF	W: v^2 p_c^2	A: v^1	R: EX^0 ADD*^{0}	P: v^1| v^2| v^3	0
	completing η-extension of step 16:					
		p_c^2 $\to$ (+ p_c^1 v^3) $\Longrightarrow_\eta$ p_c^2 $\to$ λv^3.(+ p_c^1 v^3)				
24	ADD*	W: p_c^3	A: v^1	R: EX^0 ADD*^{0}	P: v^1| v^2|	0
25	RTF	A:	W: v^1 p_c^3	R: EX^0	P: v^1| v^2	0
	completing η-extension of step 10:					
		p_c^3 $\to$ (+ p_c^2 v^2) $\Longrightarrow_\eta$ p_c^3 $\to$ λv^2.(+ p_c^2 v^2)				
26	ADD*	A:	W: p_c^4	R: EX^0	P: v^1|	0
27	RTF	W: p_c^4	A:	R:	P: v^1	0
	completing η-extension of step 4:					
		p_c^4 $\to$ (+ p_c^3 v^1) $\Longrightarrow_\eta$ p_c^4 $\to$ λv^1.(+ p_c^3 v^1)				
28	EXIT	W: p_c^4	A:	R:	P:	0

Fig. 7. ASM code execution with repeated η-extensions

pleted, when returning from the respective function calls, by means of the RTF instructions in steps 27, 25 and 23, respectively.

The following explains the interaction between code execution by the ASM processor and η-extensions, which require interpretation, in more detail.

The `AP 1` instruction, say, of step 3 creates a closure

$$\texttt{p\_clos} \;\rightarrow\; (\,\texttt{p\_f}\; v\,)$$

from the topmost two entries of stack W since the arity of the function referenced by `p_f` exceeds by one the arity of the application. With nothing else left to do, the η-extension unit is called to turn the partial into a full application. The necessary η-extension

$$(\texttt{p\_f}v) \;\Longrightarrow_\eta\; \lambda v^1.(\,\texttt{p\_f}\; v\; v^1\,)$$

is realized by pushing the argument terms v^1 and v of the full application into stack W, and the variable v^1 bound by the added λ into stack P. The η-extension unit then executes a `BRA_F p_f` instruction to branch to the code of the function f, thereby returning control to the ASM processor. This instruction also swaps the stacks A and W and pushes a separator symbol into stack P.

The same η-extensions are carried out in steps 9 and 15, except that the η-levels assigned to the variables v, as they are pushed into the stacks W and P, are routinely incremented as a safeguard against potential name clashes.

Steps 11 to 15 and steps 17 to 21 have been abbreviated since they merely repeat the instructions of steps 5 to 9.

With the reduction counter down to zero in step 17, the applications that are left in the remaining instruction sequence are turned into closures in order to construct a complete graph and to clear the ASM stacks.

The return instructions RTF which occur in this sequence, as usual, swap the stacks A and W and pop return addresses off stack R. Stack P not being empty in either of the machine states, they also pop separation symbols and, before continuing with code execution at the return addresses, call the η-extension unit. It pops the variables found on top of stack P and constructs binders in front of the terms referenced by the pointers that are actually on top of stack W, which are the closures produced by the preceding `ADD*` instructions. Control returns to code execution upon encountering another separator symbol or an empty stack P, which in the particular cases happens after popping just one variable.

The construction of the resulting graph proceeds from innermost to outermost, starting with the instruction `AP 1` in step 21. It creates a closure

$$\texttt{p\_c}^1 \;\rightarrow\; (\,\texttt{p\_f}\; v\,)$$

by removing `p_f` and v from stack W and by pushing the pointer $\texttt{p\_c}^1$ instead. This closure becomes a component of the closure

$$\texttt{p\_c}^2 \;\rightarrow\; (\,+\,\texttt{p\_c}^1\; v^3\,)$$

created by the `ADD*` instruction of step 22.

The RTF instruction of step 23 calls the η-extension unit. It pops the variable

$$
\begin{array}{l}
\texttt{p\_c}^4 \;\rightarrow\; \lambda v^1.(\; + \;\texttt{p\_c}^3\; \texttt{v}^1\;) \\
\quad\quad \texttt{p\_c}^3 \;\rightarrow\; \lambda v^2.(\; + \;\texttt{p\_c}^2\; \texttt{v}^2\;) \\
\quad\quad\quad\quad \texttt{p\_c}^2 \;\rightarrow\; \lambda v^3.(\; + \;\texttt{p\_c}^1\; \texttt{v}^3\;) \\
\quad\quad\quad\quad\quad\quad\quad \texttt{p\_c}^1 \;\rightarrow\; (\;\texttt{p\_f}\;\texttt{v}\;)
\end{array}
$$

Fig. 8. The graph obtained from the code execution as in fig. 7

v^3 off stack P, accesses the closure referenced by the pointer $\texttt{p\_c}^2$ found on top of stack W, and converts it into the abstraction

$$\texttt{p\_c}^2 \;\rightarrow\; \lambda v^3.(\; +\texttt{p\_c}^1\; v^3\;)$$

which again is placed, as a piece of the resulting graph, into the heap. The pointer $\texttt{p\_c}^2$ remains the same since the representation of graph structures in π-RED$^+$ merely requires changing descriptor entries when converting these closures into abstractions. The remaining code produces in the same way another two nesting levels of abstractions and returns the resulting graph as depicted in fig. 8.

The post-processor constructs from this graph the term

$$\lambda v^1.(\; + \;\lambda v^2.(\; + \;\lambda v^3.(\; + \;(\;\texttt{p\_f}\; v\;)\; v^3\;)\; v^2\;)\; v^1\;) \quad ,$$

In a last step, it de-references the graph pointer `p_f` to re-construct the full LETREC-construct in place of the innermost application.

9 The Performance of π-RED$^+$

With respect to an efficient implementation of the ASM on a host system we took the pragmatic approach of compiling ASM code to C. It renders π-RED$^+$ portable and also takes advantage of the target code optimization techniques available in the C compilers of the hosts.

A first version of this compiler has just been completed. It includes a type inference system which, given the argument types of the outermost function applications, generates for most programs fully typed code [11], thus closing the efficiency gap between untyped and typed languages. However, as of now, some efficiency is lost again by compiling ASM function codes to individual C functions in order to maintain the one-to-one correspondence between code execution and high-level program transformations that is necessary to support the stepwise execution mode of π-RED$^+$. Moreover, the C code also includes instructions to update reference counts and to release as early as possible heap space that is no longer needed. For some applications, early garbage collection may create a considerable overhead, but it keeps the total demand for heap space nearly minimal.

In this section we will present some comparative performance figures for π-RED implementations, particularly of ASM code interpretation versus compilation of ASM code to C. To put these figures into perspective, we also compare them with

[11] Programs that cannot be fully typed do the type-checking at runtime.

func calls total	ghc comp	hbc comp	Clean comp	SML comp	ASM comp	ASM interpr	λ-term interpr
1.5 Mio	0.3	1.6	0.2	0.4	1.8	18.0	87.5
3.0 Mio	0.6	3.2	0.4	0.8	3.5	35.8	175.4
4.5 Mio	0.9	4.9	0.6	1.2	5.3	53.8	264.7

Fig. 9. Run-time comparison between three implementations of π-RED , two HASKELL implementations, CLEAN and SML based on the example program (all times are given in seconds).

other implementations of functional systems. They include the two HASKELL compilers `ghc-0.26` of Glasgow University and `hbc-0.999.7` of Chalmers University, the Nijmegen CLEAN compiler `0.84`, and the compiler version `0.93` for Standard ML of New Jersey. HASKELL and CLEAN are based on a lazy evaluation regime, whereas SML and π-RED$^+$ use a strict regime.

Depending on particularities of the application programs, the run-time performance of ASM code interpretation is improved by factors ranging from 2 to 8 over high-level interpretation of λ-terms, as implemented in π-RED* (Schmittgen and Bloedorn and Kluge, 1992), with both interpreters written in C. Compilation of ASM code to C enhances the performance by another factor of 3 on average, and by a factor of about 10 at the maximum.

To substantiate these claims, we consider as a first benchmark program a modified version of the example program introduced in section 2, which looks like this:

```
LETREC
  f = λnuv.IF LE 1 n THEN + u v
                     ELSE LETREC
                              g = λwz.IF GT v w
                                      THEN g ( - 1 w ) z
                                      ELSE f ( - 1 n ) u v
                          IN g u v
IN f 150000  9 1
```

Since none of the other system implementations can deal with non-terminating recursions other than by time-out conditions, the equivalent to counting reduction steps had to be explicitely specified as part of the program itself by adding a monotonically decrementing parameter n to the function f. The program also allows to specify varying ratios between calls of f and g by appropriate choice of argument values for the parameters u and v of the function f (In the particular example this ratio has been chosen as $1 : 9$, which in combination with a parameter value of $n = 150\,000$ amounts to a total of 1 500 000 function calls before termination.).

Fig. 9 shows the performance figures for all seven implementations, with the total number of function calls stepped up from 1.5 Mio to 3.0 Mio to 4.5 Mio.

For this particular program which primarily tests parameter passing, and specifically the handling of non-local variables, there are ratios of about $5 : 1$ between full interpretation of λ-terms and ASM code interpretation, and of about $10 : 1$ be-

program	ghc comp	hbc comp	Clean comp	SML comp	ASM comp	ASM interpr
Takeuchi 24 16 8	0.7	4.1	0.8	11.5	3.5	33.2
Ackermann 3 7	0.2	1.1	0.2	0.8	1.1	11.0
qusort 4000	93.4	51.4	33.7	14.9	14.7	17.2
qusortbin 4000	0.3	0.6	0.2	0.5	1.5	3.17
queens 10	4.2	6.5	3.4	8.4	8.8	27.0
relax 32*32 500	37.6	* 238.9	-	13.6	2.9	3.9
fractal 9	18.4	31.0	20.0	38.4	61.5	178.7
detlist 8*8	1.6	2.0	-	1.38	6.5	13.6
detarray 8*8	42.3	53.2	-	10.9	4.8	9.5

Fig. 10. Run-time comparison of π-RED, HASKELL, CLEAN and SML implementations based on selected example programs (* the run-time of `hbc` code for the `relax` program, due to heap space overflow, is based on only 200 relaxation steps).

tween ASM code interpretation and compilation of ASM code to C. Compiled ASM code is only marginally slower than code produced by the `hbc-0.999.7` HASKELL compiler. The codes produced by the `ghc-0.26` HASKELL, CLEAN and SML compilers are faster by factors ranging from 4 to 9. This rather significant difference is due to the fact that these compilers convert the two nested tail-recursive functions into iteration loops which presumably execute completely within the same set of registers, whereas compiled ASM code performs expensive function calls.

Comparative performance measurements with a variety of other small benchmark programs look decidedly more favorable for compiled ASM code, and even for ASM code interpretation (see figure 10). These programs include the `Takeuchi` function and the `Ackermann` function which again stress the efficiency of recursive function calls and of parameter passing. The remaining six programs deal with sorting problems and with simple numerical applications. There are two versions of a quicksort program, of which `qusort 4000` sorts a list of 4000 integer values by means of a straightforward divide-and-conquer algorithm, and `qusortbin 4000` uses Augustsson's algorithm of the HASKELL program library which does completely without catenating sorted lists. The `queens 10` program solves the ten queens problem. Among the numerical applications are `fractals 9` which computes fractals with a recursion depth of 9, a relaxation program `relax` for PDEs which does 500 iterations on a $32 * 32$ array of real numbers, and two programs which compute the determinant of an $8 * 8$ matrix, of which `detlist` uses nested lists to represent the argument matrix, and `detarray` uses the array representations available in the respective languages [12].

With respect to the `Takeuchi` and `Ackermann` programs, which produce recursive function calls while computing very little in between, there is again a performance

[12] Note that some of the language features used in these programs, such as pattern matching and arrays, are not available in MINIRED but in the functional language KIR supported by π-RED$^+$ (Kluge, 1994).

gain of a factor 10 between ASM code interpretation and compilation to C. The performance gaps relative to the Glasgow HASKELL and CLEAN implementations have become marginally smaller, and to the Chalmers HASKELL and SML implementations they have completely disappeared or even been reversed.

For all other programs there is only a factor of 3 or less left between ASM code interpretation and ASM code compiled to C. This is due to the very efficient implementation of list and array operations in the ASM interpreter, which in more or less the same form are generated by the compiler as well. Relative to compiled ASM code, there is quite some variation among the performance figures of the other systems. For instance, the HASKELL and SML implementations lose considerable ground with respect to programs operating on arrays (`relax`, `detarray`) (CLEAN does not support arrays at all). HASKELL and CLEAN do poorly on the `qusort 4000` program but do decidely better on the `qusortbin 4000` and on the `fractals` programs. Compiled SML code is about on equal terms with compiled ASM code for `qusort 4000`, `queens 10`, and for `Ackermann`, is decidedly slower for all array programs and for `Takeuchi`, and decidedly better for the `qusortbin` and `detlist` programs. Except for the array programs and `qusort 4000`, not even SML shows a run-time advantage of strict over lazy evaluation, presumably due to successful strictness analysis by the lazy systems.

Representative for the consumption of memory space by compiled ASM code are the following figures:

The `queens 10` program needs to allocate 60 kBytes of heap space, including roughly 13 kBytes for descriptors, and 2 kBytes of stack space. The `qusort 4000` program takes 90 kBytes of heap space (again including descriptors) and just 100 Bytes of stack space. In comparison, the Glasgow HASKELL implementation uses 400 kBytes of heap space and 7kByte of stack space for the `queens` program, and 450 kBytes heap space and 200kBytes stack space for the `quicksort` program, i.e., it requires roughly an order of magnitude more space.

10 Conclusion

The machinery described in this paper is to support with reasonable efficiency the reduction semantics of a fully-fledged λ-calculus. Functions and variables are truly treated as first class objects. New functions (λ-abstractions) may be computed from (partial) applications of defined functions, and programs may contain free variables as naming conflicts can be correctly resolved.

Program execution is conceptually realized as a process of meaning-preserving program transformations. The user may set breakpoints by specifying an upper limit on the number of reduction steps to be performed in one shift. The intermediate program reached at the breakpoint is returned to the user interface for inspection. All programs may be reduced to normal forms of λ-terms eventually, if they exist. This may be done either in several shifts of reductions, between which the program may also be modified and the focus of control may be moved to selected subterms in which reductions are to be performed next, or in just one shift, provided the number of reduction steps is chosen large enough. If intermediate programs remain unchanged, the resulting program is guaranteed to be invariant against execution orders since stepwise execution and production runs use the same code on the same machinery.

Run-time efficiency is achieved by compiled graph reduction based on an abstract stack machine ASM as intermediate level of code generation. While executing code, the machine reduces only full function applications. Upon encountering partial function applications at top level, it switches to an interpreter mode which η-extends them to full applications in order to enable the machine to continue with further code-controlled reductions. This cycle may be repeated until a normal form is reached. Potential naming conflicts involving η-extended variables are avoided by a simple enumeration scheme which by means of unique indices distinguishes variables introduced in different η-extension steps.

Though not (yet) implemented in π-RED$^+$, breakpoints for debugging and validation purposes may be introduced by means other than just limiting reduction steps, say, by specifying a particular (sequence of) function call(s) after which the machine is to be halted. It is also possible to display just the immediate context of the function call rather than the complete intermediate program reached at the breakpoint (which may have grown to considerable size). A minor problem arises insofar as the modifications required to implement these features primarily concern pre- and post-processing, but to some lesser extent also compilation and code execution, i.e., the code used for debugging would then not exactly be the same as the one used for production runs.

Acknowledgements

We are indebted to two anonymous referees, to Thomas Johnsson and to Simon Peyton Jones for their valuable comments on the paper. We also like to thank Carsten Rathsack and Claus Reinke for doing the performance measurements.

References

Backus, J. (1978) Can Programming be Liberated from the von Neumann Style ? A Functional Style and its Algebra of Programs, *CACM*, Vol. 21,No. 4, pp. 613-643

Berkling, K. J. (1974) Reduction Languages for Reduction Machines, *Proc. 2nd Int. Symp. on Comp. Arch.*, ACM/IEEE 75CH0916-7C, pp. 133-140

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

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

Church, A. (1941) The Calculi of λ-Conversion, *Princeton University Press*

Cousineau, G., Curien, P.-L., Mauny, M. (1987) The Categorial Abstract Machine, *Science of Computer Programming*, No. 8, pp.173-202

Curry, H.B. (1936) Functionality in Combinatory Logic, *Proc. of the Nat. Academy of Science of the United States of America*, Vol. 20, pp. 584–590

Fairbairn, J.; Wray, S.C. (1988) TIM : A Simple Lazy Abstract Machine to Execute Supercombinators, *Proc. Conf. on Functional Programming and Computer Architecture*, LNCS, No. 274. Springer, pp. 34–45

Gaertner, G., Kimms, A., Kluge, W. (1992) π-RED$^+$- A Compiling Graph Reduction System for a Full-Fledged λ-Calculus, *Proc. 4th Int. Workshop on the Parallel Implementation of Functional Languages*, Aachener Informatik Berichte Nr. 92-19

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

Johnsson, T. (1984) Efficient Compilation of Lazy Evaluation, *SIGPLAN Compiler Construction Conference*, Montreal Queb., 1984

Johnson, T. (1985) Lambda Lifting: Transforming Programs to Recursive Equations, *Proc. Conf. on Functional Programming and Computer Architecture*, LNCS, No. 201, Springer, pp. 190–203

Harper, E., MacQueen,D., Milner, R. (1986) Standard ML, *Laboratory for Foundations of Computer Science*, U of Edinburg

Hindley, J.R.; Seldin, J.P. (1986) Introduction to Combinators and λ-Calculus, Cambridge University Press, *London Mathematical Society Student Texts*

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel,J., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J. (1992) Report on the Programming Language Haskell, *ACM SigPlan Notices*, Vol. 27, No.5,, pp. 17 – 42

Hudak, P., Goldberg, B. (1985) Serial Combinators: Optimal Grains for Parallelism, *Proc. Conf. on Functional Programming and Computer Architecture*, LNCS, No. 201, Springer, pp. 382–399

Hughes, R.J.M. (1982) Super-Combinators - A New Implementation Technique for Applicative Languages, *Proc. ACM Conf. on LISP and Functional Programming*, Pittsburgh, PA, pp. 1 – 19

Kluge, W. (1994) Programming the Reduction System π-RED, *Internal Report* Nr. 9419, Institut f"ur Informatik und Praktische Mathematik, CAU Kiel/Germany

Peyton Jones, S.L. (1987) The Implementation of Functional Programming Languages, Prentice Hall, Englewood Cliffs, NJ,

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

Plasmeijer, R., van Eekelen, M. (1993) Functional Programming and Parallel Graph Rewriting, Addison Wesley

Schmittgen, C., Bloedorn, H., Kluge, W. (1992) π-RED* – A Graph Reducer for a Full-Fledged λ-Calculus, *New Generation Computing*, Vol. 10, No. 2, pp. 173 – 195

Turner, D.A. (1976) A New Implementation Technique for Applicative Languages, *Software Practice and Experience*, Vol. 9, No. 1, pp. 31-49

Turner, D.A. (1986) An Overview of Miranda, *SIGPLAN Notices*, Vol. 21, No. 12, pp. 158 – 166

Allegro CL User Guide (1992), Vo. 1, Version 4.1, Franz Inc.

Sun Common LISP 4.0 User's Guide (1990), SUN Microsystems Inc., 1990