$\mathcal{L}i$SA— a Lazy Interpreter for a Full-Fledged λ-Calculus

C. Rathsack and S.B. Scholz
Dept. of Computer Science
University of Kiel
Germany
email: {car,sbs}@informatik.uni-kiel.dbp.de

September 18, 1992

Abstract

$\mathcal{L}i$SA is the lazy evaluation counterpart of the applicative-order graph reducer π-RED* developed at the university of Kiel. It truly realizes the reduction semantics of an applied λ-calculus, using full-fledged β-reduction as the major reduction rule. Thus we have full support for higher-order functions, self-applications and for interactively controlled stepwise program execution.

Internally, $\mathcal{L}i$SA uses essentially the same graph-structures as π-RED* except for the fact that nested functions containing relatively free variables are not converted into supercombinators.

A well-known problem in terms of efficiency is the instantiation of recursive functions. Systems like the G-machine or SKI-combinator reducers accomplish this either by creating a new graph or by copying a graph-template and, while doing this, immediately substituting actual for formal parameters from the current context (runtime stack-frame). The approach taken with $\mathcal{L}i$SA is to delay these substitutions until it is absolutely necessary to do so and to avoid copying or creating new graph instances altogether. In order to achieve this, it is indispensable to use sophisticated environments with argument frames chained up in the order in which the functions are statically nested in the original program. Easy access (via offsets) to actual parameters during the processing phase can be accomplished by converting the program-graph's variables into environment indices. Hence instances of user-defined functions are created by pairing a pointer to the function-graph with a pointer to the actual environment, whereas the substitution is postponed until the demand for the argument actually arises in the course of performing δ-reductions.

A competitive performance test with π-RED* shows that average program runtimes of $\mathcal{L}i$SA for a representative set of examples is less than a factor of 2 higher than that of π-RED*. This figure is a remarkable improvement over a factor of about three by which lazy evaluation is usually slower than eager evaluation under otherwise identical conditions.

1 Introduction

Lazy evaluators for functional programming languages are usually based on compiled graph reduction techniques, of which the G-machine [John84, John86, PJ87] approach has become more or less a standard. A program composed of nested functions definitions is first converted into a flat set of supercombinators which is then compiled to pieces of G-machine code [John85]. This code constructs and transforms in one conceptual step the graph of the outermost goal expression into a so-called canonical form, and in several subsequent steps into its normal form.

Compilation to efficiently executable code requires that all legitimate programs are well-typed [Miln78] and thus can run to completion without type checks at run time. However this precludes many useful higher-order functions, including self-applications and the computation of new functions from function applications, in particular from partial applications.

Rather than exploiting the full potential of the λ-calculus [Bare84, Hind86] which is often claimed to be the conceptual basis of the functional paradigm, compiled graph reduction is thus essentially confined to computing with utmost speed just (sequences of) basic values. Moreover the particular compilation rules out other than top level reductions and also globally free variables which in our view appear to be indispensible for simple symbolic computations. Interestingly enough, another popular concept — SKI-combinator reduction [Turn79] — suffers, for different reasons though, from the same limitations.

In order to avoid these deficiencies, we have concentrated our research on reduction systems which support the reduction semantics of an untyped applied λ-calculus, with a full-fledged β-reduction as the major reduction rule. An outgrow of the research is an applicative-order graph reducer π-RED [Berk76, Klug86] which exists both as a high-level interpreter π-RED* [Schm92] and as a compiled version π-RED$^+$ [Gaer91]. Both systems allow for interactively controlled reductions. On the one hand, reductions can be performed in a stepwise manner, and intermediate programs can be returned to the user for inspection in high-level notation. On the other hand, due to the referential transparency, reductions can be performed in any part of the program without creating side effects in other parts.

In this paper we give an outline of the interpreting graph reducer $\mathcal{L}i$SA (**L**azy **I**nteractive **S**imulator for an **A**pplicative Language) which is the lazy evaluation counterpart of π-RED*. Both interpreters support the same high-level language KiR (**K**$i$el **R**eduction Language) [Bloe89], use essentially the same graph representations, and have bound variables transformed into uniquely enumerated binding levels (similar to DeBruijn indices [DeBr72]) to facilitate accesses into the runtime environment.

Program execution in both systems partitions into three phases:

- a pre-processor transforms high-level programs into an internal graph representation;
- a processor performs a user-specified number of reductions on the graph, or on a user selected subgraph;
- a post-processor re-constructs from the graph returned by the processor the equivalent high-level program representation.

In the sequel we will primarily concentrate on the implementation of the $\mathcal{L}i$SA runtime environment for full fledged β-reductions. Section 2 will give an outline of the basic concept, which in section 3 is followed by a formal definition of the graph reduction mechanisms involved. In section 4 we try to compare the relative performances of $\mathcal{L}i$SA and π-RED* based on small example programs whose correct execution depends on the availability of full β-reductions.

2 Function Instantiation in $\mathcal{L}i$SA

There are basically two ways of dealing with function applications in graph reduction systems.

On the one hand, we have systems such as the G-machine [Augu87, John84, John86, PJ87] which execute code to construct and subsequently reduce the graphs of instantiated function bodies. The graphs build up only to the extent absolutely necessary to compute canonical (or normal) forms. While constructing them, pointers to argument graphs are directly inserted into the correct syntactical positions. The argument pointers are taken from a run-time stack. In order to avoid naming conflicts and to simplify accesses into the stack, the code performs supercombinator reductions, in which case environments can be represented in the form of flat stack-frames.

On the other hand, there are systems which set out with complete graph representations of the entire program, of which the subgraphs for defined functions are used as templates. Free occurrences of bound variables are either abstracted out (say by conversion to SKI-combinator terms) or replaced by indices [DeBr72] which specify offset positions relative to the top of a run-time stack. Whenever a function application is to be reduced, the function template is copied to the extent necessary and, while doing this, argument pointers are fetched from the stack and inserted into the respective syntactical positions. This concept is used in SKI combinator reduction machines [Turn79], and also in both versions of π-RED [Gaer91, Schm92].

In $\mathcal{L}i$SA we carry this approach a little further in that we take full advantage of lazy evaluation. Rather than copying function templates and directly inserting into these copies argument pointers, function instances are represented by means of so-called indirection nodes which contain a pointer each to the template graph and to the actual environment in which the graph is to be reduced. However, graph templates are never actually copied nor are arguments actually substituted. The computations are more or less directly performed in the environment. Following the head-order reduction concept proposed by Berkling [Berk86], $\mathcal{L}i$SA also works with open functions, i.e., with λ-abstractions that may contain free variables which may or may not be bound in a larger context. Thus an environment generally consists of several argument pointer frames for instantiations of formal by actual function parameters which are chained up in the same order in which function definitions are recursively nested in the original program. Whenever a function application is reduced, a new argument pointer frame is added to the actual environment.

We briefly illustrate this concept by means of two simple examples. The first one is to show how indirection nodes hook up environments to graph templates, and how the environment expands when performing β-reductions. The second one illustrates the replacement of λ-bound variable occurrences in nested function definitions by index tuples which directly identify the environment entries under which the respective argument terms can be found.

Consider first the application $(\lambda u.\lambda v.(u\ v) e_1\, e_2)$ which is assumed to be a subterm embedded in a larger context. Figure 1 depicts three phases of reducing the graph of this term in the environment E which initially is linked, via an indirection node, to the topmost application node (figure $1a$). While traversing the spine of this graph, the environment is linked to both argument terms and to the abstraction in its head position (figure $1b$). Upon reducing this application the actual environment frame is extended by the instantiations

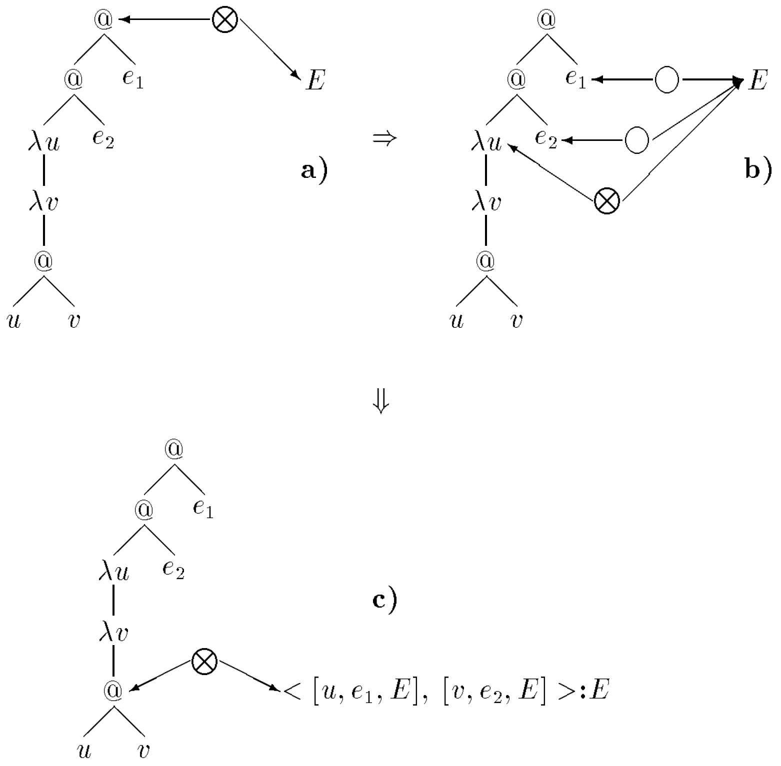

Figure 1: Function instantiation in $\mathcal{L}i$SA

for u and v, and the new environment is passed on to the abstraction body (figure 1c). In this figure the indirection nodes marked with a cross constitute the actual focus of control, $[x, e, E]$ denotes the value of the term e in the environment E which is to be substituted for free occurrences of x. The brackets $<$ and $>$ denote the delimiters of a frame and $:$ denotes the concatenation of a frame to the current environment. Note that the terms e_1 and e_2 are not actually evaluated and that none of the substitutions are actually carried out.

Consider next a small program (see figure 2a) composed of two nested function definitions, where **K** denotes the K-combinator $\lambda x.\lambda y.x$. The internal representation of this program in $\mathcal{L}i$SA has all the λ-binders replaced by nameless binders Λ and, all occurrences of λ-bound variables replaced by index tuples of the general form $(i, j)^\lambda$ according to the following rule:

- The index $i \geq 0$ denotes the number of intervening nesting levels of function definitions (or letrec's) between a variable occurrence and its binding Λ.
- The index $j \geq 0$ counts the number of Λ's that are between the binding Λ and the next inner letrec on the path to the occurrence of $(i, j)^\lambda$.

This leads to an internal representation for the example program which basically looks like figure 2b. The structure of the run-time environment in $\mathcal{L}i$SA is designed for to provide easy access to argument terms, using the index tuples $(i, j)^\lambda$ as follows:

$$
\begin{array}{l}
\textbf{letrec} \\
\quad f = \lambda u.\lambda v.\textbf{letrec} \\
\qquad\qquad\qquad g = \lambda w.((u\ v)\ w) \\
\qquad\qquad\quad \textbf{in } (g\ f) \\
\textbf{in } ((f\ f)\ \mathbf{K})
\end{array}
\qquad \textbf{a)}
$$

$\Downarrow$ variable conversion

$$
\boxed{
\begin{array}{ll}
\textbf{letrec} & \\
\quad f = \Lambda.\Lambda.\ \textbf{letrec} & \\
\qquad\qquad\qquad g = \Lambda.\ (((1,1)^{\lambda}\ (1,0)^{\lambda})\ (0,0)^{\lambda}) & env_0 = nil \\
\qquad\qquad \textbf{in } (g\ f) & \\
\textbf{in } ((f\ f)\ \mathbf{K}) & \textbf{b)}
\end{array}
}
$$

$\Downarrow$ reduction

$$
\boxed{
\begin{array}{ll}
\textbf{letrec} & \\
\quad g = \Lambda.\ (((1,1)^{\lambda}\ (1,0)^{\lambda})\ (0,0)^{\lambda}) & \\
\textbf{in } (g\ \textbf{letrec} & env_1 = \langle [\mathbf{K}, nil], [f, nil] \rangle \rightarrow nil \\
\qquad\quad f = \ldots & \\
\qquad \textbf{in } f) & \textbf{c)}
\end{array}
}
$$

$\Downarrow$ reduction

$$
\boxed{
(((1,1)^{\lambda}\ (1,0)^{\lambda})\ (0,0)^{\lambda}) \qquad env_2 = \langle [f, nil] \rangle \rightarrow env_1 \qquad \textbf{d)}
}
$$

$\Downarrow$ reduction

$$
\boxed{
\begin{array}{ll}
\textbf{letrec} & \\
\quad g = \Lambda.\ (((1,1)^{\lambda}\ (1,0)^{\lambda})\ (0,0)^{\lambda}) & \\
\textbf{in } (g\ \textbf{letrec} & env_3 = \langle [(0,0)^{\lambda}, env_2], [(1,0)^{\lambda}, env_2] \rangle \rightarrow nil \\
\qquad\quad f = \ldots & \\
\qquad \textbf{in } f) & \textbf{e)}
\end{array}
}
$$

$\Downarrow$ reduction

$$
\boxed{
(((1,1)^{\lambda}\ (1,0)^{\lambda})\ (0,0)^{\lambda}) \qquad env_4 = \langle [f, nil] \rangle \rightarrow env_3 \qquad \textbf{f)}
}
$$

Figure 2: Environment evolution in $\mathcal{L}i$SA

- i designates the number of link pointers that must be followed into the environment in order to reach the frame that must be accessed, and

- j locates the position of the argument term within that frame.

This requires that the environment is built up as follows:

- whenever a recursive function is called, a new frame containing the actual parameters is created and chained up to the environment which contains the instantiations of the (relatively) free variables;

- whenever a λ-abstraction is applied, the actual frame is copied and extended by the new actual parameters.

This may be illustrated by means of the environment structures that develop when reducing the example program of figure 2. Since only the index positions of the environment entries are required to perform variable lookups, the environment entries can be simply depicted as tupels $[body, environment]$. Environment-frames are again represented as $< ... >$, and $\rightarrow$ represents a link between two frames that are chained up.

Setting out with an empty environment env_0 (figure 2b), a first frame is added when calling the function f, which yields the environment env_1. Note that the second entry of env_1 contains a pointer to the (recursive) function definition of f (figure 2c). Since f calls its local function g next, we get another frame chained up to env_1, resulting in the environment env_2 (figure 2d), in which the body of g is to be reduced. When evaluating the body of g, the argument term substituted for the index tuple $(1,1)^\lambda$ must be looked up first (figure 2d). This yields

$$lookup((1,1)^\lambda, env_2) = lookup((1,1)^\lambda, \langle [f, nil] \rangle \rightarrow \langle [\mathbf{K}, nil], [f, nil] \rangle \rightarrow nil) = [f, nil] \ ,$$

i.e., the function f instantiated with the empty environment. Since $[f, nil]$ in turn is applied to $(1,0)^\lambda$ and $(0,0)^\lambda$, a new environment-frame is created and chained to nil, thus invoking a new traversal of the body of f in the environment env_3 (figure 2e). By the same mechanism as just described, we get the situation shown in figure 2f. This situation first leads to

$$lookup((1,1)^\lambda, env_4) = lookup((1,0)^\lambda, env_2) = [\mathbf{K}, nil]$$

and subsequently to

$$lookup((1,0)^\lambda, env_4) = lookup((0,0)^\lambda, env_2) = [f, nil]$$

resulting in $[f, nil]$.

The index tuples we have used here in fact specify binding levels for variable occurrences which can be formally defined as follows. Let

$$\dashv ...\Lambda_0... \underbrace{\Lambda...\Lambda}_{l} ... \underbrace{letrec...letrec}_{k} ...(i,j)^\lambda... \underbrace{in...in}_{k} ... \vdash$$

be a term, then with respect to Λ_0 the occurrence of $(i,j)^\lambda$ is called

- *block-protected* if $i > k$,
- *protected* if $i = k \quad \wedge \quad j > l$,
- *free* if $i = k \quad \wedge \quad j = l$,
- *bound* if $i = k \quad \wedge \quad j < l$,
- *block-bound* else $(i < k)$.

With these definitions at hand, we can define the β-reduction of an application $(\Lambda.body\, arg)$ as follows. An occurrence of $(i,j)^\lambda$ in *body* must

- decrement j by one if it is protected against Λ;
- be substituted by *arg* if it is free;
- be left as it is else;

An occurrence of $(i,j)^\lambda$ in *arg* must

- be left as it is if it is bound or block-bound;
- increase i and j by the respective numbers of letrecs and Λ's in the scopes of which *arg* is substituted.

Dynamic modifications of the index tupels as prescribed by these definitions are expensive to implement and to execute. Hence they should be avoided at run-time, i.e. during the processing phase. The way to do this is to

- leave globally free variables as they are;
- perform only top-level reductions;
- reduce only full applications and turn partial applications into some kind of closures

at run-time. All closures that are left over at the end of the processing phase are by the postprocessor reduced to weak head normal forms of new abstractions, using full-fledged β-reductions in order to perform the appropriate modifications of the index tuples.

3 Outline of the $\mathcal{L}i$SA Abstract Machine

Having developed a basic understanding of how $\mathcal{L}i$SA is supposed to work, we are now ready to define the underlying abstract machine. It is a high-level interpreter for λ-terms represented in a two-place constructor syntax, with an explicit applicator @ as the sole operator, and with index tupels $(i,j)^\lambda$ replacing bound variable occurrences. Moreover there is a special internal representation of letrec-constructs which has the form

$$\alpha\ expr_{goal}\langle expr_{f_1} \ldots expr_{f_n}\rangle$$

where α is a primitive recursion operator equivalent to an n-ary Y-combinator.

Thus, the internal syntax of legitimate $\mathcal{L}i$SA programs is as follows:

$$
\begin{array}{lll}
expr & \rightarrow & @expr_f\; expr_{arg} \\
 & & \Lambda.expr_{body} \\
 & & \alpha\; expr_{goal}\langle(f_1, expr_{f_1})\ldots(f_n, expr_{f_n})\rangle \\
 & & (i,j)^{\lambda} \\
 & & (expr, i)^{\alpha} \\
 & & const
\end{array}
$$

Using this syntax, the internal representation of the example program (figure 2) looks like this:

$$\alpha\; @@\,(f,1)^{\alpha}\,(f,1)^{\alpha}\;\mathbf{K}\left\langle\Big(f, \mathbf{\Lambda.\Lambda.}\alpha\; @\,(g,1)^{\alpha}\,(f,2)^{\alpha}\left\langle(g, \mathbf{\Lambda.}@@\,(1,1)^{\lambda}\,(1,0)^{\lambda}\,(0,0)^{\lambda})\right\rangle\Big)\right\rangle$$

where $(f,i)^{\alpha}$ designates an applied occurrence of a function. f refers to the function definition $expr_f$ contained in the letrec construct $\alpha\; expr\langle\ldots(f, expr_f)\ldots\rangle$, and i denotes the number of intervening nesting levels of letrecs between the function occurrence and its defining letrec.
The corresponding graph representation shows the recursive structure more clearly:

$$
\begin{array}{c}
\alpha \\
\text{@} \quad \Lambda \quad \Lambda \quad \alpha
\end{array}
$$

Program execution in $\mathcal{L}i$SA involves a preorder traversal of its graph representation in search for redices. The current traversal position is kept track of by a pointer into the graph. While traversing a spine of applicators, pointers to the argument terms are collected in the order from top to bottom on a run-time stack. A redex is identified by a primitive function or a Λ-abstraction in the left subtree of an applicator.

Reducing the application of a λ-abstraction, which is of primary interest in this paper, results in the extension of a copy of the actual argument pointer frame by taking as many pointers off the run-time stack as are required by the arity of the λ-abstraction. If there are fewer pointers on the stack, we have a partial application which is transformed into a closure. Graph templates and environments are linked together via indirection nodes held in yet another structure.

As in π-RED, reductions in $\mathcal{L}i$SA are controlled by a special reduction counter which, prior to every program run, must be initialized with some user-specified value. Every

instance of reduction decrements this value by one. If the counter is down to zero before having reached a weak head normal form, further reductions are suspended and $\mathcal{L}i$SA returns to the user interface the high-level representation of the graph constructed so far, i.e. an intermediate program. The primary purpose of this counter is to terminate in an orderly form potentially non-terminating recursions, but it may also be used to perform stepwise reductions under interactive control.
Thus, the abstract $\mathcal{L}i$SA machine can be defined by the state-transition function τ:

$$\tau : \Big(e, n, S, G, I, E, red\Big) \Longrightarrow \Big(e', n', S', G', I', E', red'\Big)$$

where

- e is the pointer to the current environment
- n is the pointer to the current traversal position in the graph
- S is the run-time stack
- G is the program graph
- I is the indirection node structure
- E is the environment structure
- red is the reduction counter value.

Furthermore, we use the notations

- $G\Big[n = expr\Big]$ for a subgraph term at pointer position n in the Graph G
- $\Big[a{:}B\Big]$ for the concatenation of a component a (say, a pointer or a frame of pointers) to an existing structure B.

In specifying this abstract machine, we will focus only on β-reductions, recursive function calls, the instantiation of bound variable occurrences (which are turned into index tuples), and on the orderly termination of reductions.

3.1 β-Reductions

To reduce an application $@\boldsymbol{\Lambda}.body\ expr$, two steps are necessary. Initially a pointer to the argument $expr$ is pushed onto the stack, and after that the abstraction body is evaluated: To do so, the argument is taken off the stack and concatenated to the current environment. $\mathcal{L}i$SA realizes these two steps by two state transition rules.

The first rule is responsible for placing the arguments on the stack. While doing this, each argument must be instantiated. This is done by generating an indirection node which links the graph node to the current environment.

$$\tau : \Big(e, n, S, I, G\Big[n = @\, n_f\ n\Big], E, red\Big) \Longrightarrow \Big(e, n_0, \Big[p{:}S\Big], I\Big[p = (n, e)\Big], G, E, red\Big)$$

The current point of reduction advances to the function referenced in the application.

If an abstraction is detected, $\mathcal{L}i$SA checks whether or not arguments are available on the stack. In the former case the argument is popped off the stack and a new environment frame is created by copying the actual environment frame and extending it with the popped reference. The new environment becomes the actual environment and the reduction proceeds. If the body of the abstractions contains further abstractions, they may consume the surplus arguments from the stack, if there are any.

$$\tau : \left(e, n, S, I, G\left[n = \mathbf{\Lambda}.n_b\right], E\left[e = \langle p_1 \ldots p_l\rangle \rightarrow \hat{e}\right], red\right)$$
$$\Longrightarrow \begin{cases} \left(e', n_b, S', I, G, E\left[e' = \langle p\, p_1 \ldots p_l\rangle \rightarrow \hat{e}\right], red{-}1\right) & \text{if } S = \left[p{:}S'\right] \wedge red > 0 \\ \left(e, n, S, I, G, E, red\right) & \text{else} \end{cases}$$

If a partial application is detected, which is recognized by an empty stack, the current pointer into the graph and the current environment are produced as a result. If the reduction counter expires, the leftover arguments on the stack have to be included into the result. To do so, $\mathcal{L}i$SA uses another state transition function τ_ϵ which controls the termination of the reduction. The details of τ_ϵ are explained in section 3.4.

3.2 Recursion

Since recursive functions are per sé in weak head normal form, they need not be embedded in an environment. All occurrences of function identifiers refer directly to the graph body of the function. To illustrate the reduction of letrec-constructs, we should consider another small example:

$(\lambda u.$**letrec**
$\quad f = \ldots$
in $(\lambda v. \ldots f \ldots\ e_v)\ e_u)$

After having reduced the othermost redex, the actual environment contains the argument term for u, i.e., $env = \langle (u, e_u, nil)\rangle$. Since this environment contains the instantiation of the relatively free variable u of the function f, env is required when reducing function calls of f (see also section 2).

Assuming that the traversal of the letrec-construct just results in evaluating the letrec's body, the reduction of the goal-redex leads to an environment $env'{=}\langle (v{,}e_v{,}env), (u{,}e_u{,}nil)\rangle$. Thus the environment env can not be derived from the actual environment env' in which f would be called.

In order to be able to install the environment env, a new empty environment-frame has to be created and chained up to the actual environment whenever a letrec-construct is traversed. This is described by:

$$\tau : \left(e, n, S, I, G\left[n = \alpha_m n'\langle f_1, \ldots, f_m\rangle\right], E, red\right)$$
$$\Longrightarrow \left(e', n', S, I, G, E\left[e'{=}\langle\rangle{\rightarrow}e\right], red\right)$$

Thus when the letrec's body is done, the actual environment in the above example is $env'' = \langle (v, e_v, env)\rangle \rightarrow \langle (u, e_u, nil)\rangle$. Since the pre-processor can easily determine how many frames will be built up between the actual environment and the environment that contains the relatively free variables, functions are represented as $(expr, n)^\alpha$, where n indicates the number of environment frames which have to be deleted from the actual environment. Hence the state transition specifying a function call is denoted as follows:

$$\tau : \Big(e, n, S, I, G\Big[n = (n', j)^{\alpha}\Big], E\Big[e \rightarrow e_1 \rightarrow \ldots \rightarrow e_j\Big], red\Big)$$
$$\Longrightarrow \begin{cases} \Big(e', n', S, I, G, E\Big[e' = \langle\,\rangle \rightarrow e_j\Big], red\Big) & \text{if } red > 0 \\ \Big(e, n, S, I, G, E, red\Big) & \text{else} \end{cases}$$

If the reduction counter is down to zero, $\mathcal{L}i$SA terminates the reduction process by using τ_{ϵ} for further state transitions.

3.3 Variables

Whenever the current point of reduction is an index tupel $(i, j)^{\lambda}$, representing a bound variable, processing continues at the indirection node found in the j-th entry of the i-th environment frame. Since other parts of the expression may reference the same instance of this variable, the environment must be updated with the evaluated expression. Thus $\mathcal{L}i$SA has to keep track of the environment entries that must be updated. Explicit *update marks* are placed on the S-stack to ensure that the environment is updated with the expression obtained after termination of the reduction sequence.

As a shortcut, $\mathcal{L}i$SA places an update mark onto the stack only if an expression is read from the environment which may contain redices. Atomic values are in normal form, so no update marks are needed.

$$\tau : \Big(e, n, S, I\Big[p_j = (n_j, e_j)\Big], G\Big[n = (i, j)^{\lambda}\Big], E\Big[e \rightarrow e_1 \rightarrow \ldots \rightarrow e_i = \langle p_0 \ldots p_j \ldots p_m \rangle\Big], red\Big)$$
$$\Longrightarrow \begin{cases} \Big(e_j, n_j, \Big[(\$_{Up}, p_j) \mathbin{:} S\Big], I, G, E, red\Big) & \text{if } n_j \in \{@, \alpha\} \\ \Big(e_j, n_j, S, I, G, E, red\Big) & \text{else} \end{cases}$$

Since the update mechanism is only invoked by environment lookups, the number of updates that are actually carried out is restricted to potentially shared expression parts. Many applications, especially those of primitive functions, don't need to be updated.

The update marks become relevant when popping argument terms off the stack. If an update mark $(\$_{Up}, p)$ is found, the current graph pointer, together with the actual environment, represents the evaluated expression which originally was taken from the indirection node by an environment lookup. The evaluated expression has to replace to unevaluated expression in the indirection node.

$$\tau : \Big(e, n, \Big[(\$_{Up}, p) \mathbin{:} S\Big], I\Big[p = (n', e')\Big], G\Big[n = \mathbf{\Lambda}.n_b\Big], E, red\Big)$$
$$\Longrightarrow \Big(e, n, S, I\Big[p = (n, e)\Big], G, E, red\Big)$$

3.4 Terminating the Reduction

Upon termination the state transition function τ_{ϵ} has to do some cleaning up in order to return the evaluated program in an orderly form to the user.

As mentioned before, there are two ways to terminate the reduction process: Whenever the reduction counter is down to zero, i.e. further reductions have to be suppressed, τ_{ϵ} has to reconstruct applications from arguments left over on the stack.

$$\tau_{\epsilon} : \Big(e, n, \Big[p \mathbin{:} S\Big], I, G, E, red\Big)$$
$$\Longrightarrow \Big(e, n', S, G\Big[n' = \overline{@}\, n\, p\Big], E, red\Big)$$

The applications thus created are marked since the argument terms are already embedded in environments which must be resolved by the postprocessor.

When encountering a top-level partial application, $\mathcal{L}i$SA terminates the reduction process since partial applications, by definition, can not be reduced during the processing phase.

$$\tau_\epsilon : \Big(e, n, S, I, G\Big[n = \mathbf{\Lambda}.n_b\Big], E, red\Big) \quad \text{where} \quad S = \Big[\,\Big] \vee red = 0$$
$$\Longrightarrow \tau_\epsilon\Big(e, n, S, I, G, E, red\Big)$$

4 About the Performance of $\mathcal{L}i$SA

This paper primarily focusses on how β-reductions and recursive function calls are realized in $\mathcal{L}i$SA.The performance measurements to be discussed in this section are therefore done with small example programs which stress these mechanisms in a pure setting, keeping other influences at a minimum.

These measurements are compared against those taken for the same programs on π-RED*, which is the applicative order counterpart of $\mathcal{L}i$SA, and from which many of the components of $\mathcal{L}i$SA are adopted. Thus we have a good basis for a comparison which brings out essential differences between both approaches of performing reductions.

The problem, to some extent, is the choice of a suitable set of example programs. On the one hand, they should be exactly the same for both systems in order to eliminate influences due to different algorithmic solutions for the same application problems, on the other hand we face difficulties insofar as

- programs which take full advantage of lazy evaluation, e.g., are specified as consumer-producer problems, do not terminate when executing them under an applicative order regime;

- programs which also terminate under an applicative order regime generally perform poorly when executed under a lazy regime.

However, with $\mathcal{L}i$SA and π-RED* we are in the fortunate position that both systems include a counting mechanism which allows only for pre-specified numbers of reduction-steps. Thus, we can use for comparison programs which engage in non-terminating recursions, and measure the time it takes for both systems to perform a given number of reduction steps. Moreover, since no termination conditions, e.g. in the form of **if-then-else** clause, need to be included, these programs may be simply specified as terms of a pure λ-calculus enhanced be a primitive recursion operator. Thus, we decided to use the test programs of figure 3.

The performance figures collected in table 1 show that under otherwise fairly identical conditions $\mathcal{L}i$SA run-times are by about a factor of 1.75 higher on average than those of π-RED*,which is the price to be paid for lazy vs. eager evaluation with respect to β reductions and recursions.

All the measurements are done on a SPARC-SUN IPC with 24MB main memory under the operating system SunOS Release 4.1.1. The run-times listed are the arithmetic average of three values obtained by the UNIX timer command.

$simple\_y =$ **let**
$A = \lambda x.\lambda y.(\ y\ (\ (\ x\ x\ )\ y\ )\ )$
in $(\ (\ A\ A\ )\ \lambda x.x\ )$

$omega = (\ \lambda x.(\ x\ x\ )\ \lambda x.(\ x\ x\ )\ )$

$y1 =$ **letrec**
$F = \lambda x.(\ F\ x\ )$
in $(\ F\ 3\ )$

$y2 =$ **letrec**
$F = \lambda x.\lambda y.(\ (\ F\ x\ )\ y\ )$
in $(\ (\ F\ 3\ )\ 42\ )$

$y3 =$ **let** $A = 1,$
$B = 1$
in letrec
$F = \lambda x.\lambda y.(\ (\ F\ A\ )\ B\ )$
in $(\ (\ F\ 3\ )\ 42\ )$

Figure 3: Test programs for β-reductions and primitive recursions

Example	Complexity [Reduction-steps]	Run-time $\mathcal{L}i$SA [sec]	Run-time π-RED* [sec]	Run-time-quotient $\mathcal{L}i$SA/π-RED*
simple_y	10000	0.75	0.84	0.9
	50000	3.90	–	–
	100000	7.82	–	–
omega	10000	0.63	0.39	1.62
	50000	3.16	2.00	1.58
	100000	6.31	3.85	1.64
$y1$	10000	0.62	0.33	1.88
	50000	3.16	1.64	1.93
	100000	6.04	3.26	1.85
$y2$	10000	0.74	0.44	1.68
	50000	3.66	2.19	1.67
	100000	7.40	4.37	1.69
$y3$	10000	0.73	0.44	1.66
	50000	3.68	2.20	1.67
	100000	7.36	4.38	1.68

Table 1: Run-times for the above examples

A more complex example program which takes full advantage of lazy evaluation is the well-known prime-sieve of Erathostenes, which in sugared lambda notation is given in figure 4.

In order to have it terminate on π-RED* as well, this program must be modified accordingly. In doing this we can take advantage of the fact that π-RED* internally represents the alternative terms of an **if-then-else** clause as components of a special construct which looks like a CONSed binary list whose components are evaluated only on demand, i.e. whenever the predicate term has reduced to a Boolean constant. Thus we

$$
\begin{array}{l}
\textbf{letrec} \\
\quad gen = \lambda n.\Big[n{:}(gen\ ((+\ n)\ 1))\Big] \\
\quad sel = \lambda n.\lambda l.\textbf{if}\ ((\mathrm{eq}\ n)\ 1)\ \textbf{then}\ (\mathrm{hd}\ l)\ \textbf{else}\ \Big(\Big(sel\ ((-\ n)\ 1)\Big)\ (\mathrm{tl}\ l)\Big) \\
\quad sieve = \lambda l.\textbf{let}\ p = (\mathrm{hd}\ l), \\
\qquad\qquad\qquad\quad l = (\mathrm{tl}\ l) \\
\qquad\qquad\quad \textbf{in letrec} \\
\qquad\qquad\qquad filter = \lambda l.\textbf{let}\ e = (\mathrm{hd}\ l), \\
\qquad\qquad\qquad\qquad\qquad\qquad l = (\mathrm{tl}\ l) \\
\qquad\qquad\qquad\qquad\quad \textbf{if}\ \Big(\Big(\mathrm{eq}\ ((\mathrm{mod}\ e)\ p)\Big)\ 0\Big)\ \textbf{then}\ (filter\ l)\ else\Big[e{:}(filter\ l)\Big] \\
\qquad\qquad\quad \textbf{in}\Big[p{:}(sieve\ (filter\ l))\Big] \\
\textbf{in}\ \Big((sel\ 100)\ (sieve\ (gen\ 2))\Big)
\end{array}
$$

Figure 4: Prime-sieve of Eratosthenes

can use for applicative order reductions essentially the same program except that CONSed lists must be replaced by **then-else** constructs, and **head/tail** calls must be replaced by applications to the selectors **true/false**, respectively.

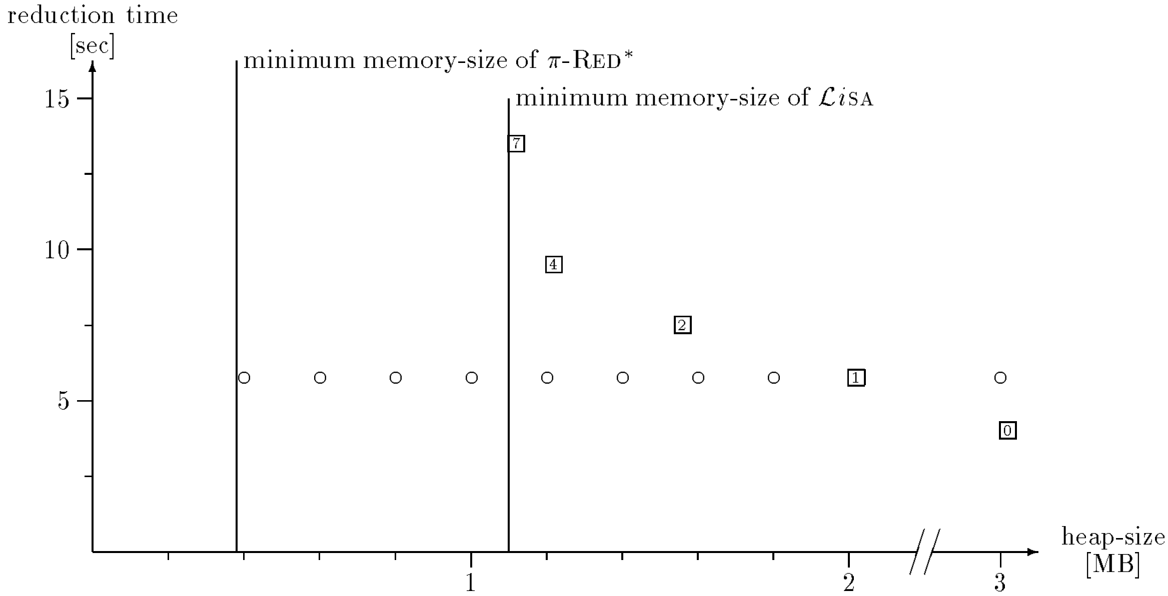

Figure 5: Run-times in relation to provided memory-size of the above example

A comparison of the performance figures for this program must take into consideration that $\mathcal{L}i$SA and π-RED* use different heap-management schemes. While $\mathcal{L}i$SA works with a conventional garbage collection scheme, π-RED* employs reference counting and releases unused heap-space as early as possible. The results are depicted in figure 5 where the symbols ◯ represent the values for π-RED*, and $\boxed{n}$ represent the values for $\mathcal{L}i$SA, with

n indicating the number of garbage collections performed during the reduction. Thus π-RED* runs the program in minimal space and does not improve its performance when supplying more space. In contrast to this, the performance of $\mathcal{L}i$SA critically depends on the available heap-space, with run-times by factor of about three higher than π-RED* with minimal heap-size, and falling below the run-time of π-RED* as the heap-size increases.

5 Acknowledgements

We are grateful to H.Blödorn for many valuable discussions about the concept and implementation of $\mathcal{L}i$SA.

References

[Augu87] L.Augustsson: *Compiling Lazy Functional Languages* PhD Thesis, Chalmers University of Technology, Göteborg, 1987

[Bare84] H.P. Barendregt: *The λ-Calculus, its Syntax and Semantics* Studies in Logic, Vol. 103, North Holland, 1984

[Berk76] K.J.Berkling: *A Symmetric Complement to the Lambda Calculus* Internal Report GMD ISF-76-7, Sankt Augustin, September 1976.

[Berk86] K.J.Berkling: *Headorder reduction: a graph reduction scheme for the operational lambda calculus* LNCS 279, 1986.

[Bloe89] H. Blödorn: *KiR- The Kiel Reduction Language Users Guide* Internal Paper, University of Kiel, 1989.

[DeBr72] N.G. De Bruijn: *Lambda–Calculus Notation with Nameless Dummies. A Tool for Automatic Formula Manipulation with Application to the Church–Rosser–Theorem* Indigationes Matematicae 34, 381–392, 1972.

[Gaer91] D. Gärtner: π − RED$^+$*: ein interaktives codeausführendes Reduktionssytem zur vollständigen Realisierung eines angewandten λ-Kalküls* PhD Thesis in German, University of Kiel, 1991.

[Hind86] J.R. Hindley; J.P Seldin: *Introduction to combinators and λ-Calculus* London Mathematical Society Student Texts 1, Cambridge University Press, 1986

[John84] T. Johnson: *Efficient compilation of lazy evaluation* Proceedings of the SIGPLAN '84 Symposium on Compiler Construction , pp. 58–69, Montreal 1984.

[John85] T. Johnson: *Lambda Lifting: Transforming Programs into Recursive Equations* FPLCA, LNCS 201, Nancy, September 1985.

[John86] T. Johnson: *Taget Code Generation from G–Machine Code* LNCS 279, 1986.

[Klug86] W. Kluge; C. Schmittgen: *Reduction Languages and Reduction Systems* LNCS 272, pp. 153-184, 1986.

[Miln78] R. Milner: *A Theorie of Type Polymorphism in Programming* Journal of Computer System Sciences, Vol. 17, pp. 348-375, 1978

[ML87] D. M. Queen et al: *Functional Programming In ML*, 1987.

[PJ87] S. L. Peyton Jones: *The Implementation Of Functional Programming Languages.* Prentice Hall, London, 1987.

[Schm92] C. Schmittgen; H. Blödorn; W. Kluge: π-RED*– *a Graph Reducer for Full-Fledged λ-Calculus* NGC, Vol. 10(2), Springer Verlag 1992

[Turn79] D.A. Turner: *A New Implementation Technique of Applicative Languages* Software Practice and Experience, Vol. 9, pp. 31-49, 1979