The Very Lazy λ-Calculus and the STEC Machine

Jan Rochel *

Universiteit Utrecht, The Netherlands
Department of Computer Science
rochel@cs.uu.nl

Abstract. Current implementations of non-strict functional languages rely on call-by-name reduction to implement the λ-calculus. An interesting alternative is *head occurrence reduction*, a reduction strategy specifically designed for the implementation of non-strict, purely functional languages. This work introduces the *very lazy λ-calculus*, which allows a systematic description of this approach. It is not based on regular β-reduction but a generalised rewriting rule called *γ-reduction* that requires fewer reductions to obtain useful results from a term. It therefore promises more efficient program execution than conventional execution models. To demonstrate the applicability of the approach, an adaptation of the Pointer Abstract Machine (PAM) is specified that implements the very lazy λ-calculus and constitutes a foundation for a new class of efficient functional language implementations.

1 Introduction

The λ-calculus is the foundation for the semantics of functional programming languages. Decades of research on the compilation and execution of non-strict functional languages has resulted in a number of different abstract machines such as in [Fairbairn 1987] [Peyton Jones 1987] [Burn 1988] [Peyton Jones 1992] [Holyer 1998] [Leijen 2005] [Krivine 2007]. They implement the λ-calculus by applying non-strict (or *lazy*) reduction strategies, such as *call-by-name* reduction.

A promising alternative is the Pointer Abstract Machine [Danos 2004], which is based on a reduction strategy that is lazier than *call-by-name* reduction in a certain sense. The Pointer Abstract Machine (PAM) is derived from a generalised version of the λ-calculus and then extended to support the range of features required for the implementation of a fully-fledged functional programming language. The result is the STEC machine, a concrete, implementation-oriented manifestation of the PAM.

After giving a brief recapitulation of the λ-calculus and lazy evaluation we introduce the *very lazy λ-calculus*, which forms the basis of the approach. It relies on a generalisation of β-reduction that leads to a new reduction strategy, called *head occurrence reduction*. We systematically develop the STEC machine, an abstract machine for

* Many thanks to Carsten Sinz, Patrik Jansson, and Daniel P. Friedman whose kind support was indispensable for the publication of this work, and also to Vincent van Oostrom and Laurent Regnier for their helpful comments.

the very lazy λ-calculus. It has unique characteristics that promises high-performance program execution. In the last section we discuss opportunities for further research in order to create a new kind of efficient functional language implementation.

2 Basics

2.1 The λ-Calculus

The pure, untyped λ-calculus [Barendregt 1984] is a term rewriting system that operates on terms called λ-*expressions*. For a given set of variables V they are defined by:

$$\begin{array}{rcll} E & ::= & \lambda V.E & (\textit{abstraction}) \\ & | & (E\,E) & (\textit{application}) \\ & | & V & (\textit{variable}) \end{array} \qquad (\lambda\textit{-expression})$$

We henceforth assume that $x, y, z \in V$ and $e, e_1, e_2 \in E$ and also that for each $v \in V$, abstractions of the form $\lambda v.E$ occur at most once in a term. This simplification conforms to the handling of the name-capture problem in the context of programming language implementation where at compile-time variables are resolved to an unambiguous representation.

Three rewrite rules are defined for the evaluation of such expressions: α-, *β*-, and η-*conversion*. For the implementation of functional programming languages, α- and η-conversion are of minor importance and are not discussed here, leaving *β-conversion* as the central evaluation mechanism of the λ-calculus.

As long as the substitution variable x occurs at most once beneath the reduced λ-abstraction, β-conversion reduces the size of the expression, therefore it is more often than not called *β-reduction* and is defined by:

$$(\lambda x.e_1)\,e_2 \longrightarrow_{\beta_x} e_1[x := e_2] \qquad (\beta\textit{-reduction})$$

A term is called a *reducible* expression (*redex*) if it has the form $(\lambda x.e_1)\,e_2$. A term in which no redexes occur is in *normal form* (NF).

2.2 Lazy Evaluation

In the implementation of programming languages the evaluation of an expression yields the result of a computation. The analogy of a result in the λ-calculus is however not fully obvious. While a term in NF can be considered a result (as it can no longer be reduced) in functional languages it turns out to be overkill to reduce any given term to NF. Instead, other forms that may still contain redexes are targeted, like *weak normal form* (WNF), *head normal form* (HNF), or *weak head normal form* (WHNF), specified as follows, where $AB^* ::= A \mid (AB)\,B^*$.

$$\begin{array}{lll} E_{\mathrm{NF}} & ::= \lambda V.E_{\mathrm{NF}} \mid V\,E^*_{\mathrm{NF}} & (\textit{normal form}) \\ E_{\mathrm{WNF}} & ::= \lambda V.E \mid V\,E^*_{\mathrm{WNF}} & (\textit{weak normal form}) \\ E_{\mathrm{HNF}} & ::= \lambda V.E_{\mathrm{HNF}} \mid V\,E^* & (\textit{head normal form}) \\ E_{\mathrm{WHNF}} & ::= \lambda V.E_{\mathrm{WHNF}} \mid V\,E^* & (\textit{weak head normal form}) \end{array}$$

If such a form semantically relates to a meaningful concept of a result, by evaluating to this form instead of NF, redundant reductions can be avoided. To ensure no effort is squandered due to such redundant reductions, a practical implementation of the λ-calculus requires a well-defined scheme (*reduction strategy*) to select for each β-reduction step a non-redundant abstraction. Well-established examples are: *normal order* reduction to NF, *hybrid normal order* reduction to NF, *applicative order* reduction to NF, *hybrid applicative order* reduction to NF, *call-by-value* reduction to WNF, *head spine* reduction to HNF, *call-by-name* reduction to WHNF [Sestoft 2002].

Generally, non-strict languages imply the use of one of the last two strategies, which never reduces redexes that occur within an argument. Moreover call-by-name never reduces redexes beneath a λ-abstraction.

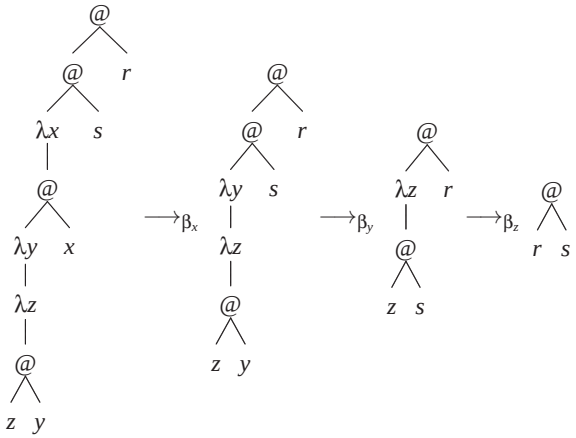


Fig. 1. Call-by-name reduction to WHNF

3 The Very Lazy λ-Calculus

To motivate the derivation of a new calculus it is helpful to relate to the properties specific to the targeted class of programming languages. Therefore we refer to language elements like constructors and case discrimination without explicitly introducing them (as a part of the calculus). Constructors can be thought of as free variables.

For the implemention of a non-strict language, it is in general desirable to increase the degree of laziness by using a normal form that requires fewer reductions. But then such a form is useless if it does not reflect the result of a computation in a sensible manner. Both WHNF and HNF fail to accomplish this with adequate precision in regard to the specifics of non-strict functional languages.

Operationally both forms of lazy evaluation (head spine reduction and call-by-name reduction) proceed by walking down the spine, reducing any occurring abstrac-

tion/application pairs on the way until the tip of the spine is reached. Further action then depends on the quality of this value.

Let us assume that an expression $e = ((\lambda x.y)\, e_1)\, e_2$ is the scrutinee in a case discrimination. The selection of the case alternative then solely depends on the constructor at the head position of its normal form $e_{NF} = y\, e_2$. However if y is a constructor, e already is in a form that at the tip of its spine without the need for any reductions reveals the value required to select the appropriate case alternative. Then why should all satisfied abstractions above y be reduced before effecting the case discrimination? After all such reductions may never affect a free variable at the tip of the spine.

Thus, we attempt to specify a calculus with a normal form that accurately captures this formulation of non-strict semantics, and a reduction strategy that efficiently reduces to this normal form. Both concepts relate to the variable at the tip of the spine, which we refer to as the *head occurrence* (hoc) [Danos 2004] of an expression:

$$\begin{aligned} hoc(\lambda x.e) &= hoc(e) \\ hoc(e_1\, e_2) &= hoc(e_1) \\ hoc(x) &= x \end{aligned} \qquad \text{(\textit{head occurrence})}$$

3.1 The Quasi Head Normal Form

The normal form of the very lazy λ-calculus is called *quasi head normal form* (QHNF) [Danos 2004]. We give a definition that is more straightforward than the original one by relating to the hoc of the NF:

$$E_{\text{QHNF}} := \{e \in E \mid hoc(e) = hoc(e_{NF})\} \qquad \text{(\textit{quasi head normal form})}$$

An optimal reduction strategy that evaluates to QHNF in a minimum number of steps must not perform unneeded reductions. The most direct approach for such a strategy is to repeatedly substitute the variable t at the tip of the spine (hoc) by reducing the corresponding abstraction λt until QHNF is obtained. This is generally not possible with β-reduction, however. The term $e = ((\lambda x.(\lambda y.y))\, e_1)\, e_2$ for instance is not in QHNF, yet the λ-calculus does not allow substituting for y, as λy occurs directly beneath another abstraction λx and therefore cannot be β-reduced before λx.

Considering this restriction of the β-reduction as an unnecessary shortcoming of the λ-calculus, we now attempt to generalise β-reduction in order to make it more powerful.

3.2 The γ-Reduction

The very lazy λ-calculus evaluates λ-expressions by applying the γ-*reduction* rule, which allows reductions of abstraction/application pairs along the spine that are not adjacent to each other. We write $e_1 \longrightarrow_{\gamma_x} e_2$ to denote a γ-reduction $e_1 \longrightarrow_{\gamma} e_2$ that uses x as a substitution variable:

$$\frac{p_0(e_1) = x}{e_1\, e_2 \longrightarrow_{\gamma_x} e_1[\lambda x.e := e[x := e_2]]} \qquad \text{(γ-\textit{reduction})}$$

Thereby p_0 is a function that implements a simple parentheses-matching algorithm treating applications as left and abstractions as right parentheses. The idea is to identify

abstraction/application pairs along the spine that would be β-reduced in the course of head spine reduction to HNF. Subsequently, any of these pairs can be reduced individually even if the abstraction node is not directly adjacent to the application node.

$$\begin{array}{ll} p_0(\lambda x.e) = x & \\ p_i(\lambda x.e) = p_{i-1}(e) \quad (i > 0) & \text{(\textit{abstraction/application matching})} \\ p_i(e_1\, e_2) = p_{i+1}(e_1) & \end{array}$$

In the definition of γ-reduction above, $p_0(e_1)$ walks down the spine to locate the abstraction that matches the argument e_2. This permits γ-reduction to skip over abstraction and application nodes that occur in-between λx and e_2 that would have been reduced by conventional non-strict reduction strategies. For an example see *Fig. 4*.

A proof of the consistency of γ-reduction with the semantics of the λ-calculus is not given here, but much as in [Kamareddine 2001] β-equivalence is easily deduced by decomposing γ-reduction into a β-reduction embedded in a sequence of β-equivalent rearrangements of the spine. Moreover γ-reduction is a generalisation of β-reduction:

$$\begin{array}{c} e_1 = \lambda x.e \implies p_0(e_1) = p_0(\lambda x.e) = x \\ \implies e_1\, e_2 \longrightarrow_{\gamma_x} e_1[\lambda x.e := e[x := e_2]] = e[x := e_2] \end{array}$$

We notice that indeed the β-irreducible expression e from above is γ-reducible:

$$((\lambda x.(\lambda y.y))\, e_1)\, e_2 \longrightarrow_{\gamma_y} (\lambda x.e_2)\, e_1$$

3.3 Quasi Head Normal Form, revisited

Based on γ-reduction, QHNF can alternatively be redefined as

$$E_{\text{QHNF}} ::= \lambda V.E_{\text{QHNF}} \mid E_{\text{QHNF}}\, E^* \mid i \qquad \text{(\textit{quasi head normal form})}$$

where i is a variable not substitutable by a γ-reduction. This is the case if either the hoc i is free (e.g. a constructor), or if the corresponding abstraction λi is *unsatisfied* (i.e. there is no matching application).

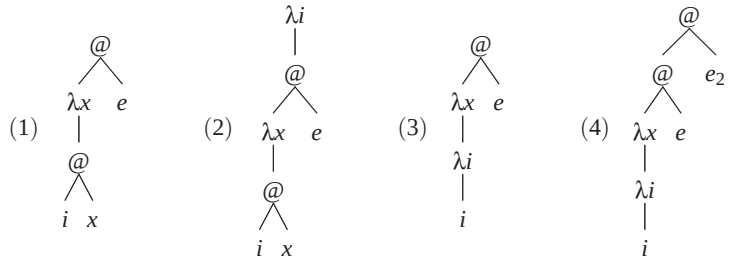
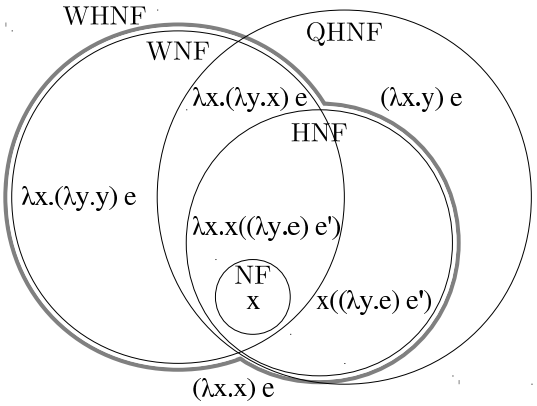


Fig. 2. Terms (1-3) are in QHNF, (4) is not as i is γ-reducible

To see that both definitions of QHNF match, we show that for some term e the $hoc(e)$ is γ-irreducible if and only if $hoc(e) = hoc(e_{NF})$. This follows from the robustness of the parentheses-matching algorithm in respect to γ-reductions, which only ever

reduce *matching* abstraction/application pairs from the spine. Because the γ-irreducible $hoc(e)$ cannot be substituted, it follows by induction that $hoc(e)$ remains at the tip of the spine during the entire γ-reduction to normal form and therefore $hoc(e) = hoc(e_{NF})$.

Conversely if $hoc(e) = hoc(e_{NF})$, γ-reduction may never substitute $hoc(e)$ since otherwise it would not be β-equivalent.



$$
\begin{aligned}
E_{\mathrm{NF}} &\subset E_{\mathrm{HNF}} \cap E_{\mathrm{WNF}} \\
E_{\mathrm{WHNF}} &= E_{\mathrm{HNF}} \cup E_{\mathrm{WNF}} \\
E_{\mathrm{QHNF}} &\supset E_{\mathrm{HNF}} \\
E_{\mathrm{QHNF}} &\not\supset E_{\mathrm{WHNF}}
\end{aligned}
$$

Fig. 3. Set relations between various normal forms

3.4 Head Occurrence Reduction

Based on this definition we can define an optimal reduction strategy to QHNF. γ-reductions that substitute the hoc are clearly sufficient and are always needed. We call the reduction strategy that in each step substitutes the hoc of the term using a γ-reduction *head occurrence reduction*:

$$\frac{e \longrightarrow_{\gamma_t} e' \quad t = hoc(e)}{e \longrightarrow e'} \qquad v \longrightarrow v \qquad \frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}$$

The evaluation of a term according to head occurrence reduction in each step needs to identify three nodes affected by the reduction of the graph: $t = hoc(e)$ at the tip of the spine, the corresponding abstraction node λt, which is located further up the spine, and the matching application node with the right-hand side e_2 even further up the spine.

Head occurrence reduction is lazier than conventional lazy reduction strategies in the sense that it reduces to a normal form that expresses the semantics of non-strict functional languages more accurately than WHNF. Thus reductions are avoided that deal with arguments of the result prematurely.

3.5 Examples

To compare our reduction strategy to conventional lazy evaluation, consider the term $((\lambda x.(\lambda y.\lambda z.zy)\,x)\,s)\,r$. Three β-reductions are required for call-by-need reduction to

WHNF (*Fig. 1*). For the same term, head occurrence reduction requires only one γ-reduction to reduce to QHNF (*Fig. 4*). Furthermore β-reduction can not produce the depicted transition. Pathological cases can be constructed, such as $(\lambda x_1 \ldots \lambda x_n.\lambda y.y)\, e_1 \ldots e_n$ or $(\lambda y(\lambda x_1 \ldots \lambda x_n.y)\, e_1 \ldots e_n)\, e$ that require n additional reductions to obtain WHNF.

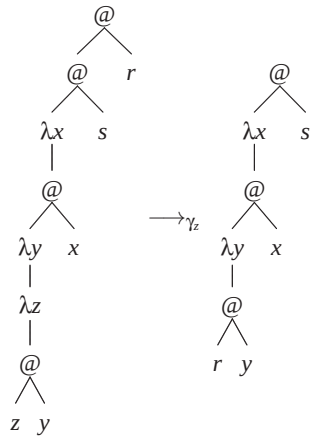


Fig. 4. head occurrence reduction to QHNF

4 The STEC-Machine

We now derive an abstract machine that implements the very lazy λ-calculus exploiting its particular properties for improved efficiency. It is an adaptation of the PAM enriched by language elements like case discrimination and primitive functions to support practical functional languages.

A dominant issue in the design of such an abstract machine is that terms representing nontrivial programs are graphs with directed cycles rather than trees. This is due to functions that are used at different sites in the program definition, and may involve (mutual) recursion. So we cannot statically unfold the graph, since the resulting tree would be of infinite size. Therefore the graph needs to be expanded incrementally during evaluation. There are various solutions to this, from simple approaches like copying parts of the graph as needed, to more sophisticated techniques like super-combinator compilation [Hughes 1982].

Here however, we explore a new direction where the abstract machine's main runtime data structures remain unmodified once instantiated. While this seems contrary to the notion of graph rewriting, the approach combines well with the very lazy λ-calculus. Let us first take a glance at the untyped language interpreted by the abstract machine.

4.1 Abstract Machine Language, pure version

The term to be evaluated is given as a program definition comprising a set of function definitions of the form:

$$f = \lambda x_1 \ldots x_m . a_0 \ldots a_n \qquad m, n \geq 0$$

The *arity* of a function f denotes the number of parameters, here $arity(f) = m$. On its right-hand side it specifies a non-empty list of arguments $args(f) = a_0 \ldots a_n$ that can be individually addressed by index: $args_i(f) = a_i$. Note that only $a_1 \ldots a_n$ represent application nodes. Consequently a_0 is not included in the argument count $|args(f)| = n$.

The language interpreted by the STEC-machine is a simple, untyped, functional language with a flat structure, i.e. all arguments of a function f are atomic, such that each of f's arguments $args(f)$ is a variable, either addressing a function or a parameter. Non-atomic expressions in the source language occur through the placement of parentheses or other language constructs that lead to the nesting of expressions. The atomicity property is easily enforced at compile-time by factoring each non-atomic argument into a separate function definition. This atomicity of the function arguments induces a certain kind of linearity that characterises the evaluation procedure to a large extent.

It is understood that in a compiled setting, numeric rather than symbolic values are used to reference functions and parameters. Functions are referenced by the address of the memory location of their definition. It is straightforward to reference parameters by their index as they occur in the function's parameter list. However, the scope of a function f extends beyond its own parameter list. On the right-hand side of f not only f's own parameters may be referenced but also parameter variables that occur free in f. Therefore to unambiguously address a specific parameter not only its index but also the associated function must be specified.[1] We use $\mathtt{P}_i^f$ to denote f's ith parameter. This may be thought of as a form of reversed de-Bruijn index [De Bruijn 1972] with a pivot.

Another technique employed by today's functional language implementations to cope with free variables is λ-*lifting*, however this transformation is just the opposite of what we want to accomplish. Rather its reverse transformation called λ-*dropping* [Danvy 2000] might integrate well with our execution model.

$$\begin{array}{lll}
\textit{program-definition} & ::= & \textit{function-definition}^+ \\
\textit{function-definition} & ::= & \textit{function-id}_{\textit{arity}}\ \textit{argument}^+ \\
\textit{arity} & ::= & \mathbb{N}^0 \\
\textit{argument} & ::= & \textit{function-id} \mid \mathtt{P}_{\mathbb{N}^+}^{\textit{function-id}}
\end{array}$$

Fig. 5. Abstract syntax of the STEC machine language

In the absence of named parameters, we do not need to maintain parameter lists. Instead we merely need to specify the arity of each function. We thus obtain a specification of the abstract machine language that represents as a function definition a term

[1] Instead of naming f explicitly, also the *nesting distance* between f and the referencing argument could be used, which is however less descriptive.

of the pure λ-calculus as a spine-sequence of abstraction nodes followed by application nodes (*Fig. 5*). Each function definition can be addressed by a unique function ID, which can be regarded as a function name. However, in compiled form is conveniently the memory address of the function definition.

What follows is a description of the dynamic behaviour of the STEC-machine and its data structures created at run-time. During the evaluation, the program definition is accessed only through the *arity*- and the *args*-functions. It is purely static data, i.e. it is generated at compile-time and no rewriting takes place on the original function definitions.

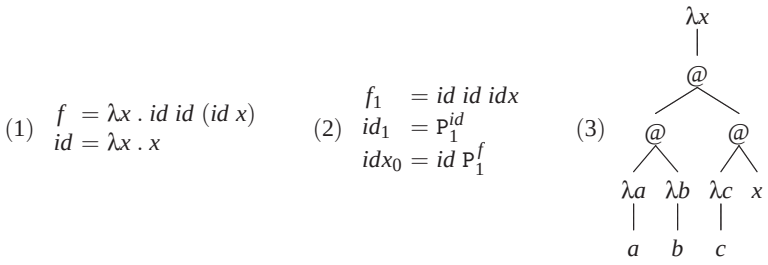


Fig. 6. Term given as (1) a λ-expression, (2) STEC machine code, (3) a fully-expanded graph

4.2 Graph Expansion

In each step of the evaluation, head occurrence reduction performs γ-reductions that substitute the variable at the tip of the spine (hoc). Therefore not only the appropriate abstraction/application pair must be located, also the hoc is usually not immediately at hand due to the fragmentation of the graph into function definitions. Thus walking down the spine to reach its tip often requires a series of graph expansions.

The root of the term to be evaluated is specified by a designated function f, whose definition directly represents the topmost fragment of the term.[2] If the hoc is not immediately visible, that is if the leftmost argument $args_0(f)$ references a function g rather than a parameter, then the graph has to be unfolded by instantiating g in order to locate the hoc of the spine within g's definition.

While this at first might seem like a description of regular non-strict function calls, those in the course of the instantiation also immediately pass the arguments supplied by its caller to the callee. There are two possibilities that perform such function calls, the *push/enter* and the *eval/apply* method [Marlow 2006]. Ultimately this is where β-reductions take place in conventional functional language implementations.

The very lazy λ-calculus however allows the β-reduction to be omitted, thus no arguments are passed to g. Therefore, according to head occurrence reduction, γ-reductions

[2] Generally this function is named `main` or similarly.

cannot take place before the tip of the spine is revealed. Until then the abstract machine simply proceeds to build the graph while walking down the spine.

As the graph is only expanded along its spine, it has a linear structure in the form of a series of functions that have been stuck together, which is easily represented using a stack. Instead of explicitly maintaining abstraction and application nodes (replicated from the function definitions), for efficiency, we use entire functions as the unit of the run-time data structure.

4.3 The Evaluation Stack

These functions are represented by function *instances*, which hold a pointer to the corresponding function definition and act as a copy of the function. Thus the primary run-time data structure of the STEC-machine is a stack of instances, the *evaluation stack*. It grows from right to left and unlike a usual stack, read accesses *within* the evaluation stack are permitted. Instances are addressed according to their stack position. The notation for an evaluation stack containing n instances:

$$E ::= I_n\ I_{n-1}\ \ldots\ I_1 \qquad \text{(\textit{evaluation stack})}$$

Besides the evaluation stack, the state of the abstract machine comprises a *status register* S that specifies the action that is to be taken next, and a *target register* T that points to the *stack address* targeted by the action:

$$STE ::= (S,T,E) \qquad \text{(\textit{configuration})}$$

Summarising, the evaluation stack encodes the current term as a sequence of function instances, each of them representing a segment of the term's spine. The graph is expanded along its spine as long as the leftmost instance t references another function in its 0th argument, namely if $args_0(f) = g$, assuming t is an instance of f. We say that an *argument request* $\mathtt{A}_0$ is issued in order to examine the 0th argument of f. A graph expansion takes place by pushing another instance (in this case of g) on the stack.

4.4 Locating an Abstraction

At some point the tip of the spine (hoc) is reached, which is indicated by the 0th argument of the leftmost instance being a parameter $\mathtt{P}_i^f$ rather than a function reference. In order to effect a γ-reduction, the corresponding abstraction/application pair must be located. The abstraction will occur somewhere further up the spine within an instance of f. However, there might be multiple f-instances on the evaluation stack, but we want only the one that corresponds to the appropriate abstraction.

To determine the correct scope of an instance t it suffices to identify the instance s that created t. We call s the *parent* of t. This corresponds to the edge from an argument node of s to its right-hand side in the term graph. This relationship is expressed by *parent edges* in the evaluation stack that connect each function instance with another instance further right in the stack. So besides the reference to the function definition it represents, a function instance also maintains a pointer to its parent. How parent

pointers are established is covered later. An instance of a function f with a parent edge to the instance at stack address a is denoted by f^a.

$$I ::= F^A \qquad \textit{(function instance)}$$

If the argument $\mathtt{P}_i^f$ occurs in a function g, then for each instance of g, the corresponding instance of f is connected by a chain of one or more parent edges.[3] When an argument of this form is encountered, the status register is set to $S = \mathtt{P}_i^f$, indicating a *parameter request*. Thereby the search for the abstraction is conducted by following parent edges, which we call *backtracing*. Backtracing is completed once the dynamic pivot (an instance of f) is located.[4] The sought-after abstraction is the ith parameter of the located function instance:

$$\begin{array}{lll}
\text{Parameter-Request: } (\mathtt{P}_i^f, a, \ldots g_a^p \ldots) & & \\
\rightarrow (\mathtt{P}_i^f, p, \ldots g_a^p \ldots) & f \neq g & \textit{(Backtrace)} \\
\rightarrow (\mathtt{A}_i, a-1, \ldots g_a^p \ldots) & f = g & \textit{(Request argument)}
\end{array}$$

4.5 Locating the application

The application node that matches this abstraction is further up the spine, and in the majority of cases (i.e. when the function application is perfectly saturated) within the function instance just to the right of f, called f's *predecessor*.[5] This is where the search for the application node begins ($T = a-1$). Thereby $i-1$ abstractions (parameters of f) have already been skipped, therefore the next $i-1$ abstraction nodes that occur further up the spine cannot belong to the abstraction that is to be γ-reduced.

To locate the corresponding application node, the spine has to be walked upwards applying the parentheses-matching algorithm. $S = \mathtt{A}_i$ indicates that $i-1$ unmatched abstraction nodes have been passed while walking upwards. Thus the next $i-1$ application nodes must be skipped. Keeping in mind that each function f represents a sequence of $\mathit{arity}(f)$ abstractions followed by $|\mathit{args}(f)|$ applications, the algorithm is implemented as follows by the abstract machine:

$$\begin{array}{lll}
\text{Argument-Request: } (\mathtt{A}_i, a, \ldots f_a^- \ldots) & & \\
\rightarrow (\mathtt{A}_{i-|\mathit{args}(f)|+\mathit{arity}(f)}, a-1, \ldots f_a^- \ldots) & |\mathit{args}(f)| < i & \textit{(Skip)} \\
\rightarrow (\mathit{args}_i(f), a, \ldots f_a^- \ldots) & |\mathit{args}(f)| \geq i & \textit{(Serve)}
\end{array}$$

Once the matching application node is found its value $\mathit{args}_i(f)$ is to substitute the tip of the spine in the subsequent γ-reduction. We say it is *served* (put into the S for examination).

---

[3] This corresponds to static links and static chains in the call stack of the run-time system of imperative programming languages.

[4] Due to the scoping rules of functional languages it is always the first occurrence of an f-instance that binds the requested parameter.

[5] Accordingly in conventional execution models parameters of a perfectly saturated function call passed directly by the caller.

4.6 Very Lazy Evaluation

Once the hoc is identified and the corresponding abstraction/application pair is located, according to the definition of γ-reduction the term is to be rewritten in multiple positions: First, each occurrence of the substitution variable is replaced by the argument's right-hand side, then the abstraction and application nodes are discarded. However, not one of these operations are performed by the abstract machine, which at first may be surprising. Then again it is natural that modifications of individual nodes cannot easily be mapped to a representation of the term where function instances capture only its macro-structure and do not reproduce the internal structure of the function definitions.

Consequently the abstract machine retains the abstraction/application pair, which is semantically correct in terms of β-equivalency. This simplifies γ-reduction considerably, as the de-Bruijn indexes remain valid so no α-conversion is necessary. Here we do not discuss sharing, so we do not address multiple occurrences of substitution variables. Thus nothing but the hoc itself must be substituted, which coincides with what is defined as *head linear reduction* [Danos 2004].

But also the substitution of the hoc can be omitted, if it does not interfere with subsequent evaluation. Indeed the 0th argument of an instance of a function f is examined only once, directly after it is pushed on the stack. Also it is not counted in $|args(f)|$ so it has no impact on the parentheses-matching algorithm. Therefore the abstract machine leaves the hoc in place leaving all function instances unmodified.

There are two cases for the value of the application node to distinguish for further action. An argument may reference either a function or a parameter. Let us first assume the former, thus $S = f$. Then f is instantiated and pushed on the evaluation stack. Thereby the function instance containing the scrutinised application node (the current value of the T-register) is registered as the parent of the new function instance. Then S is set to $\mathtt{A}_0$ and T to the address of the newly created function instance, such that again the 0th argument of the leftmost function instance is examined for the next γ-reduction step.

$$\begin{array}{ll} \text{Instantiate: } (f, a, \ldots) & \\ \rightarrow (\mathtt{A}_0, n, f^a_n \ldots) & \textit{(Push Instance)} \end{array}$$

If the argument is a parameter ($S = \mathtt{P}^f_i$) according to γ-reduction, it would substitute the hoc by this value. But once again, no such substitution is performed by the abstract machine, which saves an α-conversion. Instead, without any intermediate rewriting the argument is treated directly as if it was the hoc, according to the inference rules for parameter handling specified above.

4.7 Wrapping it up

Based on the presented mechanisms a specification of the abstract machine can be given that implements the very lazy λ-calculus. The operational semantics (*Fig. 8*) is specified in a rather unconventional but quite intuitive manner. Note that variables with no relevance to a specific rule (*don't-cares*) are denoted as '–', similarly for sequences, denoted as '…'.

Summarising, some interesting characteristics of the abstract machine can be observed:

$$
\begin{array}{lll}
STE ::= (S,T,E) & & \textit{(configuration)} \\
S \quad ::= F \mid \mathtt{P}^F_{\mathbb{N}} \mid \mathtt{A}_{\mathbb{N}} & & \textit{(status register)} \\
T \quad ::= A & & \textit{(target register)} \\
E \quad ::= I_n\, I_{n-1} \ldots I_1 & & \textit{(evaluation stack)} \\
A \quad ::= \mathbb{N} & & \textit{(stack address)} \\
I \quad ::= F^A & & \textit{(function instance)} \\
F \quad ::= \mathbb{N} & & \textit{(function address)}
\end{array}
$$

Fig. 7. Configuration grammar

- Arguments are fetched at the latest moment possible in contrast to conventional execution models where arguments are passed by the caller as soon as they are available rather than as soon as they are required, which is a form of strictness in the argument handling. Therefore it is in fact justified to consider our model lazier.
- On the evaluation stack a function instance is always directly preceded by its caller. This relation is modeled without the help of pointers. That structure is exploited by the abstract machine when fetching arguments.
- There is no need to maintain a constantly updated environment. The evaluation stack can be thought of as an incremental definition of the environment.
- Interestingly, the sequence of instances on the evaluation stack directly encodes the path from the root of the fully expanded, unreduced term to the tip of its spine.
- The term is in QHNF either if the hoc is a free variable (such as a constructor), or if the term is functional such that for a selected abstraction no matching application is found. The latter case manifests itself in an argument request attempting to cross the right boundary of the evaluation stack.
- Very lazy evaluation is linear in many aspects such as the manner in which functions are defined, the linearity of the reduction strategy, and the run-time data structure (the evaluation stack). This is possibile due to the technique of using parent pointers and because of refraining from any rewriting on the spine.

Initial State: $(\mathtt{main}, \bot, \varepsilon)$

Instantiate: $(f, a, \ldots)$

$$
\begin{array}{lll}
\to (\mathtt{A}_0, n, f^a_n \ldots) & & \textit{(Push Instance)}
\end{array}
$$

Argument-Request: $(\mathtt{A}_i, a, \ldots f^-_a \ldots)$

$$
\begin{array}{lll}
\to (\mathtt{A}_{i-|args(f)|+arity(f)}, a-1, \ldots f^-_a \ldots) & |args(f)| < i & \textit{(Skip)} \\
\to (args_i(f), a, \ldots f^-_a \ldots) & |args(f)| \geq i & \textit{(Serve)}
\end{array}
$$

Parameter-Request: $(\mathtt{P}^f_i, a, \ldots g^p_a \ldots)$

$$
\begin{array}{lll}
\to (\mathtt{P}^f_i, p, \ldots g^p_a \ldots) & f \neq g & \textit{(Backtrace)} \\
\to (\mathtt{A}_i, a-1, \ldots g^p_a \ldots) & f = g & \textit{(Request argument)}
\end{array}
$$

Fig. 8. Operational semantics

4.8 Example evaluated

To depict the evaluation as performed by the STEC machine we regard the execution of the example program from *Fig. 6*. It was chosen to exemplify the operational semantics of the STEC machine rather than to reveal the advantages of head occurrence reduction.

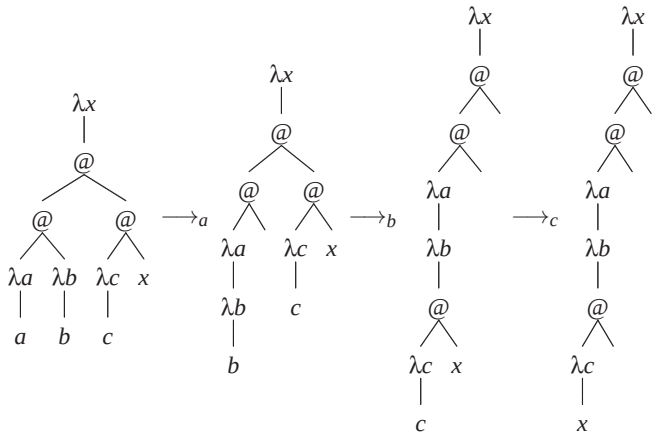


Fig. 9. Head linear reduction of the program graph of *Fig. 6*

To understand the abstract machine evaluation given below, it is helpful to identify each instance on the evaluation stack with the corresponding sequence of spine nodes in *Fig. 9*. Therefore the function definitions from *Fig. 6* need to be consulted. First we expand the term along the spine beginning from the root f to locate its hoc.

$$\begin{array}{ll} \text{Initial State:} & (f, \bot, \varepsilon) \\ \text{Push Instance:} \rightarrow & (\mathtt{A}_0, 1, f_1^{\bot}) \\ \text{Serve:} \rightarrow & (id, 1, f_1^{\bot}) \\ \text{Push Instance:} \rightarrow & (\mathtt{A}_0, 2, id_2^1\ f_1^{\bot}) \end{array}$$

The hoc is *a* (in *Fig. 9*). The corresponding argument belongs to *id*'s caller f.

$$\left.\begin{array}{ll} \text{Serve:} \rightarrow & (\mathtt{P}_1^{id}, 2, id_2^1\ f_1^{\bot}) \\ \text{Request argument:} \rightarrow & (\mathtt{A}_1, 1, id_2^1\ f_1^{\bot}) \\ \text{Serve:} \rightarrow & (id, 1, id_2^1\ f_1^{\bot}) \\ \text{Push Instance:} \rightarrow & (\mathtt{A}_0, 3, id_3^1\ id_2^1\ f_1^{\bot}) \end{array}\right\} a$$

For the next argument request in order to locate the appropriate application node, a function instance must to be skipped. In *Fig. 9* this corresponds to the abstraction node λa. The argument index is incremented by one such that the matching application node

(the one above λa) is also skipped:

$$\left.\begin{array}{lll}
\text{Serve:} & \to & (\mathtt{P}_1^{id}, 3, id_3^1\ id_2^1\ f_1^\perp) \\
\text{Request argument:} & \to & (\mathtt{A}_1, 2, id_3^1\ id_2^1\ f_1^\perp) \\
\text{Skip:} & \to & (\mathtt{A}_2, 1, id_3^1\ id_2^1\ f_1^\perp) \\
\text{Serve:} & \to & (idx, 1, id_3^1\ id_2^1\ f_1^\perp) \\
\text{Push Instance:} & \to & (\mathtt{A}_0, 4, idx_4^1\ id_3^1\ id_2^1\ f_1^\perp)
\end{array}\right\} b$$

The call of a known function *id* by *idx* is realised a spine expansion:

$$\begin{array}{lll}
\text{Serve:} & \to & (id, 4, idx_4^1\ id_3^1\ id_2^1\ f_1^\perp) \\
\text{Push Instance:} & \to & (\mathtt{A}_0, 5, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp)
\end{array}$$

Here it can be seen that some γ-reductions may not even require an update of the evaluation stack.

$$\left.\begin{array}{lll}
\text{Serve:} & \to & (\mathtt{P}_1^{id}, 5, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp) \\
\text{Request argument:} & \to & (\mathtt{A}_1, 4, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp)
\end{array}\right\} c$$

The request by *idx* for a parameter that was bound in a different function f requires a backtracing step to locate the abstraction that binds the current hoc.

$$\left.\begin{array}{lll}
\text{Serve:} & \to & (\mathtt{P}_1^{f}, 4, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp) \\
\text{Backtrace:} & \to & (\mathtt{P}_1^{f}, 1, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp) \\
\text{Request argument:} & \to & (\mathtt{A}_1, \perp, id_5^4\ idx_4^1\ id_3^1\ id_2^1\ f_1^\perp)
\end{array}\right\} x$$

The evaluation terminates because a request attempts to cross the stack boundary. That means that no abstraction/application pair could be located within the spine, thus the term is in QHNF.

4.9 Case-Discrimination and Primitives

To implement functional programming languages, two more issues need attention: case discrimination and primitives operators. They cannot be modeled by means of the pure λ-calculus, which has to be enriched for that purpose. Here, this semantic extension is only realised on the abstract machine level, not as yet another λ-calculus variant.

$$\begin{array}{lll}
\textit{program-definition} & ::= & \textit{function-definition}^+ \\
\textit{function-definition} & ::= & \textit{function-id}_{\textit{arity}}\ \textit{argument}^+\ \textit{alternative}^* \\
\textit{arity} & ::= & \mathbb{N}^0 \\
\textit{argument} & ::= & \textit{function-id} \mid \mathtt{P}_{\mathbb{N}^+}^{\textit{function-id}} \mid \mathtt{O}_{\mathbb{N}} \mid \textit{constant} \\
\textit{alternative} & ::= & \textit{integer function-id} \mid \mathtt{default}\ \textit{function-id} \\
\textit{constant} & ::= & \textit{integer} \mid \textit{float} \mid \ldots
\end{array}$$

Fig. 10. Enriched abstract syntax of the STEC machine language

In the enriched abstract machine language, a case discrimination is specified by attaching a non-empty, integer-indexed list of alternatives $alts(f)$ to a function definition f, its right-hand side $args(f)$ being the scrutinee. Constructors are mapped to integers at compile-time unambigously within the constructor set of one data type. Constructor parameters can be accessed as the function parameters of the alternatives' right hand-side. No further measures are necessary to model constructors, as they are adequately handled by the argument request mechanism. A primitive operator ($\mathtt{O}_o$) addresses a platform-specific functionality that is identified by a unique numeric identifier o.

The operational semantics needs to account for the strictness that these language constructs imply. The scrutinee of a case discrimination reveals its constructor only in QHNF. Thus, to select the correct case alternative, a *continuation*-mechanism is required to return to the case discrimination once the scrutinee is evaluated. Likewise, primitive operators are generally strict in all of their arguments, so after the evaluation of each argument, the evaluation must return to its call site, either to evaluate the next argument, or if it is saturated to apply the operator.

Strict evaluation may nest, for instance, if the scrutinee of a case discrimination involves a further case discrimination. Therefore continuations are also maintained in a stack, the *continuation stack* (C), thus we extend the abstract machine configuration to:

$$STEC ::= (S,T,E,C) \qquad \textit{(configuration)}$$

The continuation stack holds two types of tokens: case continuation tokens and operator tokens, both of which specify the stack address that the continuation returns to. Additionally an operator token needs to define the operator it represents as well as a list of previously evaluated operands.

$$\begin{array}{lr}
C ::= K^* & \textit{(continuation stack)} \\
K ::= \mathtt{C}^A \mid \mathtt{O}^A_O[V^*] & \textit{(continuation)} \\
O ::= \mathbb{N} & \textit{(operator)} \\
V ::= integer \mid float \mid \ldots & \textit{(constant value)}
\end{array}$$

Continuations are pushed on the continuation stack when an operator or a function that defines a case discrimination is served. In the latter case the evaluation (besides pushing the continuation) proceeds as before by evaluating its right-hand side (the scrutinee). If an operator is served, its first operand is requested.

$$\begin{array}{lll}
\text{Instantiate: } (f,a,\ldots) & & \\
\rightarrow (\mathtt{A}_0,n,f^a_n\ldots,\ldots) & |alts(f)| = 0 & \textit{(Push Instance)} \\
\rightarrow (\mathtt{A}_0,n,f^a_n\ldots,\mathtt{C}^n\ldots) & |alts(f)| > 0 & \textit{(Scrutinise)}
\end{array}$$

$$\begin{array}{lr}
\text{Operator: } (\mathtt{O}_{op},-,t_n\ldots,\ldots) & \\
\rightarrow (\mathtt{A}_1,n,t_n\ldots,\mathtt{O}^n_{op}[]\ldots) & \textit{(First Operand)}
\end{array}$$

As soon as the subsequent computation yields a constant value c, indicated by $S = c$, the continuation on the top of the stack is examined. For a case continuation, the correct alternative is selected and served. For operator continuations, before applying the operator it must first be checked whether more arguments are required. Only when

sufficient operands have been acquired, the operator is applied and the result of the primitive operation is propagated. This semantics is expressed in the last two groups of *Fig. 12*.

$$
\begin{array}{lll}
STEC & ::= (S,T,E,C) & \textit{(configuration)} \\
S & ::= F \mid \mathtt{P}^{F}_{\mathbb{N}} \mid \mathtt{A}_{\mathbb{N}} \mid \mathtt{O}_{O} \mid V & \textit{(status register)} \\
T & ::= A & \textit{(target register)} \\
A & ::= \mathbb{N} & \textit{(stack address)} \\
E & ::= I_n\, I_{n-1} \ldots I_1 & \textit{(evaluation stack)} \\
C & ::= K^* & \textit{(continuation stack)} \\
I & ::= F^{A} & \textit{(function instance)} \\
F & ::= \mathbb{N} & \textit{(function address)} \\
K & ::= \mathtt{C}^{A} \mid \mathtt{O}^{A}_{O}[V^*] & \textit{(continuation)} \\
O & ::= \mathbb{N} & \textit{(operator)} \\
V & ::= \mathit{integer} \mid \mathit{float} \mid \ldots & \textit{(constant value)}
\end{array}
$$

Fig. 11. Enriched configuration grammar

In this work we derived from the very lazy λ-calculus the STEC-machine, which is a concretisation of the PAM enriched by strict semantics to support case discriminations and operators. In [Danos 2004] different concepts of head linear reduction were mixed up in one definition. Here we clearly distinct between generalising β-reduction, defining a reduction strategy, and giving a concrete implementation that avoids rewriting. Furthermore we distinguish between program compilation and execution.

5 Perspectives

Even though the PAM has already been discovered years ago, it has not yet been investigated extensively. However, there is ample opportunity for further research, in particular it still remains a challenge to find efficient mechanisms for sharing as well as for garbage collection that take advantage of the abstract machine's prominent features.

While it is difficult to reason about the performance of the abstract machine compared to existing functional language implementations without taking these issues into account, there are aspects about our approach that hold much potential in this regard. Aside from the reduced amount of rewriting steps that are required by the very lazy λ-calculus, it is primarily the lean memory profile of the STEC-machine that is promising. The run-time data structures are compact, since per function instance only two pointers need to be allocated.[6] This results in a smaller memory footprint compared to conventional graph reduction models, which in each closure also maintain a set of parameters. Furthermore it is noticeable that no pointer updates are necessary resulting in very few write accesses. While partly compensated by additional read accesses (because of the need to locate abstraction/application pairs) still the advantage seems to

[6] A potential optimisation would be to allow variably-sized function instances, i.e. instances without a parent pointer for functions without free parameter variables.

Initial State: $(\texttt{main}, \bot, \varepsilon, \varepsilon)$

Instantiate: $(f, a, ..., ...)$		
$\rightarrow (\texttt{A}_0, n, f^a_n ..., ...)$	$\lvert alts(f)\rvert = 0$	*(Push Instance)*
$\rightarrow (\texttt{A}_0, n, f^a_n ..., \texttt{C}^n ...)$	$\lvert alts(f)\rvert > 0$	*(Scrutinise)*
Argument-Request: $(\texttt{A}_i, a, ... f^-_a ..., ...)$		
$\rightarrow (\texttt{A}_{i-\lvert args(f)\rvert + arity(f)}, a-1, ... f^-_a ..., ...)$	$\lvert args(f)\rvert < i$	*(Skip)*
$\rightarrow (args_i(f), a, ... f^-_a ..., ...)$	$\lvert args(f)\rvert \geq i$	*(Serve)*
Parameter-Request: $(\texttt{P}^f_i, a, ... g^p_a ..., ...)$		
$\rightarrow (\texttt{P}^f_i, p, ... g^p_a ..., ...)$	$f \neq g$	*(Backtrace)*
$\rightarrow (\texttt{A}_i, a-1, ... g^p_a ..., ...)$	$f = g$	*(Request argument)*
Operator: $(\texttt{O}_{op}, -, t_n ..., ...)$		
$\rightarrow (\texttt{A}_1, n, t_n ..., \texttt{O}^n_{op}[] ...)$		*(First Operand)*
Operand: $(v, -, ..., \texttt{O}^a_{op}[v_1, ..., v_c] ...)$		
$\rightarrow (apply_{op}(v_1, ..., v_c, v), -, ..., ...)$	$arity(op) = c+1$	*(Apply Operator)*
$\rightarrow (\texttt{A}_{c+2}, a, ..., \texttt{O}^a_{op}[v_1, ..., v_c, v] ...)$	$arity(op) > c+1$	*(Next Operand)*
Scrutinee: $(c, -, ... f^p_a ..., \texttt{C}^a ...)$		
$\rightarrow (alts_c(f), a, ... f^p_a ..., ...)$		*(Serve alternative)*

Fig. 12. Enriched operational semantics

predominate. This presumption however is yet to be validated in a comparison with a well-established execution model like the STG-machine [Peyton Jones 1992].

Implementations based on super-combinators usually compile the abstract-machine code into machine code of the target architecture that integrates the semantics of the abstract machine and therefore can be directly executed by a concrete machine. Due to the simplicity of the STEC-machine, a different compilation model, one that separates the abstract machine and the function definitions, seems to be adequate. The operational semantics can be implemented in a very small piece of executable machine code. Each function definition can be stored in a compact array as read-only data. Access to individual arguments of a function definition (as frequently performed by the STEC-machine) can be accomplished efficiently using an array lookup if a uniformly-sized representation for the arguments is chosen. This would hardly be the case when compiling the function definitions combined with the operational semantics to machine code, which would also lead to a considerable increase of the memory footprint.

Since the evaluation stack only grows, a garbage-collection mechanism is required to release memory occupied by function instances that are no longer required. In that sense the evaluation stack in fact is a heap. However, it would be short-sighted to neglect the fact that it is highly structured in comparison to a usual heap in which data is organised as memory blocks at arbitrary positions that refer to each other. Much is to be expected by a sophisticated garbage-collection mechanism that systematically exploits this structure for increased efficiency. Since the evaluation stack is an incremental definition of the environment, this would effectively be realised as a (linear) compaction of the evaluation stack.

Obviously this linearity cannot be sustained once sharing is introduced to the model, as sharing in a sense implies a non-linear structure. Still, the linearity of evaluation might offer new possibilities for integrating sharing-techniques that achieve a higher degree of sharing than full laziness by breaking the linear structure only at few, well-defined points. In particular the subject of optimal evaluation in the sense of [Lévy 1978] should be investigated in the light of very lazy evaluation.

Summarising, there is still much opportunity for completion and optimisation of the STEC-machine in order to obtain a new type of practical high-performance functional language implementation. In particular it is an interesting question which of the optimisations used by today's compilers can be applied to the STEC-machine and what new kind of possibilities for optimisations are opened up by the model.

References

Fairbairn 1987. JON FAIRBAIRN, STUART WRAY, *Tim: A simple, lazy abstract machine to execute supercombinators*. *Functional Programming Languages and Computer Architecture*, Springer, 1987, pp 34-45. *Lecture Notes in Computer Science*, Volume 274/1987.

Krivine 2007. JEAN-LOUIS KRIVINE, *A call-By-name lambda-calculus machine*. *Higher Order and Symbolic Computation*, Kluwer Academic Publishers, September 2007, pp 199-207. Volume 20, Issue 3.

Peyton Jones 1987. SIMON L. PEYTON JONES, PHILIP WADLER, PETER HANCOCK, DAVID TURNER, *The implementation of functional programming languages*, Prentice Hall International, 1987.

Burn 1988. GEOFFREY L. BURN, SIMON L. PEYTON JONES, JOHN D. ROBSON, *The spineless G-machine*. *Proceedings of the 1988 ACM conference on LISP and functional programming*, ACM, 1988, pp 244-258.

Peyton Jones 1992. SIMON L. PEYTON JONES, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5*, Department of Computing Science, University of Glasgow, July 9, 1992.

Leijen 2005. DAAN LEIJEN, *The lazy virtual machine specification*, Institute of Information and Computing Sciences, Utrecht University, August 22, 2005.

Holyer 1998. IAN HOLYER, ELENI SPILIOPOULOU, *The Brisk Machine: a simplified STG machine*, University of Bristol, Department of Computer Science, March 1998.

Barendregt 1984. H. P. BARENDREGT, *The Lambda Calculus: Its syntax and semantics*, 1984.

Danos 2004. VINCENT DANOS, LAURENT REGNIER, *Head linear reduction*, unpublished, http://iml.univ-mrs.fr/~regnier/articles.html, June 7 2004.

Danos 1996. V. DANOS, H. HERBELIN, L. REGNIER, *Game semantics and abstract machines*. *Symposium on Logic in Computer Science*, IEEE Computer Society, September 2, 1996, p 394 ff.

De Bruijn 1972. NICOLAAS GOVERT DE BRUIJN, *Lambda Calculus Notation with Nameless Dummies – a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*. *Indagationes Mathematicae*, 1972, pp 381-392.

Hughes 1982. R. J. M. HUGHES, *Super-Combinators – a new implementation method for applicative languages*. *Proceedings of the 1982 ACM symposium on LISP and functional programming*, ACM, 1982, pp 1-10.

Kamareddine 2001. FAIROUZ KAMAREDDINE, ROEL BLOO, ROB NEDERPELT, *De Bruijn's syntax and reductional equivalence of λ-terms*. *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, ACM, 2001, pp 16-27.

Danvy 2000. OLIVIER DANVY, ULRIK P. SCHULTZ, *Lambda-dropping: transforming recursive equations into programs with block structure*. Elsevier Science Publishers, 2000, pp 243-287. *Partial evaluation and semantics-based program manipulation*, Volume 248, Issue 1-2 (October 2000).

Marlow 2006. SIMON MARLOW, SIMON PEYTON JONES, *Making a fast curry: push/enter vs. eval/apply for higher-order languages*. *Journal of Functional Programming*, Cambridge University Press, August 10 2006, pp 415-449. Volume 16 2006.

Sestoft 2002. PETER SESTOFT, *Demonstrating lambda calculus reduction*. *The Essence of Computation: Complexity, Analysis, Transformation – Essays Dedicated to Neil D. Jones*, T. MOGENSEN, D. SCHMIDT, I. H. SUDBUROUGH (eds.), Springer-Verlag, 2002, pp 420-435. *Lecture Notes in Computer Science*, 2566.

Lévy 1978. JEAN-JACQUES LÉVY, *Optimal reductions in the lambda-calculus*. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. SELDIN, J. R. HINDLEY (eds.), Academic Press, 1978.