A Simple and Efficient Implementation of Strong Call by Need by an Abstract Machine

MAŁGORZATA BIERNACKA, University of Wrocław, Faculty of Mathematics and Computer Science, Poland
WITOLD CHARATONIK, University of Wrocław, Faculty of Mathematics and Computer Science, Poland
TOMASZ DRAB, University of Wrocław, Faculty of Mathematics and Computer Science, Poland

We present an abstract machine for a strong call-by-need strategy in the lambda calculus. The machine has been derived automatically from a higher-order evaluator that uses the technique of memothunks to implement laziness. The derivation has been done with the use of an off-the-shelf transformation tool implementing the “functional correspondence” between higher-order interpreters and abstract machines, and it yields a simple and concise description of the machine. We prove that the resulting machine conservatively extends the lazy version of Krivine machine for the weak call-by-need strategy, and that it simulates the normal-order strategy in bilinear number of steps.

CCS Concepts: • Theory of computation → Semantics and reasoning.

Additional Key Words and Phrases: λ-calculus, Abstract machines, Computational complexity, Reduction strategies, Normalization by evaluation

ACM Reference Format:
Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022. A Simple and Efficient Implementation of Strong Call by Need by an Abstract Machine. Proc. ACM Program. Lang. 6, ICFP, Article 94 (August 2022), 28 pages. https://doi.org/10.1145/3549822

1 INTRODUCTION

Lambda calculus is a well-known model of functional programming languages, whose practical implementations rely on a deterministic reduction strategy (also known as evaluation strategy) that restricts the general β-reduction rule in the locations where it can be applied inside a term. Popular incarnations of the lambda calculus are full-blown programming languages OCaml and Haskell, each of which employs a different evaluation strategy: weak call by value (CbV) and weak call by need (CbNd), respectively.

A strategy is called weak if it does not descend and reduce under lambda abstractions; hence, any lambda abstraction is considered to be a value. The most popular for practical programming is the weak CbV strategy, where each argument is evaluated exactly once, before the application of the function, even though the computed value may not be used in the subsequent computation. On the other hand, weak call by name (CbN) is a strategy that performs β-reduction without evaluating the argument first. In consequence, the argument may be evaluated several times if it is needed more

∗This research is supported by the National Science Centre of Poland, under grant number 2019/33/B/ST6/00289.

Authors’ addresses: Małgorzata Biernacka, mabi@cs.uni.wroc.pl, University of Wrocław, Faculty of Mathematics and Computer Science, Wrocław, Poland; Witold Charatonik, wch@cs.uni.wroc.pl, University of Wrocław, Faculty of Mathematics and Computer Science, Wrocław, Poland; Tomasz Drab, tdr@cs.uni.wroc.pl, University of Wrocław, Faculty of Mathematics and Computer Science, Wrocław, Poland.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2022 Copyright held by the owner/author(s).
2475-1421/2022/8-ART94
https://doi.org/10.1145/3549822

Proc. ACM Program. Lang., Vol. 6, No. ICFP, Article 94. Publication date: August 2022.
than once in the function body (i.e., if the variable for which it will be substituted occurs several times in the function body, and in positions that will not be eliminated by $\beta$-reduction). As a way of preventing the latter inefficiency of CbN, practical implementations (such as Haskell) optimize the strategy and evaluate arguments by need ( lazily). In the weak call-by-need strategy, function arguments are not evaluated beforehand, but whenever an argument needs to be evaluated, its computed value is memoized and fetched for the next time it is needed, without recomputation.

In some applications weak strategies turn out to be insufficient. The development of proof assistants based on dependent type theory (such as Coq or Agda) prompted further studies of strong reduction in the lambda calculus and its practical implementations. Strong reduction computes full normal forms of terms (required in proof assistants to perform type-checking), therefore it must reduce bodies of lambda-abstractions and deal with open terms. A natural way to obtain a strong strategy from a weak one is to iterate it after weakly reducing the term to a value: continue reducing inside the function body and in argument positions of rigid terms, i.e., terms of the form $x_1 t_1 \ldots t_n$, where it is certain that the variable will never be substituted and will not create a new $\beta$-redex. Strong strategies defined in this way are conservative over the respective weak strategies.

Since the direct motivation for studying strong strategies comes from the practical side, efficient implementations in the form of abstract machines have been sought for them. The first such machines were designed by Crégut: a strongly reducing extension KN of the Krivine machine that performs normal-order reduction, and its lazy variant KNL [Crégut 2007] that introduces a form of sharing of head normal forms, but does not perform full normalization and thus does not implement a strong strategy in our sense. It has been later shown how the first of these machines can be deconstructed into a normalization-by-evaluation function with call-by-name evaluation [Biernacka et al. 2020; Munk 2007]. The more recent studies include the line of work followed by Accattoli et al., who develop strong strategies in calculi based on the Linear Substitution Calculus and construct abstract machines with the goal of giving a formal account of their complexity [Accattoli and Barras 2017; Accattoli and Coen 2015; Accattoli and Guerrieri 2019], and a derivational approach initiated by Danvy et al. [Ager et al. 2003, 2004; Piróg and Biernacki 2010] and later extended to strong strategies, that connects higher-level semantic formats with abstract machines in a systematic way [Biernacka et al. 2020; Biernacka and Charatonik 2019; Biernacka et al. 2017].

The canonical implementation of the weak call-by-need strategy is the STG machine underlying Haskell [Jones 1992], which features a store used to memoize computed values. It has been shown that the STG machine can be systematically derived from a natural semantics for the STG language [Piróg and Biernacki 2010]. This is in line with a store-based approach to defining formal semantics of lazy languages due to Launchbury [Launchbury 1993; Sestoft 1997]. An alternative approach is purely syntactic and does not use explicit store, as proposed by Ariola et al. [Ariola et al. 1995]. In the latter vein are the recent studies of the Strong CbNd strategy [Balabonski et al. 2017, 2021; Barenbaum et al. 2018; Biernacka and Charatonik 2019]. Balabonski et al. [Balabonski et al. 2017] first proposed a form of reduction semantics for Strong CbNd, using a language with explicit substitutions. Quite recently, they further refined their approach and proposed a strong call-by-need calculus that admits shorter normalizing reduction sequences than the known calculus [Balabonski et al. 2021]. The semantics of [Balabonski et al. 2017] has been operationalized into a refocusable reduction semantics, and an abstract machine derived from it by Biernacka and Charatonik [Biernacka and Charatonik 2019]. The resulting abstract machine can be shown to precisely implement the corresponding strategy. The semantics and the machine operate on a language equipped with (explicit substitutions or) let-constructors to facilitate sharing of values needed to implement laziness. However, as such, it is not directly usable as an efficient implementation; it would have to be optimized and appropriate data structures introduced so that each transition is performed in constant time. It is also quite complex – it consists of 24 transition rules. On the
other hand, a lazy and practically efficient machine for strong reduction is implemented by Coq. However, it is necessarily an even more complex artefact that has not been formally studied and its underlying strategy has not been distilled and identified.

A reference point in terms of efficiency is Accattoli’s Useful MAM machine [Accattoli 2016] (which is also quite simple, consisting of 10 transition rules) that has been proved to be reasonable; specifically, it simulates the normal-order (strong call-by-name) strategy in the number of steps quadratic in the number of $\beta$-steps and linear in the size of the initial term. Just recently, a better, bilinear bound has been established for strong call by value as implemented by the machine named SCAM of Accattoli et al [Accattoli et al. 2021]. It consists of only 9 transition rules, but requires precompilation of terms. However, no reasonable (and thus efficient) abstract machines have been proposed for the strong call-by-need strategy so far. The problem seemed to be harder than for other strategies not only because of the strength of the strategy but also due to complex sharing behaviour that is required in call by need [Accattoli and Leberle 2022].

In this work we show the first efficient abstract machine for Strong CbNd, named RKNL. We show how it has been derived in a mechanical way using functional programming techniques, and we give its formal account. Our goal is twofold: (a) to provide a simple and concise definition of an abstract machine for Strong CbNd, and (b) to facilitate reasoning about the key properties of the machine and its underlying strategy, including its correctness and complexity. In order to achieve these goals, we use the derivational approach to construction of abstract machines and specifically we adapt the methodology used previously by Biernacka et al. to derive an abstract machine for Strong CbV [Biernacka et al. 2021]. The starting point is a standard, higher-order normalization-by-evaluation function that normalizes terms to full normal form and that is known to correspond to normal-order reduction strategy [Biernacka et al. 2020]. We then modify it using the standard technique of memoization to avoid recomputation of intermediate values. Next, we transform this optimized normalizer by means of functional correspondence implemented recently by Buszka and Biernacki in a Racket tool [Buszka and Biernacki 2021]. The result is a concise description of the abstract machine for Strong CbNd. We prove that the machine correctly simulates normal-order reduction, and as a measure of its efficiency we give a simple proof that it is reasonable in the sense of Accattoli et al. [Accattoli et al. 2021; Accattoli and Dal Lago 2016; Accattoli and Guerrieri 2019]. The development reported in this work showcases a versatile and principled approach to obtaining abstract machines for complex languages with a well-understood high-level semantics, from which a model of implementation can be obtained mechanically. In order to reason about the complexity of the machine, we apply the method akin to amortized cost analysis that uses a potential function [Okasaki 1999].

The comparison of RKNL with existing machines and calculi for Strong CbNd is not straightforward, and could be further investigated (cf. Figure 13 in [Danvy and Zerny 2013]). In short, we can state the following main differences with respect to RKNL: Crégut’s KN from [Crégut 2007] performs normal-order reduction, but is not call-by-need and suffers from the exponential overhead; Crégut’s KNL from [Crégut 2007] is strong and call-by-need, but performs only head reduction instead of full reduction, Balabonski et al.’s [Balabonski et al. 2021] gives a calculus and a strategy that performs optimization similar to KNL, but is not an abstract machine; Biernacka & Charatonik’s machine from [Biernacka and Charatonik 2019] implements earlier Balabonski et al.’s calculus from [Balabonski et al. 2017], and also has exponential overhead; Coq machine is a quite complex artifact and its computational complexity is still being studied (cf. introduction of Accattoli & Barras’ [Accattoli and Barras 2017]).
The specific contributions of this work are as follows:

1. a novel abstract machine for Strong CbNd, with a simple (only 11 transition rules) and clear design, derived automatically with a semantic transformer [Buszka and Biernacki 2021] from an NbE normalizer,

2. a proof that the machine is reasonable w.r.t. normal order, i.e., that it simulates normal order in the number of steps polynomial – in this case bilinear – in the number of $\beta$-steps and in the size of the initial term,

3. a proof that the machine conservatively extends a weak call-by-need machine,

4. an improvement of the overall simulation overhead of strong call by need by the only machine for such strategy [Biernacka and Charatonik 2019] from exponential (unreasonable) to polynomial (reasonable),

5. an improvement of the overall simulation overhead of normal order by Useful MAM [Accattoli 2016] from quadratic [Accattoli et al. 2021] to quasibilinear in the RAM model.

2 PRELIMINARIES

Terms $t$ in the lambda calculus are defined with the following grammar:

$$ t ::= x \mid t_1 \cdot t_2 \mid \lambda x. t $$

where $x$ ranges over an infinite, countable set of identifiers. We define free and bound variables in a term in the usual way, and we consider terms up to renaming of bound variables. Sometimes we explicitly want to note that two terms $t_1$ and $t_2$ are $\alpha$-equivalent, and we then write $t_1 =_\alpha t_2$.

Contexts can be seen as terms with exactly one “hole” (denoted $\Box$) which can occur in any position within a term:

$$ C ::= \Box \mid Ct \mid tC \mid \lambda x.C $$

Given a context $C$ and a term $t$ to plug in its hole, we can reconstruct the intended term (denoted $C[t]$) by defining the plugging function as follows:

$$ \Box[s] = s \quad (C t)[s] = C[s] t \quad (t C)[s] = t C[s] \quad (\lambda x.C)[s] = \lambda x.C[s] $$

2.1 Reduction Semantics in the Lambda Calculus

Computation in the lambda calculus consists in performing $\beta$-contraction

$$ (\lambda x.t_1) t_2 \rightarrow_\beta t_1[x := t_2] $$

in some positions within the term (here $t_1[x := t_2]$ denotes the usual, capture-avoiding substitution of $t_2$ for $x$ in $t_1$). In order to capture formally the reduction relation on terms, we can use an explicit representation of these positions using contexts. Thus,

$$ C[(\lambda x.t_1) t_2] \rightarrow_\beta C[t_1[x := t_2]] $$

defines one step of the full, nondeterministic reduction relation in the lambda calculus in a succinct way. This semantic format is called reduction semantics [Felleisen and Hieb 1992].

Example 2.1. A term $I I (I I)$, where $I := \lambda x.x$, can be reduced in one step by full $\beta$-reduction in two ways: $I (I I) \leftarrow_\beta I I (I I) \rightarrow_\beta I I I I$ because both subterms $I I$ are contractible: $I I \rightarrow_\beta I$, and both contexts $\Box (I I)$ and $I I \Box$ are correctly derived from nonterminal $C$ w.r.t. to the given grammar. At the same time, the whole term is not contractible: $I I (I I) \not\rightarrow_\beta I (I I)$ because it is not an application of an abstraction.

In the following, the reflexive-transitive closure of any one-step reduction relation $\rightarrow$ is denoted by $\Rightarrow$ (possibly with some decorations on arrows) and the reflexive-symmetric-transitive closure.
is denoted by $=_{\alpha}$, and is called conversion. Juxtaposition of two relations denotes their composition, e.g., $s \rightarrow_{\beta} =_{\alpha} t$ means that $\exists t'. s \rightarrow_{\beta} t' =_{\alpha} t$.

Example 2.2. The term $II(II)$ reduces (in three steps) to identity: $II(II) \rightarrow_{\beta} I$ because $II(II) \rightarrow_{\beta} III \rightarrow_{\beta} II \rightarrow_{\beta} I$. The first reduct $II(I)$ does not reduce to the alternative reduct $I(I)$, i.e., $III \not\rightarrow_{\beta} I(I)$, but they are $\beta$-convertible: $III =_{\beta} I(I)$ because there exists a path of $\beta$-reductions and $\beta$-expansions between them: $III \leftarrow_{\beta} II(II) \rightarrow_{\beta} I(I)$.

If we want to impose a specific, deterministic reduction strategy, we can write a more precise grammar of reduction contexts which narrows down the positions where a computation step is allowed. For example, a weak-head reduction that applies $\beta$-contraction only in leftmost-outermost positions (also known as the call-by-name strategy) can be specified by the following definition of contexts:

$$E ::= \Box \mid Et$$

and the corresponding reduction relation is as follows:

$$E[(\lambda x.t_1) t_2] \overset{cbn}{\rightarrow} E[t_1[x:=t_2]]$$

Example 2.3. The term $II(II)$ reduces in one step in call by name to $I(I)$, i.e., $II(II) \overset{cbn}{\rightarrow} I(I)$ because the context $\Box(I)$ is a call-by-name context, i.e., it is derived from the nonterminal $E$, while the context $II\Box$ is not because it cannot be derived from $E$ and thus $II(II)$ does not reduce in call by name to $III$, i.e., $II(II) \not\overset{cbn}{\rightarrow} III$.

The generalization of call-by-name strategy to strong reduction does not stop on an abstraction, but instead iterates the same strategy inside its body and in arguments to neutral terms. It can be defined with contexts $N$ as follows:

$$NO \ni N ::= \overline{N} \mid \lambda x.N$$

$$\overline{N} ::= \Box \mid \overline{N} t \mid aN$$

where $a$ stands for neutral terms and $n$ are normal forms:

$$a ::= an \mid x$$

$$n ::= \lambda x.n \mid a$$

The definition of normal terms with the grammar (the nonterminal $n$) coincides with the definition of normal terms as terms that have no contractible subexpressions, i.e., normal forms of $\beta$-reduction, i.e., terms $t$ such that $t \not\rightarrow_{\beta}$. The identity $I$ is a normal term, while $II$ is not.

Example 2.4. Similarly, $\lambda y.I$ is a normal term, while $\lambda y.II$ is not. Moreover, $y I$ is a neutral term, while $y (II)$ is not. Every neutral term is a normal term that is not an abstraction.

This strong strategy is known as the normal-order strategy, and it computes full normal forms $n$ of lambda terms. We define one step of normal-order reduction as follows:

$$N[(\lambda x.t_1) t_2] \overset{no}{\rightarrow} N[t_1[x:=t_2]]$$

The grammar of contexts $N$ as defined above builds them from the outside in. It is often convenient to build contexts inside-out and think about them as stacks of single frames, where the top of the stack is the frame surrounding the hole:
This inside-out grammar makes it explicit how contexts are constructed and deconstructed in abstract machines (and we directly use it in Lemma 5.5 later).

Example 2.5. The term \( I \) can be reduced in call by name: \( I \xrightarrow{cbn} I \) because the empty context \( \square \) is a valid call-by-name context. It can also be reduced in normal order: \( I \xrightarrow{no} I \) because every call-by-name context is a normal-order context. However, neither \( \lambda y. I \) nor \( y(I) \) can be reduced in call by name: \( \lambda y. I \xrightarrow{cbn} \) and \( y(I) \xrightarrow{cbn} \) because \( \lambda y. \square \) and \( y \square \) are not call-by-name contexts. Nonetheless, these contexts are normal-order contexts (derivable from the nonterminal \( N \)) so both terms are normalized by normal order: \( \lambda y. I \xrightarrow{no} \lambda y. I \xrightarrow{\beta} \) and \( y(I) \xrightarrow{no} y I \xrightarrow{\beta} \).

The grammar of normal-order contexts restricts them in a way that the reduction may take place only in the leftmost-outermost position. For example, in a term \( \lambda y. (\lambda z. I) I \), normal order contracts the subterm \( (\lambda z. I) I \) but cannot contract the subterm \( I I \), i.e., \( \lambda y. (\lambda z. I) I \xrightarrow{no} \lambda y. (\lambda z. I) I \xrightarrow{no} \lambda z. I I \) because \( \lambda y. (\lambda z. \square) I = \square[\lambda y. \square][\square I][\lambda z. \square] \) is not a normal-order context. Contexts derivable from the nonterminal \( N \) can be seen as non-applicative (cf. [Accattoli and Dal Lago 2016]) normal-order contexts, while derivable from \( N \) as normal-order contexts not starting with an abstraction.

García-Pérez and Nogueira have shown the regular grammar of contexts presented as a finite automaton [García-Pérez and Nogueira 2014]. Later it was shown how to use it to transform inside-out and outside-in grammars into each other [Biernacka et al. 2020]. Many other strategies defined via reduction semantics and their properties can be found in [Biernacka et al. 2022b].

2.2 The Call-by-need Strategy

The “lazy” variant of the normal-order strategy (or of the weak call-by-name strategy) needs to use some form of sharing. To our knowledge, it has not been expressed with a simple restriction of general contexts with the \( \beta \)-contraction without any extension of the syntax of lambda terms. It seems that it cannot be done easily in the pure \( \lambda \)-calculus because of the tension between the need to substitute the needy head variable and the need to postpone substitution of the remaining ones.

Therefore, in order to provide a reduction semantics for it, some syntax extension is often used. We can either use a language with explicit substitutions [Balabonski et al. 2017, 2021] (or let-constructs [Ariola et al. 1995]) to explicitly handle bindings of terms to variables, or simulate a store structure within the reduction rules [Biernacka and Danvy 2007]. Another known possibility is to use a nonstandard contraction rule with a complex grammar of context [Chang and Felleisen 2012]. In this paper, we use an operational semantics for the weak call-by-need strategy in the form of an abstract machine with explicit store from [Danvy and Zerny 2013], presented in Section 5.6.

3 HIGHER-ORDER NORMALIZERS

3.1 Normal-order Normalizer

In Listing 1 we show a Racket program that performs normal-order (strong call-by-name) normalization of closed lambda terms into full normal form, and is an instance of a normalization-by-evaluation algorithm. This program is a slightly modified version of an OCaml NbE normalizer reconstructed from Crégut’s abstract machine KN in [Biernacka et al. 2020], and similar to Filinski and Rohde’s normalizer [Filinski and Rohde 2005].
The idea of normalization by evaluation is to evaluate expressions of an object language (here, lambda terms) into a semantic domain of values in such a way that equivalent terms are mapped to the same value, and from the values it is possible to extract a syntactic representation of the normal form (of the equivalence class). Here, our metalanguage is Racket and the domain of values consists of neutral terms and meta-level functions corresponding to lambda abstractions. Intuitively, meta-level functions should take values and return values. However, Racket is an eager language, and in normal-order strategy we do not want to evaluate an argument before the evaluation of the caller function, so the evaluation of arguments must be delayed. Therefore, meta-level functions operate on thunks (i.e., delayed values) and return values, as in [Filinski and Rohde 2005].

The algorithm is based on three functions: eval that translates a lambda term to a semantic value and calls its evaluation; apply-value that says how to evaluate an application of a meta-level function \( v \) to a thunk \( w \); and reify that translates semantic values to lambda terms in normal form.

Names \(*\text{var}*, *\text{app}, *\text{lam} are just constructor of lambda terms. Environments are dictionaries that associate names of free variables with thunks. In Racket, dictionaries are accessed by hash-ref and updated by hash-set functions.

To increase the readability of the final result, semantic functions are coupled with the original variable name (\( x \) in line 21) by the constructor Abs and later used in the generation of fresh variables in function reify in line 8; since name generation is called inside reification, all bound variables in the final result are different. The \( \alpha \)-renaming in full reduction is obligatory because otherwise a term \((\lambda x.x)(\lambda y.\lambda y.y)\) would incorrectly reduce to \((\lambda y.\lambda y.y)\) (cf. possible capture in an example in the penultimate paragraph of Section 5.1).
To normalize a closed term it is enough to evaluate it in the empty environment (hasheq) and to reify its value – this is done in line 24.

### 3.2 An Optimized, Call-by-need Normalizer

We optimize the normalizer for normal-order by introducing the standard technique of memothunks to cache computed values and reuse them later, if needed. We also extend it to handle open terms. The resulting normalizer is shown in Listing 2.

We introduce two more constructors to explicitly indicate the state of a memothunk: todo stores plain thunks to be computed and done stores their already computed results. A memothunk consists of Racket’s built-in mutable memory cell called box storing one of these two. The function memothunk converts a plain thunk into a memothunk by putting it into todo and box. To access a memothunk we use the procedure force that just returns the result of a computation if it is already computed, or it runs the stored plain thunk, remembers its result in the box, and then returns the result.

There are several changes in the normalizer, let us start with less important ones. Recall that normalization consists of two phases: evaluation in the semantic domain followed by reification in the syntactic domain (i.e., in the object language). Therefore things that happen on the borderline between the two phases may be called either from eval or from reify function. In particular, generation of fresh names is moved from line 8 in Listing 1 to line 27 in Listing 2. In consequence, while in the normal-order normalizer a fresh name is generated for each reification of each abstraction, in the lazy normalizer reification of each abstraction generates at most one fresh name. However, this is fine because normal forms retrieved from the same memothunk do not overlap (cf. the two binding occurrences of the variable $z_0$ in step 27 in Table 2). Similarly, the whole code of reification from lines 8–9 in Listing 1 is moved as a thunk to lines 27–29 in Listing 2.

A more important change from the efficiency point of view is that the semantic function constructed in line 26 converts a plain thunk that it receives into a memothunk when it adds it into the environment. By creating a memothunk for the argument, we thus avoid its recomputation in the semantic domain. Moreover, in line 27 we create a memothunk for the second phase, which avoids recomputation of the reification in the syntactic domain. Now the Abs constructor carries two parameters: the memothunk used for computation of normal forms in the syntactic domain in line 14 and the semantic function used for evaluation in the semantic domain in line 19. The two types of memothunks are then forced in line 24 (for evaluation in the semantic domain), and in line 14 (for normalization in the syntactic domain).

There is a design choice in putting a memothunk in line 26 rather than in line 25, where a plain thunk is passed as an argument. Both would be correct, but the one presented is a bit more sparing. If the left term $t$ in line 25 evaluates to an Abs, the value of right term $u$ is put into memothunk immediately anyway. However, if the left term evaluates to a neutral term, we know that the right term’s value will be reified exactly once, so we can avoid memoization overhead.

To be able to handle open input terms, we also add a default value for hash-ref in line 24: when the variable $x$ does not occur in the environment $e$, a box with the value $x$ is returned.

### 3.3 Derivation of an Abstract Machine

We used a semantic transformation tool sent developed by Buszka and Biernacki [Buszka and Biernacki 2021] to automatically transform the initial evaluator for Strong CbNd into an abstract machine. In Listing 3 we present the input code for sent written in IDL (Interpreter Definition Language), which differs from the Racket evaluator slightly in syntax. Constructors of lambda terms are Var, App, and Lam. The input code contains a handful of annotations that guide the derivation process, and a “dummy” implementation of the functions gensym and force. The reason for this
A Simple and Efficient Implementation of Strong Call by Need by an Abstract Machine

List 2. A call-by-need normalizer

1 (struct todo (t))
2 (struct done (v) #:transparent)
3
4 (define (force mt)
5   (match (unbox mt)
6     [(done v) v]
7     [(todo t) (let ([v t]) (set-box! mt (done v) v))])
8
9 (define (memothunk thunk)
10   (box (todo thunk)))
11
12 (define (reify v)
13   (match v
14     [(Abs c _) (force c)]
15     [t (t)])
16
17 (define (apply-value v w)
18   (match v
19     [(Abs _ f) (f w)]
20     [t (*app t (reify (w)))]))
21
22 (define (eval e t)
23   (match t
24     [(*var x) (force (hash-ref e x (box (done (*var x)))))]
25     [(*app t u) (apply-value (eval e t) (λ () (eval e u)))]
26     [(lam x t) (let ([f (λ (v) (eval (hash-set e x (memothunk v) t)))]
27                 (Abs (memothunk (λ () (let ([x1 (gensym x)])
28                           (*lam x1 (reify
29                             (f (λ () (*var x1))))))))))]])
30
31 (define (normalize t)
32   (reify (eval (hasheq t))))

latter tweak is that the transformer does not handle stateful computation, i.e., it does not support mutable memory cells. In order to get the correct implementation of laziness, after we transform the evaluator into the machine, we then replace back the implementation of the force function with that of Listing 2, and we use the default fresh-name generating function.

The transformation tool implements a refinement of the known derivation methodology of Danvy et al.’s functional correspondence [Ager et al. 2003], and it performs the following steps:

1) translation to administrative normal form (ANF), where all intermediate results of computation are bound by let-constructors,

2) selective transformation into continuation-passing style (a.k.a. CPS transformation),

3) selective defunctionalization of higher-order function spaces into first-order data structures and the corresponding dispatch functions.

The paper [Ager et al. 2003] presents the methodology explained on several examples including Krivine and CEK machines.
The selective CPS transformation does not alter functions annotated with `#:atomic` which are treated as atomic, "trivial" operations. In our case these are the functions operating on environments, and the `gensym` function that generates fresh names. In the final stage of the transformation defunctionalization is performed on all higher-order functions except those that have been annotated with `#:non-defun`, which are left untouched. In our case, again, we do not touch the function `extend`. 

Listing 3. Input specification to `sem`
that operates on environments. The annotations #:name and #:apply (each with a single argument) are used to supply non-generic names for the datatype and the apply-function constructed by defunctionalization.

4 THE ABSTRACT MACHINE

In this section we present the machine obtained as the result of the transformation. We show its definition translated into mathematical notation so that it can be easily read and understood without the need to resort to Racket syntax.

The machine is presented in Table 1. It works with pure lambda terms, local environments, a stack and a global store. Environments assign locations on the store to variables. The syntactic category of values contains terms (intuitively, lambda terms in normal form) and intermediate values (intuitively, abstractions) being closures consisting of a lambda abstraction paired with a local environment and annotated with a location on the global store. The store assigns storable values, or memothunks, to locations. Memothunks are annotated with either \( \times \) (which stands for to do and indicates a code that needs to be executed) or \( \checkmark \) (which stands for done and indicates that a value is already computed and memoized). A stack is simply a sequence of frames; intuitively, it represents the evaluation context of the currently evaluated term. There are two kinds of configurations corresponding to two modes of operation: in configurations \( \langle c, s, \sigma \rangle_v \) the machine evaluates the closure \( c \) to a weak normal form, and in \( \langle v, s, \sigma \rangle_v \) it continues with already computed value \( v \).

The machine starts by loading the input term to the initial configuration with empty environment, empty stack and empty store. Then it proceeds through successive transitions. The first six transitions are standard, they directly correspond to transitions of the lazy variant of Krivine machine from [Danvy and Zerny 2013] (we discuss this correspondence in Section 5.6). To evaluate application \( t_1 t_2 \), transition (1) calls the evaluation of \( t_1 \) and pushes a closure pairing \( t_2 \) with the current environment to the stack. In the case of a lambda abstraction \( \lambda x.t \), transition (2) allocates a fresh location \( \ell \) on the store, fills it with a placeholder for the (strong) normal form of \( \lambda x.t \) and changes the mode of operation. Transitions (3) and (4) apply when the value of the formal parameter \( x \) is actually needed. In this case, the actual parameter is expected to be stored in a thunk at location \( e(x) \) indicated by the environment \( e \). If the thunk contains code to be executed, the location is pushed to the stack and the code is forced by transition (3); otherwise, the thunk contains a memoized value which is simply read by transition (4). If the environment \( e \) does not have an entry for \( x \), then \( x \) turns out to be an open variable and it is taken as a value. Transition (5) implements memoization of actual parameters: when evaluation of an actual parameter is finished, the top of the stack contains the location assigned to the formal parameter, and this location is updated with the computed value. Transition (6) implements \( \beta \)-contraction and delays the evaluation of the actual parameter: new location \( \ell_2 \) is created, a thunk with the actual parameter is stored at \( \ell_2 \), and the evaluation of the body of the lambda abstraction (with an appropriately updated environment) is called.

Transitions (7) and (8) implement normalization of lambda abstractions. They apply when the currently processed closure is a lambda abstraction (annotated with a location \( \ell \)) and there is no argument on the stack (so that transition (6) does not apply). If the thunk at \( \ell \) contains a memoized value, transition (8) simply reads this value. Otherwise it contains the placeholder \( \bot \times \) and transition (7) starts the normalization of the abstraction: the location \( \ell \) is pushed to the stack; the variable \( x \) is \( \alpha \)-renamed to a fresh variable \( \bar{x} \) and stored in a fresh location \( \ell_2 \); the environment is acknowledged about \( \alpha \)-renaming; a context \( \lambda \bar{x}.\Box \) is pushed to the stack; the evaluation of the body of the abstraction is called. When this evaluation returns, transition (11) reconstructs the normalized lambda abstraction and then transition (5) memoizes it at location \( \ell \).
Table 1. The RKNL abstract machine, a reasonable and lazy variant of KN

Identifiers $\ni \ x$

Terms $\ni \ t ::= x \mid t_1 t_2 \mid \lambda x. t$

Locations $\ni \ell$

Envs $\ni \ e <: \text{Identifiers} \rightarrow \text{Locations}$

Closures $\ni \ c ::= (t, e)$

Values $\ni \ v ::= t \mid \ell ::= (\lambda x. t, e)$

Storable Values $\ni \ _ ::= \bot x \mid c x \mid v\checkmark$

Stores $\ni \ \sigma <: \text{Locations} \rightarrow \text{Storable Values}$

Frames $\ni \ f ::= \square c \mid t \square \mid \lambda x. \square \mid \ell ::= \square$

Stacks $\ni \ s ::= [\ ] \mid f ::= s$

Conf$s $ $\ni \ k ::= \langle c, s, \sigma\rangle\checkmark \mid \langle v, s, \sigma\rangle\checkmark$

Transitions:

$t \mapsto \langle(t, [\ ], [\ ], [\ ])\rangle\checkmark$

(1)

$\langle(t_1 t_2, e), s, \sigma\rangle\checkmark \mapsto \langle(t_1, e), \square (t_2, e) :: s, \sigma\rangle\checkmark$

(2)

$\langle(\lambda x. t, e), s, \sigma\rangle\checkmark \mapsto \langle\ell ::= (\lambda x. t, e), s, \sigma[\ell \mapsto \bot x]\rangle\checkmark$

(3)

$\langle(x, e), s, \sigma\rangle\checkmark \mapsto \langle(t, e_2), \ell ::= \square :: s, \sigma\rangle\checkmark$ where $\ell = e(x)$, $\sigma(\ell) = (t, e_2)\checkmark$

(4)

$\langle\ell ::= (\lambda x. t, e), s, \sigma\rangle\checkmark \mapsto \langle\ell \mapsto \bot x\rangle\checkmark$

(5)

$\langle\ell ::= (\lambda x. t, e), s, \sigma\rangle\checkmark \mapsto \langle\ell \mapsto (t, e_2)\rangle\checkmark$

(6)

$\langle\ell ::= (\lambda x. t, e), s, \sigma\rangle\checkmark \mapsto \langle\ell \mapsto (t, e[x := \ell_2])\rangle\checkmark$

(7)

where $\sigma(\ell) = \bot x$

Transition (9) implements normalization of neutral terms. Here $t$ is in normal form, but not an abstraction (see Lemma 5.4), so it is a neutral term; there is an argument on the top of the stack, and this argument is now evaluated. Transitions (10) and (11) reconstruct the final normal forms: transition (10) deals with neutral terms and (11) with lambda abstractions. Finally, the result is unloaded from a final configuration.

**Remark 1.** The whole machine is intended to be naturally implementable as a persistent data structure (cf. [Okasaki 1999]). That means that old configurations of the machine are not destroyed by any transition so they are accessible if the user has references to them. This approach is imposed if data constructors are immutable, which is default in most functional languages including Racket, OCaml and Haskell.
4.1 Elaborate Example Execution

We present the behaviour of the RKNL machine in the elaborate example execution in Table 2. It normalizes the term \((\lambda a \cdot a a) ((\lambda z. \lambda z. z) \Omega)\) where \(I := \lambda a. a\) is the identity, \(\Omega := \omega \omega\) is a well-known divergent term (with \(\omega := \lambda x. x x\) and \(c\) is a free variable. This is one of the simplest examples that uses all transitions of the machine and demonstrates its main features: the machine is able to evaluate open terms (the variable \(c\) is free) and terms under \(\lambda\)-abstractions (the subterm \(\lambda z. I z\) is reduced to \(\lambda z. z\); not-needed arguments are not evaluated (the variable \(y\) is not used in \(\lambda y. \lambda z. I z\), so \(\Omega\) is not evaluated); needed arguments are evaluated only once (even if \(x\) is used twice in \(\lambda x. c x x\), the actual argument \((\lambda y. \lambda z. I z) \Omega\) is evaluated only once).

We apply a convention that locations introduced by transitions (2) are taken from the sequence \(\diamond, \lozenge, \blacklozenge, \triangledown\) and those introduced by transitions (6) and (7) from \(x, y, z, \triangleright\). To shorten the text repeating in Table 2, we define an auxiliary term \(A := \lambda y. \lambda z. I z\), an environment \(e y z := \langle y \mapsto y, z \mapsto z\rangle\), a closure \(x^\ast := \langle x, [x \mapsto \triangleright] \rangle\), and a store \(\sigma_1 := \langle x \mapsto d := \langle \lambda z. z, [y \mapsto y] \rangle, y \mapsto (\Omega, [\triangleright]) x \rangle\).

We also employ here refocusing notation. It is a general technique to shorten the presentation of an abstract machine and to make it more readable. It means that instead of writing the configuration explicitly (the following is the configuration after step 14):

\[
\langle (I z, e y z), \lambda z_0. \diamond :: d := \diamond :: c \quad \triangledown x^\ast :: [], \quad \sigma_1 \ast [d] \mapsto \perp x, z \mapsto z_0 \triangleright \rangle
\]

we recompose the term and the stack into a processed term and use the angle brackets to mark the place of decomposition: \(c \langle d := \lambda z_0. \langle (I z, e y z) \rangle \rangle \triangledown x^\ast \sigma_1 \ast [d] \mapsto \perp x, z \mapsto z_0 \triangleright\). The store remains on the right-hand side after a vertical bar.

We consistently omit locations in the store that are not reachable from the recomposed term. They can be thought of as garbage collected by reference counting. This is why the location \(\diamond\) disappears just after its allocation and the store is empty in the end. Otherwise, the location \(\diamond\) would contain the \(\perp x\) till the end of the execution.

In a higher-level perspective, the machine performs three \(\beta\)-reductions: in step 3 the term \((\lambda x. c x x) (A \Omega)\) is reduced to \(c (A \Omega) (A \Omega)\); in step 11 the latter term, which is \(c (\lambda y. \lambda z. I z) (A \Omega)\), is reduced to \(c (\lambda z. I z) (A \Omega)\); in step 17 the last term is reduced to \(c (\lambda z. z) (A \Omega)\). The argument \(A \Omega\) is normalized in steps 8–21, memoized in step 22 and reused in step 26.

In a lower-level perspective, the machine first moves its focus to \((\lambda x. c x x)\) (step 1), recognizes it as an abstraction and gives it a location \(\diamond\) (step 2). It immediately applies the abstraction to closure \((A \Omega, []\), so the closure moves to the store with a fresh location \(x\) and location \(\diamond\) disappears because it is no longer reachable (step 3). Then the environment \([x \mapsto \triangleright]\) percolates through the applications to the variables (steps 4–5). The variable \(c\) turns out to be free, so it is considered as a value (step 6) and the focus moves to the first remaining \(x\) (step 7). It turns out to be substituted by an unevaluated closure whose value will be memoized at location \(x\) (step 8). Then the focus moves to \(A\) (step 9), which is recognized as an abstraction (step 10) and immediately applied to the
Table 2. Elaborate example execution in refocusing notation

\[
\begin{array}{ll}
0: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} [] \\
1: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} [] \\
2: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
3: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
4: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
5: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
6: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
7: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
8: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
9: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
10: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
11: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
12: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
13: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
14: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
15: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
16: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
17: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
18: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
19: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
20: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
21: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
22: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
23: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
24: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
25: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
26: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
27: & \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow \langle(\lambda x. c \ x \ x) (A \Omega), [] \rangle_{\gamma} \Rightarrow x \ x \ [x \mapsto (A \Omega), []] x \\
\end{array}
\]
closure ($\Omega[::])$, so $A$’s body is evaluated in the extended environment (step 11). The body is also an abstraction (step 12). The abstraction is memoized at location $x$ (step 13). It has no arguments, so it is normalized: the parameter $z$ is renamed to a fresh name $z_0$, and the normal form of the abstraction will be memoized at location $d$ (step 14). Note that $d:=\Box$ is outside the focus, so it is a frame on the stack now. The focus moves to $I$ and the environment $e_{yz}$ percolates (step 15). The identity is an abstraction (step 16), and it is applied to closure ($z, e_{yz}$), so the closure moves to the store with a fresh location $\nu$ (step 17). The variable $x$ constituting the identity’s body looks up the location $\nu$ containing the unevaluated argument (step 18). By subsequent lookup, it is evaluated to $z_0$ (step 19) and memoized at $\nu$ (but it is unreachable and disappears) (step 20). The abstraction $\lambda z_0.z_0$ is reconstructed (step 21) and memoized at $d$ as a normal form of $\lambda z.I z$ (step 22). The machine reconstructs $c_0 \lambda z_0.z_0$ (step 23) and moves focus to the second $x$ (step 24). The variable $x$ looks up the location $x$ with the value of $(A \Omega[::])$ (step 25) that looks up the location $d$ for its normal form by transition $(8)$ (step 26). Finally, the result $c_0 (\lambda z_0.z_0) \lambda z_0.z_0$ is reconstructed (step 27).

To summarize, the application $A \Omega$ is evaluated only once, its value is normalized only once, and the nontermination of $\Omega$ is contained. Occurrences of $x$ (in $c x x, I$, and $\Omega$) do not collide, the bound variable $z$ is renamed, and the free variable $c$ stays untouched.

### 4.2 Empirical Execution Lengths

Abstract cost models including abstract machines give us an opportunity to measure the program complexity independently of the efficiency of a physical machine. Thanks to that, such results should be easily reproducible. In Table 3 we present the number of steps to the normal form for normal order, KN (implemented as in Table 4 of [Crégut 2007]), and RKNL for five term families. The first two families are the families from Table 9 of [Crégut 2007] up to renaming. We name a few more terms to define these families ($\lambda x$ is the normal form of term $K I$).

\[
\begin{align*}
K &:= \lambda x.\lambda y.x, \quad \lambda x.\lambda y.y, \quad \text{pair} := \lambda x.\lambda y.\lambda f. f x y, \quad \text{dub} := \lambda x.\lambda f. f x x \\
\text{pred} &:= \lambda n.\lambda f. \lambda x. \left( n \left( \lambda e. \right. \left. \text{pair}(\text{e K})(\text{f(e K)}) \right) \text{(pair x x)} \right) x
\end{align*}
\]

The numbers of steps were measured by running the machines KN an RKNL and counting their steps (and $\beta^*$-steps of KN for $\rightarrow^*$). The closed forms are verified for natural numbers from 1 to 9 because some of the sequences have a different value for $n = 0$.

| term family | $n=0$ | KN | RKNL |
|-------------|-------|----|------|
| $c_n c_2 I$ | $3 \cdot 2^n - 1$ | $15 \cdot 2^n - 6$ | $10 \cdot 2^n + 5n + 5$ |
| $\text{pred } c_n$ | $6n + 8$ | $26n + 25$ | $30n + 41$ |
| $\lambda x.\lambda y.x$ | $2^n + 1$ | $12 \cdot 2^n - 3$ | $9n + 15$ |
| $c_n \text{dub } I$ | $2^n + 1$ | $23 \cdot 2^n - 14$ | $18n + 15$ |
| $c_n \text{dub } (\lambda x.I x)$ | $2 \cdot 2^n + 1$ | $26 \cdot 2^n - 14$ | $18n + 20$ |

For family $\text{pred } c_n$, the overhead of memoization of RKNL with respect to the number of steps of KN is visible. Nevertheless, RKNL is asymptotically not worse than KN. It is intuitive because of RKNL construction, and possibly it could be formally proven, but here we limit ourselves to empirical evidence supporting this hypothesis. For family $c_n c_2 I$, RKNL even reduces the coefficient.

Proc. ACM Program. Lang., Vol. 6, No. ICFP, Article 94. Publication date: August 2022.
of the exponential element. For the three remaining families, RKNL achieves normal form in a number of steps logarithmic w.r.t. normal-order strategy, so it is implosive in terms of [Accattoli et al. 2021].

5 PROPERTIES OF THE MACHINE

In this section we discuss correctness of the derived machine and its complexity.

5.1 Decoding

We say that a configuration $k$ is reachable if there exists a sequence of machine transitions starting in an initial configuration and ending in $k$. We define a decoding of reachable configurations (denoted $\cdot_k$) to terms in order to refer micro-step operational semantics of the machine to small-step operational semantics of normal order reduction. Formally, our goal is to show that whenever $k \rightarrow k'$ for a reachable configuration $k$, then $k \xrightarrow{a} k'$. We prove it in Lemma 5.8.

Table 4. Decoding of RKNL

| Configuration | Decoding |
|---------------|----------|
| $(t_1, t_2, e), \Sigma_c$ | $(t_1, e), \Sigma_c (t_2, e), \Sigma_c$ |
| $(\lambda x. t), \Sigma_c$ | $\lambda \overline{x}. (t, e[x:=\overline{x}]), \Sigma_c$ |
| $(x, e), \Sigma_c$ | $\begin{cases} \overline{x} & : e(x) = \overline{x} \text{ defined by the rule above} \\ (t_2, e_2), \Sigma_c & : (t_2, e_2) \overline{x} \text{ initialized } e(x) \text{ with (6) in } \Sigma \\ \hat{x} & : \hat{x} \overline{v} \text{ initialized } e(x) \text{ with (7) in } \Sigma \\ x & : e(x) \text{ not defined and not initialized} \end{cases}$ |
| $t, \Sigma_c \cdot$ | $t$ |
| $\ell := (\lambda x. t), \Sigma_c$ | $\lambda \overline{x}. (t, e), \Sigma_c$ |
| $(t, e), \sigma), \Sigma_k$ | $s, \Sigma_s [(t, e), \Sigma_c]$ |
| $(v, s, \sigma), \Sigma_k$ | $s, \Sigma_s [v], \Sigma_c$ |
| $k_k := k, \Sigma_k$ | $\begin{cases} \TOP & : []_c := \TOP \\ (t, e) ::= s, \Sigma_s := s, \Sigma_s [(t, e), \Sigma_c] \\ t \TOP ::= s, \Sigma_s := s, \Sigma_s [t \TOP] \\ \lambda x. \TOP ::= s, \Sigma_s := s, \Sigma_s [\lambda x. \TOP] \\ \ell ::= s, \Sigma_s := s, \Sigma_s \end{cases}$ |

Our decoding is presented in Table 4. It uses decodings $\cdot_c$ of closures, $\cdot_v$ of values, and $\cdot_s$ of stacks. For the most part, the decoding is simple and standard (the currently processed term or closure is decoded and plugged in the context obtained from the decoding of the stack), however the case of variables requires an explanation. As mentioned above, we are interested in the correspondence between our machine and the normal-order strategy. Since the latter recomputes function arguments that occur more than once in the function body, there are moments in time when some of the occurrences are already computed and the others are not. To keep track of this, in the decoding of a variable we are not interested in the current state of the store, but instead we prefer the state of the variables requires an explanation. As mentioned above, we are interested in the correspondence between our machine and the normal-order strategy. Since the latter recomputes function arguments that occur more than once in the function body, there are moments in time when some of the occurrences are already computed and the others are not. To keep track of this, in the decoding of a variable we are not interested in the current state of the store, but instead we prefer the state of the store from the moment of the initialization of the variable. Therefore each of the decoding functions carries an additional argument $\Sigma_k$, namely the full sequence of configurations from the initial one to $k$. This additional argument is also implicitly present in decodings $k_k$ of configurations; we omit it only to shorten the notation.

Another subtle point is the treatment of free and bound variables. We treat them differently in order to be able to handle open terms. Free variables are never added to any environment and
thus they remain untouched in the decoding. On the other hand, bound variables are overlined. This way the closure \((\lambda x. \lambda y. xy)y, []\) decodes to \((\lambda x. \lambda y. x\overline{y})y\) and its reduct \((\lambda y. xy, [x \mapsto y])\) decodes to \(\lambda \overline{y}. y\overline{y}\) thus avoiding the capture of variable \(y\).

Alternative decodings. There are also two alternative decodings independent of the execution history that can be considered. We could naturally decode variables to values that are currently in the store. Then the transition (5) would be decoded into a replacement of terms in many places in the decoding at once. The second alternative decoding could decode variables to terms reconstructed below appropriate \(\ell := \square\) frames. Then the transition (6) would be decoded to parallel \(\beta\)-reduction in multicontexts as in [Accattoli et al. 2021] and the transition (5) would not change the decoded term.

5.2 Ghost Abstract Machine for RKNL

One of the goals in the design of the normalizer of Listing 2 was to keep the derived abstract machine simple. Indeed, the machine has only 11 transitions, that is 5 transitions more than the machine for weak call by need (recalled in Section 5.6) and as many as 13 transitions fewer than the known machine for strong call by need in [Biernacka and Charatonik 2019]. One of the design choices to achieve this goal was not to include the grammar of normal forms in the syntactic category of values. However, to reason about correctness, we need to know that only terms in normal form may appear as values. Therefore, for theoretical purposes, we introduce a ghost abstract machine RKNLi – a more refined version of RKNL with explicitly expressed invariants – presented in Table 5.1. The ghost machine bisimulates RKNL so that every reachable configuration of RKNL is a projection of a reachable configuration of the ghost machine. It can be seen as a proof technique tailored to reason about invariants of abstract machines (in this case, RKNL, in particular in Lemmas 5.5 and 5.12).

The main difference between RKNL and RKNLi is the grammar of normal terms in RKNLi. Note that each neutral term is in normal form. For technical reasons we want to distinguish between neutral terms and normal forms, so we add an explicit coercion: a neutral term \(\overline{a}\) becomes a normal term \([\overline{a}]\). Introduction of normal terms triggers further refinements in stacks and configurations. Now we have two variants of stacks: applicative stacks may have an argument on top while non-applicative stacks certainly do not have arguments on top. This in turn introduces two variants of \(\Delta\)-configurations: one with applicative stacks and one with non-applicative stacks. Yet another difference is the introduction of a separate store: RKNLi has two stores, one for evaluated arguments and one for normalized lambda abstractions.

Changes in the grammars lead to changes in transitions. RKNLi has two versions of transition (5): one for normal terms and one for values (which includes neutral terms). It also has an additional transition (9a) that implements the coercion from neutral to normal terms: after finishing the evaluation of an argument of a neutral term initialized by transition (9) (as well as after evaluating the body of a lambda abstraction initialized by transition (7) and after completing the whole evaluation), the result being a neutral term must be coerced to a normal term. From the point of view of the RKNL machine, this step is silent. Lemma 5.4 states formally the correspondence between the two machines.

5.3 Soundness

In this section we prove the soundness of the RKNL machine with respect to the normal-order strategy. Note that we do not rely here on the correctness of the transformations used in the sent 1The ghost machine is also derivable by the same methodology from a refined variant of the NbE normalizer [Biernacka et al. 2020].
Table 5. The RKNLi ghost abstract machine, RKNL with explicit shape invariant

\[
\begin{align*}
\text{Identifiers} & \ni x \\
\text{Terms} & \ni t ::= x \mid t_1 t_2 \mid \lambda x . t \\
\text{Normal Terms} & \ni n ::= \lambda x . n \mid [a] \\
\text{Neutral Terms} & \ni a ::= x \mid a n \\
\text{Locations} & \ni \ell \\
\text{Envs} & \ni e <: \text{Identifiers} \rightarrow \text{Locations} \\
\text{Closures} & \ni c ::= (t, e) \\
\text{Locations}' & \ni \ell' \\
\text{Values} & \ni v ::= a \mid \ell' ::= (\lambda x . t, e) \\
\text{Storable Values} & \ni _- ::= c_x \mid v_{\checkmark} \\
\text{Stores} & \ni \sigma <: \text{Locations} \rightarrow \text{Storable Values} \\
\text{Optional Normal Terms} & \ni _- ::= \bot x \mid n_{\checkmark} \\
\text{Stores'} & \ni \sigma' <: \text{Locations'} \rightarrow \text{Optional Normal Terms} \\
\text{Applicative Stacks} & \ni \alpha ::= \ell ::= \Box ::= \alpha \mid \Box c ::= \alpha \mid [\Box] ::= v \\
\text{Non-applicative Stacks} & \ni v ::= \ell' ::= \Box ::= v \mid [\Box] \mid \lambda x . \Box ::= v \mid a \Box ::= \alpha \\
\text{Conf} & \ni k ::= \langle c, \alpha, \sigma, \sigma' \rangle_{\checkmark} \mid \langle v, \alpha, \sigma, \sigma' \rangle_{\Delta} \mid \langle n, v, \sigma, \sigma' \rangle_{\Delta'} \\
\text{Transitions} & \ni t \mapsto \langle \langle t, [\Box], [\Box] :: [\Box], [\Box] \rangle_{\checkmark} \\
\langle t_1, t_2, e, \alpha, \sigma, \sigma' \rangle_{\checkmark} & \rightarrow \langle \langle t_1, e, \Box :: (t_2, e) :: \alpha, \sigma, \sigma' \rangle_{\checkmark} \\
\langle (\lambda x . t, e), \alpha, \sigma, \sigma' \rangle_{\checkmark} & \rightarrow \langle \ell' ::= (\lambda x . t, e), \alpha, \sigma, \sigma' \ast (\ell' \mapsto \bot x) \rangle_{\Delta} \\
\langle (x, e), \alpha, \sigma, \sigma' \rangle_{\checkmark} & \rightarrow \langle \langle t_2, e \rangle, \ell ::= \Box :: \alpha, \sigma, \sigma' \rangle_{\checkmark} \\
\text{where } \ell & = e(x), \sigma(\ell) = (t, e_2) x \\
\text{where } \sigma(e(x)) & = v_{\checkmark} \lor (v = x \notin e) \\
\langle v, \ell ::= \Box :: \alpha, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle v, \alpha, \sigma \ast (\ell ::= v_{\checkmark}) \rangle_{\Delta} \\
\langle n, \ell' ::= \Box :: \alpha, \sigma, \sigma' \rangle_{\Delta'} & \rightarrow \langle n, v, \sigma, \sigma' \ast (\ell' ::= n_{\checkmark}) \rangle_{\Delta'} \\
\langle \ell' ::= (\lambda x . t, e), \Box ::= (t_2, e_2) :: \alpha, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle \langle t, e \ast (x := e_2) \rangle, \alpha, \sigma \ast (\ell_2 \mapsto (t_2, e_2) x), \sigma' \rangle_{\checkmark} \\
\langle \ell' ::= (\lambda x . t, e), \Box ::= v, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle \langle t, e \ast (x := \lambda x . t) \rangle, \Box ::= \lambda x . \Box :: \ell' ::= \Box :: v, \sigma \ast (\ell' \mapsto \checkmark v), \sigma' \rangle_{\checkmark} \\
\text{where } \sigma'(\ell') & = \bot x & (7) \\
\langle \ell' ::= (\lambda x . t, e), \Box ::= v, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle n, v, \sigma, \sigma' \rangle_{\Delta'} & (8) \\
\langle a, \Box ::= (t_2, e_2) :: \alpha, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle \langle t_2, e_2, \Box ::= a \Box :: \alpha, \sigma, \sigma' \rangle_{\checkmark} \\
\langle a, \Box ::= v, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle [a], v, \sigma, \sigma' \rangle_{\Delta'} & (9) \\
\langle n, \alpha, \Box ::= \alpha, \sigma, \sigma' \rangle_{\Delta} & \rightarrow \langle a n, \alpha, \sigma, \sigma' \rangle_{\Delta} & (10) \\
\langle n, \lambda x . \Box ::= v, \sigma, \sigma' \rangle_{\Delta'} & \rightarrow \langle \lambda x . n, v, \sigma, \sigma' \rangle_{\Delta'} \\
\langle n, [\Box], \sigma, \sigma' \rangle_{\Delta'} & \rightarrow n & (11)
\end{align*}
\]
tool as the tool is not formally verified. Instead, we directly show that the machine faithfully realizes the strategy.

The result is proved by a sequence of lemmas showing that each transition of the machine is sound, where soundness of a transition \( k \rightarrow k' \) is understood as reducibility (up to \( \alpha \)-renaming) of the decoding of \( k \) to the decoding of \( k' \). Note that since \( \beta \)-reduction commutes with \( \alpha \)-conversion, a sequence of sound transitions is sound.

**Lemma 5.1 (load correctness).** \( ([t, []], [[], []])_{\mathcal{R}_k}^\alpha = t \)

**Proof.** As the closure \((t, [])\) is decoded recursively, the environment is extended only by overlined bound variables of the source term. While bound source variables are decoded to overlined versions of themselves and bound by the appropriate abstractions, the free variables and the structure of the term remain untouched, resulting in an \( \alpha \)-renamed version of the loaded term. \( \square \)

**Lemma 5.2 (overhead transitions).** If \( k \overset{(i)}{\rightarrow} k' \) then \( \underline{k}_k = k'_{k} \) for \( i \notin \{4, 6, 7, 8\} \).

**Proof.** Transitions (1), (9), (10) and (11) only rotate the processed term. Store modifications of transitions (2) and (5) do not affect the decoding. Transition (3) replaces a variable by a closure that is already its decoding. \( \square \)

**Lemma 5.3 (alpha transition).** If \( k \) is a reachable configuration of RKNL and \( k \overset{(*)}{\rightarrow} k' \), then \( \underline{k}_k = \underline{k'}_k \).

**Proof.** Let \( C \) be the context obtained from the decoding of the stack in \( k \). Then \( C[\lambda \bar{x}. \Box] \) is the decoding of the stack in \( k' \). The decoding of the value \( \ell := (\lambda x. t, e) \) in \( k \) is \( \lambda \bar{x}. (t, e[x:=\bar{x}]), \Sigma_{k'} \), where all occurrences of \( x \) are decoded to \( \bar{x} \). On the other hand, the decoding of the closure in \( k' \) is \( (t, e[x:=\ell_2]), \Sigma_{k'} \). Since \( \Sigma_{k'} \) differs from \( \Sigma_{k} \) only by the last configuration initializing \( \ell_2 \) with \( \bar{x} \), this gives the same term with \( \bar{x} \) replaced by \( \bar{x} \). Hence \( C[\ell := (\lambda x. t, e), \Sigma_{k'}] \) is \( \alpha \)-equivalent to \( C[\lambda \bar{x}. [(t, e[x:=\ell_2]), \Sigma_{k'}]] \), which means that \( \underline{k}_k = \underline{k'}_k \). \( \square \)

In the following, by a projection of a stack (\( \alpha \) or \( \nu \)) of RKNLi we mean this stack with removed all occurrences of the frame \( \Box \) and removed all tags \( \cdot \) from normal terms. This notion of projection lifts naturally to terms, configurations and grammars occurring in the definition of RKNLi in Table 5.

The following lemma is proved by a simple induction and case analysis of all transitions. In particular it implies that there are two types of stacks \( s \) occurring in reachable configurations of RKNL: projections of stacks generated by grammars of \( \alpha \) and \( \nu \) from the definition of RKNLi.

**Lemma 5.4 (shape invariant).** Every reachable configuration of RKNL is a projection of a reachable configuration of RKNLi.

**Lemma 5.5 (stack shape invariant).** For any reachable configuration \( k \) with stack \( s \), the stack decodes to a normal-order context: \( s, \Sigma_{k} \in \text{NO} \).

**Proof.** By Lemma 5.4 the stack \( s \) is a projection of an \( \alpha \)- or \( \nu \)-stack of RKNLi. Frames \( \ell := \Box \) are ignored in the decoding from Table 4. Grammars of \( \alpha \) and \( \nu \) with removed frames of this form correspond directly to the inside-out grammar of normal-order contexts from Section 2.1. \( \square \)
Lemma 5.6 (Beta Transition). If \( k \) is a reachable configuration of RKNL and \( k \xrightarrow{(6)} k' \), then \( k \xrightarrow{no} k' \).

Proof. Let \( C \) be the context obtained from the decoding of the stack in \( k \). The left-hand side of transition (6) decodes to an application of \((\lambda x.t, e), \Sigma_k \), which is \( \lambda x.([t[e[x:=\overline{x}]], \Sigma_k] \) to \((t_e, e), \Sigma_k \) in context \( C \). The right-hand side decodes to \((t[e[x:=\ell_2]], \Sigma_{k'} \) in the same way.

Since \( \ell_2 \) is initialized to \((t_2, e) \) in \( \Sigma_k \) and \((t_2, e), \Sigma_{k'} = (t_2, e), \Sigma_k \), we have \((t[e[x:=\ell_2]], \Sigma_{k'} \) = \((t[x:=\ell_2], e_{x}), \Sigma_k \). Here \( e_{x} \) denotes the environment \( e \) without definition for \( x \). This implies that \( k \xrightarrow{no} k' \) is an instance of a \( \beta \)-reduction. Since \( C \) is a normal-order context by Lemma 5.5, we have \( k \xrightarrow{no} k' \).

Lemma 5.7 (Bypass Transitions). If \( k \) is a reachable configuration of RKNL and \( k \xrightarrow{(i)} k' \) then \( k \xrightarrow{no} k' \) for \( i \in \{4, 8\} \).

Proof. First observe that both transitions (4) and (8) involve some location \( \ell \) on the store: in the case of (8) \( \ell \) is explicitly mentioned while in the case of (4) \( \ell = e(x) \). Location \( \ell \) was created with transition (2), (6) or (7), in all three cases it was fresh at the moment of creation. Since none of transitions in Table 1 overwrites a storable value of the form \( v_{\lambda} \), such a value is uniquely determined by the location.

If \( \ell \) was created with transition (7), then \( i = 4 \) and \( k \xrightarrow{no} k' \) so we are done. For the rest of the proof we assume that \( \ell \) was created with transition (2) or (6), so at the moment of creation it contained no value.

Now the proof goes by induction on locations \( \ell \) in the order in which they are filled by transition (5). This transition requires the frame \( \ell := \square \) on top of stack. Let \( k \) be the configuration in which \( \ell := \square \) was pushed on the stack (an instance of the left-hand side of transition (3) or (7)). Let \( k \) be the configuration in which \( \ell \) was filled (an instance of the left-hand side of transition (5)). The sequence \( k \xrightarrow{\ell} k \) contains no transitions (4) and (8) involving \( \ell \) and thus by Lemmas 5.2–5.6 and by induction hypothesis it is sound, so it decodes to \( k \xrightarrow{no} k \).

Now we have two cases.

\( i = 4 \): In this case \( k = \langle (x, e), s, \sigma \rangle_{\lambda} \) and \( k' = \langle v, s, \sigma \rangle_{\lambda} \), the frame \( \ell := \square \) was pushed with transition (3), \( \ell = e(x) \), so for some stack \( s' \) and store \( \sigma' \) we have

\[ k_{\ell} = \langle (x, e), s', \sigma' \rangle_{\lambda} \] and \( k_{\ell} = \langle v, s, \sigma \rangle_{\lambda} \)

Then \( \sigma_{\ell} = s', \Sigma_{\ell} = [(x, e), \Sigma_{\ell} \] and \( \sigma_{\ell} = s', \Sigma_{\ell} = \Sigma_{\ell} \Sigma_{\ell} \]

Moreover, \( s', \Sigma_{\ell} \Sigma_{\ell} = s', \Sigma_{\ell} \)

and this context is a normal-order context by Lemma 5.5. In the light of \( \sigma_{\ell} = \sigma_{\ell} \) this gives that \( \langle x, e \rangle, \Sigma_{\ell} = \sigma_{\ell} \)

Since \( s, \Sigma_{\ell} \Sigma_{\ell} = s, \Sigma_{\ell} \) and \( v, \Sigma_{\ell} = v, \Sigma_{\ell} \) and the former context is a normal-order context by Lemma 5.5, we obtain that

\[ k \xrightarrow{no} k \]

\( i = 8 \): In this case \( k = \langle (\lambda x.t, e), s, \sigma \rangle_{\lambda} \) and \( k' = \langle v, s, \sigma \rangle_{\lambda} \), the frame \( \ell := \square \) was pushed with transition (7), so for some stack \( s' \) and store \( \sigma' \) we have

\[ k_{\ell} = \langle (\lambda x.t, e), s', \sigma' \rangle_{\lambda} \]

and \( k_{\ell} = \langle v, \ell := \square, s', \sigma' \rangle_{\lambda} \).
Then \( k_{\ell} = s', \Sigma_{k_{s'}} \[(\lambda x.t, e), \Sigma_{k_{\ell}} \] \) and \( k_{v} = s', \Sigma_{k_{v}} \[v, \Sigma_{k_{v}} \] \). Moreover, \( s', \Sigma_{k_{v}} = s', \Sigma_{k_{s'}} \) and this context is a normal-order context by Lemma 5.5. In the light of \( k_{\ell} \rightarrow_{no} k_{v} \), this gives that \( (\lambda x.t, e), \Sigma_{k_{s'}} \rightarrow_{no} = \alpha v, \Sigma_{k_{v}} \).

Since \( s, \Sigma_{k_{s}} = s, \Sigma_{k_{s'}} \) and \( v, \Sigma_{k_{v}} = v, \Sigma_{k_{v}}' \) and the former context is a normal-order context by Lemma 5.5, we obtain that

\[ k = s, \Sigma_{k_{s}} \[(\lambda x.t, e), \Sigma_{k_{\ell}} \] \rightarrow_{no} = \alpha s, \Sigma_{k_{s'}} \[v, \Sigma_{k_{v}} \] = k'. \]

\( \square \)

The following lemma is an immediate consequence of Lemmas 5.2–5.7.

**Lemma 5.8 (Step Soundness).** If \( k \) is a reachable configuration of RKNL and \( k \rightarrow k' \) then \( k_{\ell} \rightarrow_{no} k' \).

**Lemma 5.9 (Unload Correctness).** For any reachable configuration \( \langle t, [], \sigma \rangle_{\Delta} \) it decodes to a term \( t \) in normal form: \( \langle t, [], \sigma \rangle_{\Delta} k = t \rightarrow_{\beta} \).

**Proof.** The decoding \( _\Delta \) of a term in a \( \Delta \)-configuration returns the same term. By the shape invariant, it is a normal term.

\( \square \)

Lemmas 5.1–5.9 lead to the main result of this section:

**Theorem 5.10 (Soundness).** If the machine starts from a (possibly open) term \( t_{0} \) and computes \( t \) (i.e., \( \langle t_{0}, [], [], [] \rangle \rightarrow \langle t, [], [], [] \rangle \)), then \( t_{0} \) reduces to a normal form \( t \) in the normal order reduction in zero or more steps (i.e., \( t_{0} \rightarrow_{no} t \rightarrow_{\beta} \)).

### 5.4 Complexity

We introduce the potential function \( \Phi_{k} \) to bound the number of steps the machine in a given configuration can make without performing \( \beta \)-transition (6). It is defined together with auxiliary potential functions for terms, values, stacks and stores in the Table 6. The number of consecutive non-\( \beta \)-transitions from a given configuration is bound by the two following lemmas.

**Table 6. The potential function for RKNL**

\[
\begin{align*}
\Phi_{t}(t_{1} t_{2}) & := 3 + \Phi_{t}(t_{1}) + \Phi_{t}(t_{2}) \\
\Phi_{t}(\lambda x.t) & := 4 + \Phi_{t}(t) \\
\Phi_{t}(x) & := 2 \\
\Phi_{v}(t) & := 0 \\
\Phi_{v}(t := (\lambda x.t, e)) & := 1 \\
\Phi_{\sigma}(k) & := \sum_{t \in k \land \alpha(t) = (t, e) \land \ell := \emptyset} \Phi_{t}(t) + \sum_{t := (\lambda x.t, e) \in k \land \sigma(t) = \bot \land \ell := \emptyset} (2 + \Phi_{t}(t)) \\
\Phi_{k}(\langle t, e, \sigma \rangle_{\gamma}) & := \Phi_{t}(t) + \Phi_{s}(s) + \Phi_{v}(k) \\
\Phi_{k}(\langle v, s, \sigma \rangle_{\Delta}) & := \Phi_{v}(v) + \Phi_{s}(s) + \Phi_{\sigma}(k)
\end{align*}
\]

Proc. ACM Program. Lang., Vol. 6, No. ICFP, Article 94. Publication date: August 2022.
Lemma 5.11 (decrease). If \(k \rightarrow k'\) then \(\Phi(k(k') > \Phi(k')\).

**Proof.** By case analysis on transition rules:

\[
3 + \Phi_1(t_1) + \Phi_2(t_2) + \Phi_3(s(s) + \Phi_4(k) > \Phi_1(t_1) + 2 + \Phi_3(s) + \Phi_5(t_2) + \Phi_4(k)
\]

(1)

\[
4 + \Phi_1(t) + \Phi_3(s) + \Phi_6(k) > 1 + \Phi_3(s) + \Phi_6(k) + (2 + \Phi_1(t))
\]

(2)

\[
2 + \Phi_3(s) + \Phi_7(k' + \Phi_8(t)) > \Phi_1(t) + (1 + \Phi_3(s) + \Phi_8(k'))
\]

(3)

\[
2 + \Phi_3(s) + \Phi_6(k) > \Phi_9(v) + \Phi_3(s) + \Phi_6(k)
\]

(4)

\[
\Phi_9(v) + 1 + \Phi_3(s) + \Phi_4(k) > \Phi_3(v) + \Phi_3(s) + \Phi_4(k)
\]

(5)

\[
1 + \Phi_3(s) + (\Phi_6(k') + 2 + \Phi_1(t)) > \Phi_1(t) + 1 + 1 + \Phi_3(s) + \Phi_4(k')
\]

(7)

\[
1 + \Phi_3(s) + \Phi_6(k) > 0 + \Phi_3(s) + \Phi_6(k)
\]

(8)

\[
0 + 2 + \Phi_3(s) + \Phi_1(t_2) + \Phi_6(k) > \Phi_1(t_2) + 1 + \Phi_3(s) + \Phi_6(k)
\]

(9)

\[
0 + 1 + \Phi_3(s) + \Phi_6(k) > 0 + \Phi_3(s) + \Phi_6(k)
\]

(10)

\[
0 + 1 + \Phi_3(s) + \Phi_6(k) > 0 + \Phi_3(s) + \Phi_6(k)
\]

(11)

Lemma 5.12 (increase). If \(k\) is a configuration reachable from a term \(t_0\), then \(k \rightarrow k'\) implies \(\Phi(k(k) + \Phi_1(t_0) > \Phi(k')\).

**Proof.** Table 5 shows that abstractions are only constructed when rebuilding normal forms, which cannot be applied. Therefore, any applied abstraction has to be a subterm of the source term and, using the notations from (6), \(\Phi_1(t_0) > \Phi_1(t)\). Here \(\Phi_6(k') = \Phi_6(k) + \Phi_1(t_2)\), so

\[
1 + 2 + \Phi_3(s) + \Phi_1(t_2) + \Phi_6(k) + \Phi_1(t_0) > \Phi_1(t) + 1 + 1 + \Phi_3(s) + \Phi_6(k) + \Phi_1(t_2).
\]

We can observe the behaviour of the potential by plotting it. Figure 1 shows potentials of configurations and stores of example from Table 2. For reference, the potentials of selected subterms are: \(\Phi_1(I) = 6\), \(\Phi_1(\omega) = 11\), \(\Phi_1(\Omega) = 25\), \(\Phi_1(\lambda x.c x x) = 47\), \(\Phi_1(\lambda x.c x x) = 16\). The store potential counts only those locations that are present in configurations in Table 2. Note that the overall potential in this example is never increased because each abstraction is either applied or normalized, but not both, so their potentials paid the costs of their \(\beta\)-reductions. Arbitrarily many periodic increases of the potential could be observed, for example, in the normalization of the term \(\Omega\) alone that needs it to diverge as expected.

We can summarize the two lemmas above by the theorem that the machine simulates the normal-order strategy in a bilinear number of steps, i.e., linear in the number of \(\beta\)-reductions and linear in the size \(|t_0|\) of the input term \(t_0\) (because the potential of the input term is bounded by its size times a constant factor).

**Theorem 5.13.** Let \(\rho\) be a sequence of consecutive machine transitions starting from term \(t_0\) to configuration \(k'\), \(|\rho|\) be the number of steps in \(\rho\) and \(|\rho|_{\beta}\) be the number of normal-order \(\beta\)-reductions from \(t_0\) to \(k'\). Then \(|\rho| \leq (|\rho|_{\beta} + 1) \cdot \Phi_1(t_0)\).

The only non-local operations done by the RKNL machine are operations on environments and the store. The store is never duplicated so it can be implemented as locations on the heap which is standard. Then operations on the store are assumed to be performed in constant time.

Environments are dictionaries coupled only with subterms of the source term. Thus, they require at most as many entries as variables in the source term. Their implementation with balanced
trees gives the time complexity of operations logarithmic in the number of entries. Alternatively, implementation with tries gives the complexity proportional to the length of the longest identifier in the source term.

Normal-order strategy can be simulated with Useful Milner Abstract Machine with quadratic overhead [Accattoli 2016; Accattoli et al. 2021]. We improve this overhead by showing that at least \( n \) steps can be simulated in time of single transition times a bilinear number of steps. The cost of a single transition is bound to be logarithmic in the size of the input term what entails that the overall complexity of the simulation is quasibilinear. Precisely, the computational time complexity in the RAM model of the presented simulation is of quasibilinear order that can be expressed as \( O((|\rho|_\beta + 1) \cdot |t_0| \log |t_0|) \). We guess that in practice the number of distinct source variables can be bounded by a constant and that would cause the log \( |t_0| \) part of the given formula disappears. However, improving the practical efficiency would require practical profiling of the RKNL abstract machine or the corresponding normalizer from Listing 2.

5.5 Completeness

The last thing we show concerning the correctness of RKNL with respect to normal order is that the machine will not get stuck nor fell into a silent loop. We express it with the following lemma.

**Lemma 5.14 (Step Completeness).** If \( k \) is a reachable configuration and \( k \xrightarrow{\#} t' \), then there exists \( k' \) such that \( k \rightarrow k' \) and \( t' \xrightarrow{\#} k' \).

**Proof.** The pattern matching of the machine is exhaustive, which is best visible in Table 5, and thus the machine never gets stuck. Since \( k \) decodes to a non-normal form (of normal order), by Lemmas 5.2 and 5.3 the machine cannot reach the terminal configuration without executing one of (4), (6) or (8) transitions. By Lemma 5.11 one of these transitions will be executed within a finite number of steps; let \( k' \) be the reached configuration. Now Lemma 5.8 guarantees \( t' \xrightarrow{\#} k' \) because normal order is a deterministic strategy.
The lemma can be related to the existence of a normal form in general as in the following theorem.

**Theorem 5.15 (Completeness).** If a (possibly open) term \( t_0 \) reduces in zero or more steps to a normal form \( t \) (i.e., \( t_0 \beta t \rightarrow \beta t \rightarrow \beta t \cdots \rightarrow \beta t \)), then the machine that starts from \( t_0 \) computes \( t \) (i.e., there exists \( t' \) such that \( t =^\alpha t' \) and \( \langle (t_0, [\; ], [\; ], [\; ]), \rangle \rightarrow \langle t', [\; ], \sigma \rangle_\Delta \)).

**Proof.** Normal order is a complete strategy, so from \( t_0 \beta t \rightarrow \beta t \rightarrow \beta t \) we know that \( t_0 \rightarrow_0 t \rightarrow_\beta \). Normal order is deterministic, so we can iterate Lemma 5.14 over the normal-order reduction path. When the machine decodes to a normal form, it has to reach the terminal configuration in a finite number of steps because of the decreasing potential. \( \square \)

### 5.6 Connection with Weak Call by Need

In this section we show that the strategy realized by our machine is a conservative extension of the standard weak call-by-need evaluation. We show that the machine RKNL extends the lazy variant of the Krivine machine, which, according to [Danvy and Zerny 2013], is a canonical implementation of the weak call-by-need evaluation strategy using actual substitutions. KL is displayed in Table 7.

| Terms | \( t \) | \( t_1 t_2 \) | \( \lambda x. t \) |
|-------|--------|-------------|----------------|
| Values | \( \nu \) | \( \lambda x. t \) |
| Contexts | \( E \) | \( E t \) | \( x := E \) |
| Stores | \( \sigma \) | \( \epsilon \) | \( \sigma[x := t] \) |
| Confs | \( k \) | \( \langle t, E, \sigma \rangle_{\text{term}} \) | \( \langle E, \nu, \sigma \rangle_{\text{cont}} \) |

**Transitions:**

1. \( t \rightarrow \langle t, \Box, \epsilon \rangle_{\text{term}} \)
2. \( \langle \lambda x. t, E, \sigma \rangle_{\text{term}} \rightarrow \langle E, \lambda x. t, \sigma \rangle_{\text{cont}} \)
3. \( \langle x, E, \sigma \rangle_{\text{term}} \rightarrow \langle t, E[x := \Box], \sigma \rangle_{\text{term}} : t = \sigma(x), t \notin \text{Values} \)
4. \( \langle x, E, \sigma \rangle_{\text{term}} \rightarrow \langle E, \nu, \sigma \rangle_{\text{cont}} : \nu = \sigma(x) \)
5. \( \langle E[x := \Box], \nu, \sigma \rangle_{\text{cont}} \rightarrow \langle E, \nu, \sigma[x := \nu] \rangle_{\text{cont}} \)
6. \( \langle E[\Box t_2], \lambda x. t, \sigma \rangle_{\text{cont}} \rightarrow \langle t[x'/x], E, \sigma[x' = t_2] \rangle_{\text{term}} : x' \notin \text{dom}(\sigma) \)
7. \( \langle \Box, \nu, \sigma \rangle_{\text{cont}} \rightarrow \langle \nu, \sigma \rangle_{\text{ans}} \)

The relation between RKNL and KL is described by a direct correspondence between the first six transitions of RKNL and transitions of KL. Intuitively, for \( i = 1, \ldots, 6 \), as long as a value is not reached, transition \( i \) of RKNL does the same as transition \( i \) of KL, only using different term representation. The only essential difference is that RKNL is environment based while KL is substitution based. However, substitutions in KL occur only in transition (6), where they are used to \( \alpha \)-rename a variable with a fresh one representing a location on the store; the same information is kept in environments in RKNL. Inessential differences are that RKNL uses stack to represent contexts (which are directly used in KL) and it stores some additional information not used by KL. In the rest of this section we formalize this correspondence.
Recall that weak call by need is a reduction strategy meant to evaluate only closed terms. For a given closed term \( t \) by the weak evaluation of \( t \) we mean the longest sequence of successive configurations of RKNL that starts with loading \( t \) and uses only transitions (1) to (6). We refer to configurations occurring in this sequence as to weak configurations.

Table 8 shows the translation \( \sigma^c \) of weak configurations of RKNL to configurations of KL. It is based on translations \( \sigma^v \) of closures, \( \sigma^s \) of stacks, \( \sigma^v \) of values and \( \sigma^s \) of stores.

| Configuration in RKNL | Configuration in KL |
|-----------------------|---------------------|
| \( t \)               | \( \ell \)          |
| \( \lambda x.t \)     | \( \ell := (\lambda x.t) \) |
| \( e \)               | \( e^c := e(x) \)   |
| \( (t_1, t_2, e) \)   | \( (t_1, t_2, e)^c := (t_1, t_2, e)^c \) |
| \( \sigma \)          | \( \sigma^c := \sigma^c \) |

The correctness of the translation follows from several observations:

- All values occurring in a weak evaluation are annotated lambda abstractions of the form \( \ell := (\lambda x.t) \). This is because free variables introduced by transition (4) do not occur in closed terms, and all other types of values are introduced in transitions (7), (10) and (11), which are not used in weak evaluations. Therefore, all values occurring in a weak evaluation can be translated with \( \sigma^v \).
- Frames \( \square \) and \( \lambda x.\square \) are introduced by transitions (9) and (7) and thus do not occur in weak evaluations. Therefore, all stacks occurring in a weak evaluation can be translated with \( \sigma^s \).
- The last configuration in a weak evaluation is a \( \triangle \)-configuration. This is because in every \( \nabla \)-configuration one of transitions (1)–(4) is fireable.
- The stack in the last configuration in a weak evaluation is empty. This is because with a nonempty stack a transition (5) or (6) is fireable. Therefore the translation of the last configuration in a weak evaluation is directly unloaded to an answer of KL.

The following lemma can be proved by an easy induction.

**Lemma 5.16.** If \( K \rightarrow K' \) is a transition in a weak evaluation of a closed term in RKNL, then \( \overline{K}^k \rightarrow \overline{K'}^k \) is a transition of KL.

In consequence, the evaluation of a closed term by KL simulates a weak evaluation of the same term by RKNL. Having in mind that both machines are deterministic and that both sequences stop at the same moment, this simulation is a bisimulation. Since weak evaluation is only an initial part of an evaluation by RKNL, we obtain that RKNL conservatively extends KL. In other words, the strategy realized by RKNL is a conservative extension of the call-by-need strategy.

### 6 CONCLUSION

We have presented a simple and efficient abstract machine for the Strong Call-by-Need strategy in the lambda calculus. The machine has been derived from a normalization-by-evaluation higher-order functional program by means of a series of off-the-shelf program transformation techniques, and it has been carried out in Racket, almost automatically. We proved the expected properties.
of the resulting machine: its soundness and completeness with respect to normal order, and its reasonability. Specifically, we proved that the number of steps in an execution is bilinear in the number of $\beta$-steps in the corresponding reduction sequence and in the size of the initial term. In the proofs we use a ghost abstract machine and potential-function techniques.

This work also confirms the versatility of the derivational approach to semantics; its additional benefit is the relative simplicity of the design and of reasoning about abstract machines. As future work, we plan to adapt the present methodology to deconstruct and find connections between other existing artefacts of strong call by need, in particular Crégut’s lazy KNL machine [Crégut 2007], a new, non-conservative strong call-by-need strategy proposed recently by Balabonski et al. [Balabonski et al. 2021], and Accattoli and Leberle’s study of useful sharing by explicit substitutions in call by need [Accattoli and Leberle 2022].

DATA AVAILABILITY STATEMENT
The code accompanying this paper is available at the Zenodo repository [Biernacka et al. 2022a].

ACKNOWLEDGMENTS
We would like to thank Dariusz Biernacki and the anonymous reviewers of the paper and of the software artefact for their helpful comments, Maciej Buszka for making seqt available to us for experimentation, Beniamino Accattoli for the encouragement to work on strong call-by-need strategy, and Tomasz Wierzbicki for teaching a course based on Chris Okasaki’s book [Okasaki 1999] at the University of Wroclaw.

REFERENCES
Beniamino Accattoli. 2016. The Useful MAM, a Reasonable Implementation of the Strong $\lambda$-Calculus. In Logic, Language, Information, and Computation - 23rd International Workshop, WoLLIC 2016, Puebla, Mexico, August 16-19th, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9803). Springer, 1–21. https://doi.org/10.1007/978-3-662-52921-8_1
Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP’17), Namur, Belgium, October 9 - 11, 2017, Wim Vanhoof and Brigitte Pientka (Eds.). ACM, 4–16. https://doi.org/10.1145/3131851.3131855
Beniamino Accattoli and Claudio Sacerdoti Coen. 2015. On the Relative Usefulness of Fireballs. In 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. IEEE Computer Society, 141–155. https://doi.org/10.1109/LICS.2015.23
Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. 2021. Strong Call-by-Value is Reasonable, Implosively. In 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470630
Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed, In Logical Methods in Computer Science. Logical Methods in Computer Science 12. https://doi.org/10.2168/LMCS-12(1:4)2016
Beniamino Accattoli and Giulio Guerrieri. 2019. Abstract machines for Open Call-by-Value. Sci. Comput. Program. 184 (2019). https://doi.org/10.1016/j.scico.2019.03.002
Beniamino Accattoli and Maico Leberle. 2022. Useful Open Call-By-Need. In 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference) (LIPIcs, Vol. 216), Florin Manea and Alex Simpson (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:21. https://doi.org/10.4230/LIPIcs.CSL.2022.4
Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden. ACM, 8–19. https://doi.org/10.1145/888251.888254
Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. Inf. Process. Lett. 90, 5 (2004), 223–232. https://doi.org/10.1016/j.ipl.2004.02.012 Extended version available as the research report BRICS RS-04-3.
Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The Call-by-Need Lambda Calculus. In Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 233–246. https://doi.org/10.1145/199448.199507
A Simple and Efficient Implementation of Strong Call by Need by an Abstract Machine

Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. 2017. Foundations of strong call by need. PACMPL 1, ICFP (2017), 20:1–20:29. https://doi.org/10.1145/3110264

Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. 2021. A Strong Call-By-Need Calculus. In 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (LIPIcs, Vol. 195), Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:22. https://doi.org/10.4230/LIPIcs.FSCD.2021.9

Pablo Barenbaum, Eduardo Bonelli, and Kareem Mohamed. 2018. Pattern Matching and Fixed Points: Resource Types and Strong Call-By-Need: Extended Abstract. In Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018, David Sabel and Peter Thiemann (Eds.). ACM, 6:1–6:12. https://doi.org/10.1145/3236950.3236972

Małgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. 2020. An Abstract Machine for Strong Call by Value. In Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020. Proceedings (Lecture Notes in Computer Science, Vol. 12470), Bruno C. d. S. Oliveira (Ed.). Springer, 147–166. https://doi.org/10.1007/978-3-030-64437-6_8

Małgorzata Biernacka and Witold Charatonik. 2019. Deriving an Abstract Machine for Strong Call by Need. In 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131), Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:20. https://doi.org/10.4230/LIPIcs.FSCD.2019.8

Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2021. A Derived Reasonable Abstract Machine for Strong Call by Value. In 23rd International Symposium on Principles and Practice of Declarative Programming (Tallinn, Estonia) (PPDP 2021). Association for Computing Machinery, New York, NY, USA, Article 6, 14 pages. https://doi.org/10.1145/3479394.3479401

Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022a. Abstract Machines Workshop. https://doi.org/10.5281/zenodo.6786796 (software artifact).

Małgorzata Biernacka, Witold Charatonik, and Klara Zielinska. 2017. Generalized Refocusing: From Hybrid Strategies to Abstract Machines. In 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs, Vol. 84), Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 10:1–10:17. https://doi.org/10.4230/LIPIcs.FSCD.2017.10

Małgorzata Biernacka and Olivier Danvy. 2007. A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines. Theor. Comput. Sci. 375, 1-3 (2007), 76–108. https://doi.org/10.1016/j.tcs.2006.12.028

Maciej Buszka and Dariusz Biernacki. 2021. Automating the Functional Correspondence Between Higher-Order Evaluators and Abstract Machines. In Logic-Based Program Synthesis and Transformation - 31st International Symposium, LOPSTR 2021, Tallinn, Estonia, September 7-8, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13290), Emanuele De Angelis and Wim Vanhoof (Eds.). Springer, 38–59. https://doi.org/10.1007/978-3-030-98869-2_3

Stephen Chang and Matthias Felleisen. 2012. The Call-by-Need Lambda Calculus, Revisited. In Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211), Helmut Seidl (Ed.). Springer, 128–147. https://doi.org/10.1007/978-3-642-28869-2_7

Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019, Ekaterina Komendantskaya (Ed.). ACM, 9:1–9:14. https://doi.org/10.1145/3354166.3354174

Pierre Crécut. 2007. Strongly reducing variants of the Krivine abstract machine. High. Order Symb. Comput. 20, 3 (2007), 209–230. https://doi.org/10.1007/s10990-007-9015-z A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming.

Olivier Danvy and Ian Zerny. 2013. A Synthetic Operational Account of Call-by-need Evaluation. In Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (Madrid, Spain) (PPDP ’13). ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2505879.2505898

Matthias Felleisen and Robert Heib. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. Theor. Comput. Sci. 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Andrzej Filinski and Henning Korsholm Rohde. 2005. Denotational aspects of upyped normalization by evaluation. RAIRO Theor. Informatics Appl. 39, 3 (2005), 423–453. https://doi.org/10.1051/ita:2005026 A preliminary version was presented at FOSSACS 2004.

Álvaro García-Pérez and Pablo Nogueira. 2014. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. Sci. Comput. Program. 95 (2014), 176–199. https://doi.org/10.1016/j.scico.2014.05.011
Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. J. Funct. Program. 2, 2 (1992), 127–202. https://doi.org/10.1017/S0956796800000319

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154. https://doi.org/10.1145/158511.158618

Johan Munk. 2007. A Study of Syntactic and Semantic Artifacts and its Application to Lambda Definability, Strong Normalization, and Weak Normalization in the Presence of State. Master’s thesis. DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark. https://www.brics.dk/RS/08/3/BRICS-RS-08-3.pdf BRICS research report RS-08-3.

Chris Okasaki. 1999. Purely functional data structures. Cambridge University Press. https://doi.org/10.1017/CBO9780511530104

Maciej Piróg and Dariusz Biernacki. 2010. A systematic derivation of the STG machine verified in Coq. In Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010, Jeremy Gibbons (Ed.). ACM, 25–36. https://doi.org/10.1145/1863523.1863528

Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. J. Funct. Program. 7, 3 (May 1997), 231–264. https://doi.org/10.1017/s0956796897002712