Strong Call by Value is Reasonable for Time

Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab
Institute of Computer Science, University of Wrocław, Poland
{mabi,wch,tdr}@cs.uni.wroc.pl
https://ii.uni.wroc.pl/~{mabi,wch,tdr}

Abstract. The invariance thesis of Slot and van Emde Boas states that all reasonable models of computation simulate each other with polynomially bounded overhead in time and constant-factor overhead in space. In this paper we show that a family of strong call-by-value strategies in the $\lambda$-calculus are reasonable for time. The proof is based on a construction of an appropriate abstract machine, systematically derived using Danvy et al.’s functional correspondence that connects higher-order interpreters with abstract-machine models by a well-established transformation technique. This is the first machine that implements a strong CbV strategy and simulates $\beta$-reduction with the overhead polynomial in the number of $\beta$-steps and in the size of the initial term. We prove this property using a form of amortized cost analysis à la Okasaki.

Keywords: $\lambda$-calculus · Abstract machines · Computational complexity · Reduction strategies · Normalization by evaluation.

1 Introduction

The invariance thesis of Slot and van Emde Boas [27] states that all reasonable models of computation simulate each other with polynomially bounded overhead in time and constant-factor overhead in space. For a long time it was not known whether there exist variants of the $\lambda$-calculus reasonable in this sense. In particular, it was not known whether there are evaluation strategies for the $\lambda$-calculus that can be simulated on Turing machines in time bounded by a polynomial of the number of performed $\beta$-reductions. Recently this question has been answered positively, for both time and space, with the weak call-by-value (CbV) [23] and, for time, with the strong call-by-name (CbN) [6] strategies. Here, by constructing an appropriate abstract machine, we show that a family of strong CbV strategies are reasonable for time.

It is well known that the $\lambda$-calculus provides a foundation for functional programming languages such as OCaml or Haskell, and more recently—for proof assistants such as Coq or ELF. A typical functional programming language uses a weak reduction strategy (e.g., call by value or call by need) that only reduces

* This research is supported by the National Science Centre of Poland, under grant number 2019/33/B/ST6/00289.
terms until a weak value is reached, and in particular it does not descend under lambda abstractions. On the other hand, proof assistants—which can be seen as functional languages with a rich system of dependent types—require a strong (i.e., full-reducing) reduction strategy for type checking. The study of strong normalization strategies in the $\lambda$-calculus, their properties and efficient implementation models, is therefore directly motivated by the need for efficient tools to handle large-scale, complex verification tasks carried out in such proof assistants [21,12].

Abstract machines provide implementation models for $\beta$-reduction that operationalize its key aspects: the process of finding a redex (decomposition) and the meta-operation of substitution ($\beta$-contraction). They are low-level devices, pretty close to Turing machines. An abstract machine is typically deterministic and it makes specific choices when searching for redices, i.e., it implements a specific reduction strategy of the calculus. Accattoli et al. in his work on the complexity of abstract machines [3,2,8] advocate the following notion of reasonability: a machine is called reasonable if it simulates the strategy with the time overhead bounded by a polynomial in the number of $\beta$-steps and in the size of the initial term.

The main technical goal of our work is to construct a reasonable (in the formal sense defined above) abstract machine for strong CbV. This has been a nontrivial [1,4] open problem (partial results include [7,8,13]) and an important step in the research program aimed at developing the complexity analysis of the Coq main abstract machine formulated by Accattoli [2]. Since there are known simulations of Turing machines in the lambda calculus and of abstract machines by Turing machines, the existence of such a machine implies reasonability (for time) of strong CbV in the sense of Slot and van Emde Boas.

**Related work.** In the case of weak reduction there is a vast body of work on abstract machines, their efficiency and construction techniques, but in the strong case the territory is largely uncharted. The first machine for strong normalization in the $\lambda$-calculus is due to Crégut and it implements Strong CbN [17]. A normalization function realizing strong CbV was proposed by Grégoire & Leroy and implemented in their virtual machine extending the ZAM machine [21]. Another virtual machine for strong CbV was derived by Ager et al. [10] from Aehlig and Joachimski’s normalization function [9]. Recently, a strong call-by-need strategy has been proposed by Kesner et al. [12], and the corresponding abstract machine has been derived by Biernacka et al. [14]. Finally, in a recent work [13], Biernacka et al. introduced the first abstract machine for strong CbV, and more specifically for the right-to-left strong CbV strategy. That machine has been derived by a series of systematic transformation steps from a standard normalization-by-evaluation function for the CbV lambda calculus. All three interpreters of strong CbV [10,13,21] are not reasonable in the sense defined above.

Our work builds on previous developments in the derivational approach to the construction and study of semantic artefacts, and in particular we use Danvy et al.’s functional correspondence and standard techniques used in functional programming [11]. The outcome of the derivation is an abstract machine whose
control stacks arise as defunctionalized continuations of the CPS-transformed evaluator. In the case of strong CbV (as well as other hybrid strategies) these stacks are not uniform; their structure corresponds directly to multiple-kindred reduction contexts and this correspondence can be described by a suitable shape invariant of the stacks. Hybrid strategies and their connection with machines have been studied by Garcia-Perez & Nogueira, and by Biernacka et al. [20, 15].

In between the weak and the strong CbV strategies, one can consider the Open CbV strategy that extends the usual weak CbV strategy in that it works on open terms and generalizes the notion of value accordingly. In a series of articles, Accattoli et al. study Open CbV in the form of so-called fireball calculus. They also show several abstract machines (the GLAM family) and study their complexity properties with the aim of providing reasonable and efficient implementations of Open CbV [8].

Contributions. In order to construct a reasonable abstract machine for strong CbV we refine the techniques from [13]. Our starting point is the same NbE evaluator as in [13] but we modify it using the standard memoization technique to store weak values together with their strong normal forms (if they get computed along the way). Next, we apply Danvy et al’s functional correspondence [11] to transform this new interpreter into a first-order abstract machine. This way the obtained machine is a systematically constructed semantic artefact rather than an ad-hoc modification of an existing one. The commented code of the development is available in [28].

In order to argue about the efficiency of the derived machine, we apply a form of amortized cost analysis that uses a potential function akin to Okasaki’s approach [25].

Thus, the contributions of this paper include:

1. a derivation of a reasonable abstract machine for strong CbV,
2. two variants of a reasonable machine: an environment-based one and a substitution-based one that uses delimited substitution,
3. a proof of correctness of the machine and of its reasonability,
4. a corollary that strong CbV is reasonable for time.

Outline. In Section 2 we introduce the basic concepts of the lambda calculus and the strong CbV strategy. In Section 3 we present the derivation of the machine starting from the NbE evaluator of the CbV lambda calculus. In Section 3.4 we present the resulting abstract machine with environments and in Section 4.2 its substitution-based variant. In Section 6 we prove the soundness of the machine and in Section 7 its reasonability. Section 8 concludes.

2 Preliminaries

2.1 Basics of λ-calculus

We work with pure lambda terms given by the following grammar:

\[
t ::= x \mid t_1 \ t_2 \mid \lambda x. \ t
\]
where $x$ ranges over some set of identifiers. As usual, in the sequel the use of a nonterminal restricts the range of this meta-variable (and its versions with primes or subscripts) to the defined family.

We define sets of free and bound variables in a term, and the substitution function as follows:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\lambda x. t) &= FV(t) \setminus \{x\}
\end{align*}
\]

\[
\begin{align*}
BV(x) &= \emptyset \\
BV(t_1 t_2) &= BV(t_1) \cup BV(t_2) \\
BV(\lambda x. t) &= BV(t) \cup \{x\}
\end{align*}
\]

\[
x'[x := t] = \begin{cases} 
  t & : x = x' \\
  x' & : x \neq x'
\end{cases}
\]

\[(t_1 t_2)[x := t] = t_1[x := t] t_2[x := t]\]

\[(\lambda x'. t')[x := t] = \begin{cases} 
  \lambda x', t' & : x = x' \\
  \lambda x', t'[x := t] & : x \neq x'
\end{cases}\]

This “raw” form of substitution makes it possible to capture free variable occurrences of a substituted term when it is substituted under lambda (the last case). In order to avoid this problem, it is standard to introduce $\alpha$-conversion to make sure that bound and free variables are distinct and the substitution is capture-avoiding.

We first define $\alpha$-contraction sufficient to rename bound variables:

\[
\frac{x' \notin FV(t) \cup BV(t)}{\lambda x. t \rightarrow^\alpha \lambda x'. t'[x := x']}
\]

In general, a contraction relation is local and can be lifted to a reduction relation defined on any term by contextual closure in the following way. A context is a term with exactly one free occurrence of a special variable $\square$ called hole. Assuming that $\square$ is not used as a bound variable, contexts in the lambda calculus can be defined by the following grammar:

\[
C ::= t C \mid C t \mid \lambda x. C \mid \square
\]

Now, for any contraction relation $\rightarrow$ and any context $C$ we define contextual closure as follows:

\[
\frac{t_1 \rightarrow t_2}{C[t_1] \xrightarrow{\rightarrow} C[t_2]}
\]

The notation $C[t]$ is a shortcut for $C[\square := t]$ and it denotes a term obtained by plugging the hole of the context with the given term. If the symbol above the arrow is omitted then the contraction can be done at any location in a term.

The reflexive-transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$ and the reflexive-symmetric-transitive closure is denoted by $=\vdash$, and is called conversion. Juxtaposition of two relations denotes their composition, e.g., $s \rightarrow_\beta =_\alpha t$ means that $\exists t'. s \rightarrow_\beta t' =_\alpha t$. 

Reduction semantics in the lambda calculus. Based on the ingredients introduced so far, we can define operational semantics for the lambda calculus in the form of reduction semantics that specifies $\beta$-contraction as the atomic computation step, and a reduction strategy that prescribes locations in a term where $\beta$-reduction can take place.

We define $\beta$-contraction in the standard way (and its contextual closure determines $\beta$-reduction):

$$
(\lambda x. t_1) t_2 \rightarrow^\beta t_1'[x := t_2]
$$

In order to define a specific strategy we need to restrict general contexts $C$. For the CbV strategy we also need to restrict $\beta$-contraction.

2.2 Call-by-value strategies

Weak reduction, i.e., reduction that does not ‘go under lambda’, can be defined as reduction in weak contexts $W$ defined by the following grammar:

$$
W ::= t W | W t | □
$$

In a call-by-value strategy, function arguments need to be evaluated before the function is applied. If only closed terms are considered, function arguments are reduced to lambda abstractions, but in the open case we need a more general notion of a weak normal form, which is a normal form of $W^\beta$. Weak normal forms $w$ can be expressed by the following grammar, where the auxiliary category $i$ denotes inert terms:

$$
w ::= \lambda x. t | i
$$

$$
i ::= i w | x
$$

To prevent the substitution of reducible terms we restrict $\beta$-contraction by requiring that the argument is a weak normal form:

$$
(\lambda x. t) w \rightarrow^\beta_w t'[w := w]
$$

This way we obtain the relation $W^\to_{\beta_w}$, which is exactly the reduction of the fireball calculus, where weak normal forms are called fireballs [7]. This calculus is nondeterministic but strongly confluent and hence all derivations to weak normal forms have the same length. It also restores the property (called harmony in [7]) that its normal forms are substitutable, i.e., there are no stuck terms.

The fireball reduction can be made deterministic by narrowing the space of possible evaluation contexts by only allowing to search for redices in the left part of an application when its right-hand side is a weak value:

$$
F ::= t F | F w | □
$$
The relation $F \rightarrow_{\beta_w}$ (which is $\beta_w$-contraction in contexts generated by $F$-contexts) is a restriction of $W \rightarrow_{\beta_w}$ to a right-to-left strategy, and it is a deterministic extension of the closed weak, right-to-left CbV strategy to the open case.

To obtain a conservative extension of a weak CbV strategy to a full-reducing one, we can just iterate reduction under lambdas after reaching a weak normal form. Such iterations of $W \rightarrow_{\beta_w}$ are also strongly confluent and therefore any such strategy is a good representative of the strong CbV strategy.

In this paper, we study the twice right-to-left call-by-value strategy (in short, rrCbV), that is an example of a deterministic extension of $F \rightarrow_{\beta_w}$ to a fully reducing strategy: it is deterministic [13], uses the same contraction relation, and $F$-contexts are a subset of the rrCbV context family. This strategy is denoted $R \rightarrow_{\beta_w}$ and its grammar of contexts can be defined using three context nonterminals $H$, $R$ and $F$ (the latter defined as above for the weak strategy):

\[
R ::= \lambda x. R \mid H \mid F \\
H ::= i R \mid H n
\]

where $n$ are normal forms of the strategy $\rightarrow_{\beta}$, and $a$ are neutral terms, both defined as follows:

\[
n ::= \lambda x. n \mid a \\
a ::= a n \mid x
\]

rrCbV is a hybrid strategy, i.e., it combines different substrategies: the $F$-contexts define the weak CbV substrategy, $R$ is the starting nonterminal defining the substrategy that either weakly reduces, or fully reduces weak normal forms using the auxiliary $H$-strategy, which in turn is responsible for fully reducing inert terms.

3 Construction of a Reasonable Machine

3.1 A Higher-Order Evaluator

Our starting point is the higher-order evaluator from [13] presented in Figure 1. It computes $\beta$-normal forms by following the principles of normalization by evaluation [19], where the idea is to map a $\lambda$-term to an object in the meta-language (here OCaml) from which a syntactic normal form of the input term can subsequently be read off.

1 The applicative order, i.e., leftmost-innermost reduction, where arguments of a function are directly reduced to a strong normal form, is not a conservative extension of weak CbV and therefore we do not consider it as a strong CbV strategy. Moreover, it does not support recursion [26] and may take a different number of steps than strong CbV strategies as defined here.

2 The right-to-left direction of evaluation in applications is traditional and used, e.g., in OCaml.
Strong Call by Value is Reasonable for Time

(* syntax of the lambda-calculus with de Bruijn indices *)

```ocaml
type index = int

type term = Var of index | Lam of term | App of term * term
```

(* semantic domain *)

```ocaml
type level = int

type sem = Abs of (sem -> sem) | Neutral of (level -> term)
```

(* reification of semantic objects into normal forms *)

```ocaml
let rec reify (d : sem) (m : level) : term =
  match d with
  | Abs f ->
    Lam (reify (f (Neutral (fun m' -> Var (m'-m-1))))(m+1))
  | Neutral l ->
    l m
```

(* sem -> sem as a retract of sem *)

```ocaml
let to_sem (f : sem -> sem) : sem = Abs f

let from_sem (d : sem) : sem -> sem =
  fun d' ->
    match d with
    | Abs f ->
      f d'
    | Neutral l ->
      Neutral (fun m -> let n = reify d' m in App (l m, n))
```

(* interpretation function *)

```ocaml
let rec eval (t : term) (e : sem list) : sem =
  match t with
  | Var n -> List.nth e n
  | Lam t' -> to_sem (fun d -> eval t' (d :: e))
  | App (t1, t2) -> let d2 = eval t2 e
    in from_sem (eval t1 e) d2
```

(* NbE: interpretation followed by reification *)

```ocaml
let nbe (t : term) : term = reify (eval t []) 0
```

**Fig. 1.** An OCaml implementation of the higher-order compositional evaluator from [13]: an instance of normalization by evaluation for a call-by-value β-reduction in the λ-calculus.

The evaluator works with lambda terms in de Bruijn notation. In this notation, lambda terms are generated by the grammar \( t ::= n \mid t_1 t_2 \mid \lambda t \) where \( n \) ranges over natural numbers. There are two flavours of the notation: de Bruijn indices, where \( n \) denotes the number of lambdas between the represented variable and its binder; and de Bruijn levels, where \( n \) denotes the number of
lambdas between the root of a term and the binder of the variable. For example, \( \lambda x. x (\lambda y. y x) \) is represented as \( \lambda 0 (\lambda 0 1) \) with de Bruijn indices and as \( \lambda 0 (\lambda 1 0) \) with levels. Using de Bruijn notation has a clear benefit of avoiding problems with \( \alpha \)-conversion, but there is some cost of having less intuitive \( \beta \)-reduction.

The normalization function first evaluates terms into the semantic domain represented by the recursive type \( \text{sem} \) – it is completely standard and implemented by the function \( \text{eval} \). Then the normal form is extracted from the semantic object by the function \( \text{reify} \) that mediates between syntax and semantics in the way known from Filinski and Rohde’s work [19] on NbE for the untyped \( \lambda \)-calculus.

Using Danvy et al.’s functional correspondence [11] between higher-order evaluators and abstract machines, the evaluator of Figure 1 is transformed in [13] to an abstract machine that implements a full-reducing call-by-value strategy for pure \( \lambda \)-calculus. The obtained machine is not reasonable in terms of complexity [8]: it cannot simulate \( n \) steps of \( \beta \)-reduction in a number of transitions that is polynomial in \( n \) and in the size of the initial term. The reason is that it never reuses constructed structures, so it has to introduce each constructor of the resulting normal form in a separate step. This can lead to an exponential blow-up, as the following example shows. Consider a family of terms

\[
\omega := \lambda x. x x \quad e_n := \lambda x. c_n \omega x
\]

where \( c_n \) denotes the \( n \)th Church numeral. Each \( e_n \) reduces to its normal form in the number of steps linear in \( n \), but the size of this normal form is exponential in \( n \).

Remark 1. The interpreters of [10,13,21] all suffer from exponential time overhead because of the size explosion problem.

### 3.2 A Pitfall of de Bruijn Indices and Levels

Consider term families defined as follows (here \( x, z \) are free variables):

\[
A_0 := x \quad B_0 := z \\
A_{n+} := A_n (\lambda y. y) x \quad B_{n+} := z \lambda w. \ B_n \\
Q_n := \lambda x. (\lambda z. \ B_n) \ A_n
\]

Note that there are \( n + 1 \) free occurrences of variable \( z \) in \( B_n \), each of which is under a different number of lambdas (on a different de Bruijn level). Terms of family \( Q \) are closed and they reach their normal forms in one \( \beta \)-reduction which substitutes \( A_n \) for \( z \) in \( B_n \). The size of \( A_n \) is linear in \( n \) and it is substituted for linearly many \( z \)s in \( B_n \) resulting in a normal form of quadratic size.

Terms \( A_n \) can be easily shared in memory and the resulting representation has size linear in \( n \). It is not however possible with de Bruijn indices nor levels...
representation because the resulting normal form has quadratically many constructors of distinct subterms. We illustrate this issue with the term $Q_2$ and its normal forms using 3 different representations: with names, indices and levels.

$$Q_2 = \lambda x. (\lambda z. \lambda w. z) (x (\lambda y. y) x (\lambda y. y) x)$$

$N_{\text{name}}(Q_2) = \lambda x. (x (\lambda y. y) x (\lambda y. y) x) (x (\lambda y. y) x (\lambda y. y) x) (x (\lambda y. y) x (\lambda y. y) x)

$N_{\text{index}}(Q_2) = \lambda \, (0 \, (\lambda 0) \, 0 \, (\lambda 0) \, 0) \, \lambda \, (1 \, (\lambda 0) \, 1 \, (\lambda 0) \, 1) \, \lambda \, (2 \, (\lambda 0) \, 2 \, (\lambda 0) \, 2)

$N_{\text{level}}(Q_2) = \lambda \, (0 \, (\lambda 1) \, 0 \, (\lambda 1) \, 0) \, \lambda \, (0 \, (\lambda 2) \, 0 \, (\lambda 2) \, 0) \, \lambda \, (0 \, (\lambda 3) \, 0 \, (\lambda 3) \, 0)

Here we have three occurrences of the term $A_2 = (x (\lambda y. y) x (\lambda y. y) x)$, each on a different de Bruijn level. Therefore, in the index notation, the subterm $x$ has a different representation in each of these occurrences. Similarly, in the level notation, the subterm $\lambda y. y$ has a different representation in each of these occurrences. In consequence, none of the two notations allows sharing of different occurrences of $A_2$. This example shows that when working with de Bruijn representations it is not possible to bound the amount of work per single $\beta$-reduction by a quantity proportional to the size of the initial term. It does not mean that machines working with these representations must be unreasonable but a proof involving this quadratic dependency may be more complex than one without this problem. Therefore, we choose to work with names.

### 3.3 A Reasonable Higher-Order Evaluator

Now we present a higher-order evaluator that will be transformed into an abstract machine. The implementation is given in OCaml [24]. It is a modified version of the evaluator from Figure 1, with three major changes. First, it uses names instead of de Bruijn indices to represent variables in lambda terms. Second, it abstracts the environments (the second argument of the function $\text{eval}$) in the sense that they are no longer directly implemented as lists, but as different data structures implementing dictionaries. Third, it uses caches as a form of sharing.

**Terms.** We start with the syntax of $\lambda$-terms with names:

```ocaml
type identifier = string
type term = Var of identifier
            | App of term * term
            | Lam of identifier * term
```

**Environments.** Environments are dictionaries storing values assigned to identifiers. To handle open terms an environment returns abstract variables for undefined identifiers (with the same name) and we make sure they will not be captured during abstraction reification. Here we extend the name in order to mark that the variable is free and then free variables of the initial term are replaced with variables with the extended name in the resulting term.
module Dict = Map.Make(
  struct type t = identifier let compare = compare end)

type env = sem Dict.t

let rec env_lookup (x : identifier) (e : env) : sem =
  match Dict.find_opt x e with
  | Some v -> v
  | None    -> abstract_variable (x ^ "_free")

Caches. To achieve a reasonable implementation for strong CbV we need to introduce a form of sharing in order to avoid the size explosion problem.

We employ a mechanism similar to memothunks. This allows us to reuse already computed subterms in normal forms. An $\alpha$-cache is a place where a result of type $\alpha$ can be stored and later used to prevent invoking the same delayed computation many times. It is implemented as follows:

type 'a cache = 'a option ref

let cached_call (c : 'a cache) (t : unit -> 'a) : 'a =
  match !c with
  | Some y -> y
  | None   -> let y = t () in c := Some y; y

Values. In the original strong CbV evaluator there are two kinds of values: abstractions and delayed neutral terms which after defunctionalization correspond to weak normal forms and inert terms, respectively. Here we add another constructor to allow annotation of values with caches for their normal forms.

type sem = Abs of (sem -> sem)
  | Neutral of (unit -> term)
  | Cache of term cache * sem

Reification. In normalization by evaluation reification plays a role of a read back of concrete syntactic objects from abstract semantic values. It corresponds to full normalization of weak normal forms in the obtained machine.

Here reification uses two auxiliary functions. Fresh names are generated in a standard way using one extra memory cell. In the representation with names, abstract variables are delayed neutral terms which just return a variable for a given identifier.

let gensym : unit -> int =
  let c = ref 0 in
  fun () ->
let res = !c in
  c := res + 1;
res

let abstract_variable (x : identifier) : value =
  let vx = Var x in
  Neutral (fun () -> vx)

The reification of abstractions and neutral terms is accomplished just as in the original evaluator: abstractions are called with an abstract variable with a freshly generated name, and the result is reified under lambda with the same name; delayed neutral terms are simply forced. In the case of values with cache lookup, Cache (c,v), if the result of reification of v is known, it is simply read from the cache c; otherwise it is computed and stored in the cache.

let rec reify : sem -> term =
  function
  | Abs f ->
    let xm = "x_" ^ string_of_int (gensym ()) in
    Lam (xm, reify (f @@ abstract_variable xm))
  | Neutral l ->
    l ()
  | Cache (c, v) -> cached_call c (fun () -> reify v)

Value application. Values as elements of a λ-calculus model should also play role of endofunctions on themselves. Abstractions are wrappings of such functions so it is enough to unwrap them. A neutral term applied to a normal form creates a new neutral term, and it can be constructed by delaying the forcing of this neutral term and the reification of the argument. However, if a neutral term is annotated with cache, the cache should be consulted when the delayed computation is forced. In contrast, application does not normalize abstractions (abstractions are changed by reduction) and therefore their caches are ignored in such a situation.

let rec from_sem : sem -> (sem -> sem) =
  function
  | Abs f -> f
  | Neutral l -> apply_neutral l
  | Cache (c, Neutral l) -> apply_neutral
    (fun () -> cached_call c l)
  | Cache (c, v) -> from_sem v
and apply_neutral (l : unit -> term) (v : sem) : sem =
  Neutral (fun () -> let n = reify v in App (l (), n))

Evaluation. Evaluation uses an auxiliary function that annotates a value with the empty cache, provided it is not already annotated. Evaluation reads a λ-
expression as source code. Source variables merely indicate values in environment. Source abstractions are translated to abstraction values that evaluate their bodies with environments extended by an argument annotated with cache. Applications are evaluated right-to-left and the left value is applied to the right one as described earlier.

```ocaml
let mount_cache (v:sem) : sem =
    match v with
    | Cache(_,_) -> v
    | _ -> Cache(ref None, v)

let rec eval (t : term) (e : env) : sem =
    match t with
    | Var x -> env_lookup x e
    | Lam (x, t') -> to_sem (fun v -> eval t' @@ Dict.add x (mount_cache v) e)
    | App (t1, t2) -> let v2 = eval t2 e
                      in from_sem (eval t1 e) v2
```

The normalization-by-evaluation function is the composition of evaluation in the empty environment and reification.

```ocaml
let nbe (t : term) : term = reify (eval t Dict.empty)
```

### 3.4 Abstract Machine

Using Danvy et al.’s functional correspondence [11], the evaluator constructed in Section 3.3 is transformed to an abstract machine. The most important steps in this transformation are the same as on the path from the evaluator in Figure 1 to its corresponding abstract machine in [13]: closure conversion, transformation to continuation passing style, defunctionalization of continuations to stacks, entanglement of defunctionalized form to an abstract machine. All these transformations are described in the supplementary materials [28]. The machine obtained by derivation is presented in Figures 2 and 3.

**Values.** Machine representations of values are representations of weak normal forms that can additionally be annotated with heap locations. These locations are used to cache full normal forms. The grammar presented here is a bit counter-intuitive as it allows nesting of locations like \((v^ℓ)^{ℓ'}\) or applications of closures to other values. The machine maintains invariants guaranteeing that there are no nested locations and that all values are decoded to weak normal forms (in particular, all applications involve inert terms). It is possible to write a more precise grammar here; however, this would lead to many new syntactic categories and to an increase in the number of transitions of the resulting machine.
Strong Call by Value is Reasonable for Time

Identifiers $\ni x$

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

Locations $\ni \ell$

Values $\ni v ::= V(x) \mid v_1 v_2 \mid [x, t, E] \mid v^{\ell}$

Envs $\ni E ::= \text{Identifiers} \rightarrow \text{Values}$

Frames $\ni F ::= [t, E] \square \mid \square v \mid v \square \mid \square t \mid \lambda x. \square \mid @[\ell]$

Stacks $\ni S ::= \bullet \mid F ::= S$

Term Optionals $\ni t^? ::= \bullet \mid [t]$

Heaps $\ni H ::= \text{Locations} \rightarrow \text{Term Optionals}$

Counters $\ni m \in \mathbb{N}$

Confs $\ni K ::= \langle t, E, S, m, H \rangle_{\mathcal{E}} \mid \langle S, v, m, H \rangle_{\mathcal{C}} \mid \langle S, t, m, H \rangle_{S} \mid \langle t^?, \ell, S, v, m, H \rangle_{M}$

Fig. 2. Syntactic categories used in the environment-based machine

**Environments.** Environments are dictionaries whose keys are identifiers and values are annotated machine values of the form $v^{\ell}$. They represent assignments of values (weak normal forms) to variables. They can be implemented as association lists (as it is done explicitly in [13]); other, more efficient options are considered in Section 7.4.

The content of the initial environment init corresponds to the result of lookup in the higher-order evaluator when nothing is assigned to any variable. We represent it by the empty collection, but as stated before, the initial environment in our implementation returns an abstract variable $V(x_{\text{free}})$ for variable $x$.

**Stack and Frames.** Stacks are machine representations of contexts; technically they are just sequences of frames. The first five frames in the grammar of $F$ are the same as in the KNV machine of [13]. They are used in representations of rCrB contexts, maintaining the same invariants as KNV; we discuss it in Section 6.2. The last frame, $@[\ell]$, has similar meaning to an extra frame in Crégut’s KL and KNL of [17]: it is used to cache a computed normal form under location $\ell$. Here we use heap to indicate explicitly which structures of the machine have to be mutable in an implementation.

The stack is the only mechanism responsible for managing the continuation; in particular the machine has no component that could be recognized as a dump which is present for example in [8].

**Counter.** The machine has a counter that is stored in every configuration and is not duplicated. It can be seen as a register. Its role is to generate fresh names for abstract variables (see transition (9) in Figure 3) and it is only incremented.
It could also be decremented in rule (18) to maintain de Bruijn level as it is done in KN and KNV, but then the freshness of variables would be less obvious.

**Heap.** The heap can be seen as a dictionary whose keys are locations and (optional) values are terms in normal form. As it is the case with the counter, the heap appears in every configuration exactly once and hence it can be implemented in the RAM model as mutable memory. Then locations \( \ell \) can be seen as pointers to such memory locations. In the purely functional setting this can be simulated with a dictionary, causing logarithmic overhead. However, even with the use of mutable state, a configuration of the machine can be seen as a persistent data structure. This is because every mutable location is used only to memoize the normal form of a determined value: when the normal form is computed, the stored value never changes. No mutable pointers are stored in the heap. Therefore one can say that every pointer points to a lower level in the pointing hierarchy. Thus no reference cycles are created and garbage collection can rely solely on reference counting.

In terms of \([2]\) the heap can be recognized as a global environment. It is introduced in the evaluator and preserved by the derivation, so it is present in the machine. Moreover, together with the local environment used in the machine, it can be seen as a derived split environment, because every value in the local environment has to be annotated with a location coming from a distinguished set of identifiers pointing to the global environment.

**Configurations.** The machine uses four kinds of configurations corresponding to four modes of operation: in \(E\)-configurations the machine evaluates some subterm to a weak normal form; in \(C\)-configurations it continues with a computed weak normal form and in \(S\)-configurations it continues with a computed strong normal form. \(M\)-configurations are used to manipulate access to memory.

**Transitions.** The transitions of the machine are presented in Figure 3. The first one loads an input term to the initial configuration; similarly the last one unloads the computed strong normal form from the final configuration.

Transitions (1)–(3) are completely standard. In order to evaluate an application of terms, transition (1) calls the evaluation of the argument and pushes the current context represented by a closure pairing the calling function with the current environment to the stack. Note that this implements the right-to-left choice of the order of evaluation of arguments. A lambda abstraction in (2) is already a weak normal form, so we simply change the mode of operation to a \(C\)-configuration. Transition (3) simply reads a value of a variable from the environment (which always returns a wnf) and changes the mode of operation.

Configurations of the form \(\langle S, v, m, H \rangle_C\) continue with a wnf \(v\) in a context represented by stack \(S\), with heap \(H\) and the value of the variable counter equal to \(m\). There are two goals in these configurations: the first is to finish the evaluation (to wnf's) of the closures stored on the stack \(S\) according to the weak
call-by-value strategy; the second is to reduce \( v \) to a strong normal form. This is handled by rules (4)–(12), where rules (4)–(8) are responsible for the first goal, and rules (9)–(12) for the second. The choice of the executed transition is done by pattern-matching on \( v \) and \( S \); we always choose the first matching transition. This means, in particular, that \( \square v^\ell \) is a right application of an annotated value in transition (5), but in transition (6) we have a right application of a value that is not annotated.

In rule (4) the stack contains a closure, so we start evaluating this closure and push the already computed wnf to the stack; when this evaluation reaches a wnf, rules (6) or (7) applies. Rule (6) is responsible for an application of a not-annotated abstraction closure to a not-annotated wnf; in this case a new location \( \ell \) is created on the heap and the wnf is annotated with \( \ell \) (later the computed normal form of the wnf may be stored in \( \ell \)). Rule (5) is responsible for an application of a not-annotated lambda abstraction to an annotated wnf and implements \( \beta \)-contraction by evaluating the body of the abstraction in the
appropriately extended environment. Rule (7) is responsible for an application of an annotated abstraction; since $\beta$-contraction is going to change the normal form of the computed term, we simply remove the information about the cache; this rule is immediately followed by transition (5) or (6) and (5). Rule (8) applies when $i$ is an inert term and $v$ is an arbitrary (annotated or not) wnf; in this case we reconstruct the application of this inert term to the wnf popped from the stack (which gives another wnf).

Rules (9)—(12) are applied when there are no more wnfs on the top of the stack; here we pattern-match on the currently processed wnf $v$. If it is a closure, transition (9) implements the ‘going under lambda’ step: it pushes the elementary context $\lambda x. \Box$ to the stack (note that $x$ is a fresh variable here), increments the variable counter, creates a new location $\ell$ on the heap, adds the (annotated) abstract variable $V(x_m)^\ell$ to the environment, and starts the evaluation of the body. If $v$ is an abstract variable, we reach a normal form; rule (10) changes the mode of operation to a $S$-configuration. If $v$ is an application $i v$, rule (11) delays the normalization of $i$ by pushing it to the stack and continues with $v$; note that this implements the second of our right-to-left choices of the order of reduction. Finally, if $v$ is an annotated value, we change the mode of operation to an $M$-configuration to check if the normal form has already been computed.

In configuration $\langle t^? \ell, S, v, m, H \rangle_M$ heap $H$ contains location $\ell$ pointing to $t^?$ which is an optional of $v$’s strong normal form. If it contains the normal form $n$ then transition (13) immediately returns $n$ and changes the mode of operation to an $S$-configuration. Otherwise, if it is empty, rule (14) pushes the location $\ell$ to the stack and calls the normalization of $v$. When this normalization is finished, rule (15) stores the computed normal form at location $\ell$.

Configurations of the form $\langle S, t, m, H \rangle_S$ continue with a (strong) normal form $t$ in a context represented by $S$. The goal in these configurations is to finish the evaluation of inert terms stored on the stack and to reconstruct the final term. This is handled by transitions (15)—(18); the choice of the transition is done by pattern-matching on the stack. If there is an inert term $v$ on the top of the stack, rule (16) pushes the already computed normal form on the stack and calls normalization of $v$ by switching the mode of operation to a $C$-configuration. Otherwise there is a previously computed normal form or a $\lambda x. \Box$ frame on the top of the stack; in these cases transitions (17) and (18) reconstruct the term accordingly. Finally, when the stack is empty, the machine stops and unloads the final result from a configuration.

Remark 2. The machine works on a term representation that allows sharing, as in OCaml or in [16]. In particular, caching of the computed normal forms allows their reuse in the construction of the output term. For example, exponentially big normal forms of the family $e_n = \lambda x. c_n \omega x$ consume only a linear in $n$ amount of memory and are computed in linear time. We assume that unloading of the final result does not involve unfolding it to the unshared term representation (which might require exponential time).
4 A Substitution-Based Machine

The machine from Section 3.4 uses environments to represent delayed substitutions. In this section we replace environments with delimited substitutions. This variant provides an intermediate step between machine configurations and their decodings, and makes correctness proofs in Section 6 easier to follow.

4.1 From variables as pointers to delimited substitution

Accattoli and Barras in [2] present a technique that represents variables as pointers and enables substitution and lookup of a substituted value in constant time. They use it to obtain an efficient version of Milner Abstract Machine. Here we apply this technique to eliminate local environments.

The technique relies on a modified representation of terms, where identifiers in abstractions and variables are replaced by pointers to mutable memory. With this representation, a substitution can be performed in constant time by simply writing to an appropriate memory cell; similarly the lookup of a variable in an environment boils down to a single reading operation. However, by overwriting a variable, the substitution destroys the original term, which, in consequence, cannot be shared. In order to avoid incorrect modification of shared terms, some subterms must be copied. In [8] it is observed that a new copy is needed only when it comes to $\beta$-reduction: only bodies of $\lambda$-abstractions must be copied and the only identifier that needs to point to a fresh memory cell is the one of the abstraction to be applied. Intuitively, this corresponds to an $\alpha$-renaming of a $\lambda$-abstraction before a $\beta$-reduction step. It is called renaming on $\beta$.

In the next section we fuse copying a term and overwrite-based substitution into a substitution function working on purely functional term representation. The tricky bit is that the copying function does not copy already substituted variables but shares them instead. To preserve this property, we use substitution delimiters of the form $\langle v \rangle$ informing that there are no occurrences of the substituted variable in subterm $v$ and that $v$ can be shared.

4.2 A substitution-based machine

The new machine is presented in Figures 4 and 5. The transitions in Figures 3 and 5 are in one-to-one correspondence and the two machines bisimulate each other. The style of the presentation of the new machine is more implicit, e.g., the mechanism of fresh name generation and the heap component, though implicitly present, are not spelled out in configurations. Some new notation is used in Figure 5: $x^*$ denotes a fresh variable, $H(\ell)$ is the content of location $\ell$ on the implicit heap $H$ and $[\ell := \cdot]$ is an update of location $\ell$.

The syntax of terms is extended with a substitution delimiter $\langle \rangle$ carrying substituted values. It is the only change in term representation; input terms do not have to be compiled before being loaded to the machine because they fall within the extended grammar. Output terms are free of substitution delimiters.
Every configuration of the environment-based machine can be translated into a substitution-based one by executing delayed substitution for free variables and denoting it with substitution delimiters. The main part of this translation, i.e., translation of closures $\cdot$, is given below, where $E \setminus [x \mapsto -]$ denotes the environment $E$ with removed binding for $x$. This establishes a strong bisimulation between the machines.

$$x, E = \begin{cases} \langle E(x) \rangle & : x \in E \\ x & : x \notin E \end{cases}$$

$$t_1 \ t_2, E = t_1, E \ t_2, E$$

$$\lambda x. t, E = \lambda x. t, E \setminus [x \mapsto -]$$

Syntax:

- $x \in \text{Identifiers}$
- $t \in \text{Locations}$
- $t ::= x \ | \ t_1 \ t_2 \ | \ \lambda x. t \ | \ \langle v \rangle$
- $v ::= V(x) \ | \ v_1 \ v_2 \ | \ [x, t] \ | \ v^f$
- $F ::= t \ \square \ | \ \square v \ | \ [v \ \square] \ | \ [\lambda x. \ | \ @[\ell]]$
- $S ::= \bullet \ | \ F :: S$
- $t^\ell ::= \bullet \ | \ [\ell]$
- $K ::= \langle t, S \rangle_E \ | \ \langle S, v \rangle_C \ | \ \langle S, t \rangle_S \ | \ \langle t^\ell, \ell, S, v \rangle_M$

Substitutions:

$$x'[x := t] = \begin{cases} t : x = x' \\ x' : x \neq x_1 \end{cases}$$

$$t_1 \ t_2[x := t] = t_1[x := t] \ t_2[x := t]$$

$$(\lambda x'. t')[x := t] = \begin{cases} \lambda x'. t' : x = x' \\ \lambda x'. t'[x := t] : x \neq x' \end{cases}$$

$$(\langle v \rangle)[x := t] = \langle v \rangle$$

(strip$(x)$) = $V(x)$

(strip$(\langle v \rangle)$) = $v$

Fig. 4. Syntactic categories and substitutions used in the substitution-based machine.
5 A substitution-based higher-order evaluator

Using functional correspondence between evaluators and abstract machines it is also possible to derive a higher-order evaluator corresponding to the substitution-based machine. Here we present a fragment of this evaluator. The first pattern matching reveals that the type of terms is extended with the constructor for substitution delimiters.

\[
\begin{align*}
  t & \mapsto \langle t, \bullet \rangle_c \\
  \langle t_1, t_2, S \rangle_c & \mapsto \langle t_2, t_1 \square :: S \rangle_c \\
  \langle \lambda x. t, S \rangle_c & \mapsto \langle S, [x, t] \rangle_c \\
  \langle t, S \rangle_c & \mapsto \langle S, \text{strip}(t) \rangle_c \\
  \langle t \square :: S, v \rangle_c & \mapsto \langle t, \square v :: S \rangle_c \\
  \square v :: S, [x, t] & \mapsto \langle t[x := \langle v' \rangle], S \rangle_c \\
  \square v :: S, [x, t] & \mapsto \langle \square v' :: S, [x, t] \rangle_c \\
  \square v :: S, [x, t] & \mapsto \langle \square v :: S, [x, t] \rangle_c \\
  \square v :: S, [x, t] & \mapsto \langle \square v :: S, [x, t] \rangle_c \\
  \langle S, [x, t] \rangle_c & \mapsto \langle t[x := \langle V(x') \rangle], \lambda x^* \cdot \square :: S \rangle_c \\
  \langle S, V(x) \rangle_c & \mapsto \langle S, x \rangle_S \\
  \langle S, v_1, v_2 \rangle_c & \mapsto \langle v_1, \square :: S, v_2 \rangle_c \\
  \langle S, v' \rangle_c & \mapsto \langle H(t), \ell, S, v \rangle_M \\
  \langle \ell, t, S, v \rangle_M & \mapsto \langle S, t \rangle_S \\
  \langle \bullet, \ell, S, v \rangle_M & \mapsto \langle \emptyset[\ell] :: S, v \rangle_c \\
  \langle \emptyset[\ell] :: S, t \rangle_S & \mapsto \langle S, t \rangle_S[t := [\ell]] \\
  \langle v \square :: S, t \rangle_S & \mapsto \langle \square t :: S, v \rangle_c \\
  \langle \square t_1 :: S, t_2 \rangle_S & \mapsto \langle S, t_1, t_2 \rangle_S \\
  \langle \lambda x. \square :: S, t \rangle_S & \mapsto \langle S, \lambda x. t \rangle_S \\
  \langle \bullet, t \rangle_S & \mapsto t
\end{align*}
\]

Fig. 5. Transitions of the substitution-based machine
fun v -> eval (subst x (Subs (mount_cache v)) t')
| App (t1, t2) -> let v2 = eval t2
            in from_sem (eval t1) v2
| x        -> env_lookup x

6 Soundness of the machine

The soundness of the obtained machines with respect to the rrCbV strategy could be argued by reasoning starting from the soundness of KNV \[13\] and showing its preservation by the subsequent program transformations. Instead, we present a sketch of a direct proof.

6.1 Decoding of the machine

Below we define a decoding of stacks to contexts and of machine terms, values and configurations to source terms.

\[
\begin{align*}
[t_1, t_2]_k &= [t_1]_t, [t_2]_t \\
[\lambda x. t]_k &= \lambda x. [t]_t \\
[x]_k &= x \\
[v_1, v_2]_v &= [v_1]_v [v_2]_v \\
[[v, t]]_v &= \lambda x. [t]_v \\
[V(x)]_v &= x \\
[v']_v &= [v]_v
\end{align*}
\]

\[
\begin{align*}
\langle t, S \rangle_E^K &= [S]_S[[t]_t] \\
\langle S, v \rangle_C^K &= [S]_S[[v]_v] \\
\langle t', \ell, S, v \rangle_M^K &= [S]_S[[v]_v] \\
\langle S, t \rangle_S^K &= [S]_S[[t]_t]
\end{align*}
\]

6.2 Shape invariants

We state more precise shape invariants to assert that evaluation contexts and values follow the rules of the rrCbV strategy. Such invariants can be derived from evaluators explicitly expressing their own invariants. In the grammars below numerical subscripts will also discriminate grammar symbols. The syntactic categories \( n \) and \( a \) of normal forms and neutral terms are those defined in Section 2.2.

\[
\begin{align*}
v_w ::= [x, t] & \mid v_i & \mid v_w^\ell & \mid n^\ell \\
v_i ::= V(x) & \mid v_i v_w & \mid v_i^\ell \\
n^\ell ::= \bullet & \mid [a] \mid [a]
\end{align*}
\]
Lemma 1. All reachable configurations are well-formed, i.e., are in forms: \((t, S_1)_\mathcal{E}\), \((S_1, v_w)_\mathcal{C}\), \((S_2, v_i)_\mathcal{C}\), \((n', \ell, S_3, v_w)_\mathcal{M}\), \((a', \ell, S_2, v_i)_\mathcal{M}\), \((S_2, a)_S\), \((S_3, n)_S\).

Proof (idea). The initial configuration is well-formed and is preserved by all transitions.

Corollary 1. If \(S\) is a reachable stack of the machine then context \([S]_S\) is a \(rrCbV\) context.

Proof (sketch). In [13] it is shown that all \(rrCbV\) contexts are generated by the (outside-in) grammar of contexts from Section 2.2, with the starting symbol \(R\). It is also shown that this grammar is equivalent to the following (inside-out) grammar with the starting symbol \(S_1\):

\[
S_1 ::= S_1[t \Box] \mid S_1[\Box w] \mid S_3
S_2 ::= S_2[\Box n] \mid S_3
S_3 ::= S_2[i \Box] \mid S_3[\lambda x. \Box] \mid \Box
\]

Decodings of well-formed stacks follow this grammar.

Corollary 2. If \(v\) is a reachable value of the machine then term \([v]_V\) is a weak normal form.

The second corollary states one of the invariants mentioned in Subsection 3.4.

To capture the second fact that annotations \(\ell\) cannot be stacked, a more precise shape invariant could be established. This, however, would require more grammar symbols and well-formed configurations.

6.3 Interpretation of transitions

To omit some technical details we focus on the machine soundness for closed input terms. It is sufficient because open terms can be closed by abstractions before processing.

Lemma 2. If \(K \overset{\iota}{\to} K', \iota \notin \{5, 9, 13\}\) and term \([K]_K\) is closed then \([K]_K = [K']_K\).

Proof. By case analysis on transition rules.

Lemma 3. If \(K\) is a reachable configuration, term \([K]_K\) is closed and \(K \overset{5}{\to} K'\) then \([K]_K \overset{R}{\to} [K']_K\).
Proof (sketch). Transitions maintain the invariant that all free variables of terms under delimiters are bound by the stack. Hence they are not captured during substitution, and substitution can be delimited: the substituted variable does not occur under delimiter and \( \beta \)-contraction is simulated properly. From Corollaries 1 and 2 it follows that an evaluation context is an \( R \)-context and a substituted value decodes to a weak normal form.

Lemma 4. If \( K \) is a reachable configuration, term \( \llbracket K \rrbracket_K \) is closed and \( K \rightarrow (9) K' \) then \( \llbracket K \rrbracket_K \rightarrow_{\alpha} [K']_K \).

Proof (sketch). Thanks to the fresh variable \( \alpha \)-contraction is simulated correctly. As in Lemma 3, the substituted variable (here \( x \)) does not occur under delimiters and the free variable \( x^* \) is bound by the stack.

Lemma 5 (bypass). If \( K \) is a reachable configuration, term \( \llbracket K \rrbracket_K \) is closed and \( K \rightarrow (13) K' \) then \( \llbracket K \rrbracket_K \rightarrow_{R : \beta_w} =_{\alpha} [K']_K \).

Proof (idea). A normal form can be memoized only by getting off a \( @[\ell] \) frame by transition (15). After pushing it on the stack by transition (14) the only way to do that is to compute the full normal form of a given weak normal form. Thus, if machine had used transition (14) instead of (13) it would have maintained the shape invariant, the evaluation context would be still a \( R \)-context, and the computed full normal form would be \( \alpha \)-equivalent. By standard properties of \( \alpha \)-conversion its uses in transition (9) can be postponed.

Proposition 1. If \( K \) is a reachable configuration, term \( \llbracket K \rrbracket_K \) is closed and \( K \rightarrow K' \) then \( \llbracket K \rrbracket_K \rightarrow_{R : \beta_w} =_{\alpha} [K']_K \).

Proof. This is an immediate consequence of Lemmas 2–5.

Theorem 1 (soundness). If machine starting from \( t_0 \) computes \( t \) (i.e., \( \langle t_0, \bullet \rangle \rightarrow (\bullet, t) \mathcal{S} \)), then \( t \) reduces in many steps to a normal form \( t' \) (i.e., there exists \( t' \) such that \( t_0 \rightarrow (R : \beta_w) t' \) and \( t' \rightarrow (R : \beta_w) t =_{\alpha} t' \)).

Proof. By Lemma 1 terminal configuration decodes to a term in normal form, so \( t \) is in normal form. By Proposition 1 \( t_0 \) reduces to \( t \).

7 Complexity Analysis and Completeness of the Machine

7.1 The conversion problem

One of the motivations for constructing an efficient machine for strong CbV comes from the conversion problem, which asks if two given terms are \( \beta \)-convertible. In general the problem is undecidable, but it has important applications in proof assistants and thus it is desirable to find efficient partial solutions.

If both input terms normalize in the strong CbV strategy, then one straightforward solution is to normalize input terms by the abstract machine and then
to check if they are $\alpha$-equivalent. Normalization yields a shared representation of terms, avoiding possibly exponential size of the normal forms in explicit representation. As Condoluci et al. state in [16], the $\alpha$-equivalence of shared terms can be checked in time linear w.r.t. the size of shared representations. Therefore convertibility check can be done in time proportional to the time of normalization of both terms. In the following we analyse the cost of normalization.

In [13] it is shown that, using a technique called streaming of terms, the convertibility check can be short-circuited if partial results of normalization differ. We do not consider it here as fusion of term streaming and shared equality goes beyond the scope of this paper.

### 7.2 Execution length

Probably the simplest approach to the complexity analysis of an abstract machine is to find a (constant) upper bound on the number of consecutive administrative steps of the machine. Then the overall complexity of an execution is this bound times the number of $\beta$-transitions times the cost of a single transition.

In this section we estimate the global number of transitions executed by the machine on a given term. Unfortunately, the simple approach outlined above does not work here as the following example shows. Let $c_n$ be the $n^{th}$ Church numeral, $dub := \lambda x. \lambda p. p\ x\ x$ and $I$ – the identity. The execution of $c_n\ dub\ I$ starts with two $\beta$-reductions substituting $dub$ and $I$. Then next $n$ reductions result in a value $r_n$, where $r_0 = [x, x]$ and $r_{n+} = [p, p\ \langle r_n^\ell \rangle\ \langle r_n^{\ell'} \rangle]$ for some $\ell$s. Value $r_n$ decodes to a normal form, but it takes the machine $\Omega(n)$ administrative steps to construct this normal form. Thus, sequences of administrative transitions are not bounded by any constant (see also Fig. 6).

To overcome the problem of long sequences of administrative steps we perform a kind of amortized analysis of the execution length. Following [25], we define a potential function $\Phi_K$ of configurations. Then, for most of the transitions (more precisely, for all but $\overset{7}{\longrightarrow}$) the cost of the transition is covered by the change in the potential of the involved configurations (cf. Lemma 8). Transition $\overset{7}{\longrightarrow}$ is a preparatory step for a $\beta$-reduction and its cost is covered by the involved $\beta$-reduction (cf. Lemma 10).

The potential of a configuration depends on the potentials of its components: term, value, stack and the implicit heap. The potential functions $\Phi_t$, $\Phi_v$ and $\Phi_S$ are defined as follows.
\[ \Phi_t(t_1, t_2) = 6 + \Phi_t(t_1) + \Phi_t(t_2) \quad \Phi_S(\bullet) = 0 \]

\[ \Phi_t(\lambda x. t) = 4 + \Phi_t(t) \quad \Phi_S(t \Box :: S) = 5 + \Phi_S(S) + \Phi_t(t) \]

\[ \Phi_t(x) = 4 \quad \Phi_S(\Box v :: S) = 4 + \Phi_S(S) + \Phi_v(v) \]

\[ \Phi_t(\nu) = 4 \quad \Phi_S(v \Box :: S) = 2 + \Phi_S(S) + \Phi_v(v) \]

\[ \Phi_v(v_1, v_2) = 3 + \Phi_v(v_1) + \Phi_v(v_2) \quad \Phi_S(\Box t :: S) = 1 + \Phi_S(S) \]

\[ \Phi_v([x, t]) = 3 + \Phi_t(t) \quad \Phi_S(\nu \Box :: S) = 1 + \Phi_S(S) \]

\[ \Phi_v(V(x)) = 1 \quad \Phi_S(\nu \Box :: S) = 1 + \Phi_S(S) \]

\[ \Phi_v(v^\ell) = 3 \pm \Phi_v(\nu^\ell) \]

Intuitively, these potential functions indicate for how many machine steps a given construct is responsible. For example, \( \Phi_t(t_1, t_2) \) says that if an application \( t_1, t_2 \) appears somewhere in a configuration, it may generate 6 transitions of the machine plus the work generated by the two subterms. The crucial observation is that when a normal form of a value \( v \) is known and stored under location \( \ell \), then the normalization of \( v \) involves only a constant (more precisely, 2) steps and it does not involve recomputation of \( v \) – thus the amount of work generated by \( v^\ell \) is bounded by 3 (one transition involves memoizing the normal form).

The potential function \( \Phi_H \) estimates the cost of maintaining the heap (which is implicitly present in each configuration). It takes into account all values that have their place on the heap (expressed by the condition \( v^\ell \in K \) below, meaning that \( v^\ell \) appears somewhere in the current configuration), but are not yet normalized (expressed by \( H(\ell) = \bullet \) and are currently not being evaluated (expressed by \( \Box \ell \notin S \), meaning that \( \Box \ell \) does not appear in the stack and thus the evaluation of \( v \) has not yet started). It also takes into account the moment when the evaluation of \( v^\ell \) starts, i.e., when the current configuration is of the form \( (\bullet, \ell, S, v)_M \). Formally, \( \Phi_H \) is defined as follows:

\[ \Phi_H(K) = \sum_{(\ell, v) \text{ s.t. } K = (\bullet, \ell, S, v)_M \vee (v^\ell \in K \land H(\ell) = \bullet \land \Box \ell \notin S)} \Phi_v(v) \]

Now we define the potential function \( \Phi_K \) for configurations. We use Iverson bracket in the clause for \( C \)-configuration (\( [\varphi] = 1 \) if \( \varphi \) and \( [\varphi] = 0 \) otherwise) to denote advancement of transition (6):

\[ \Phi_K(\langle t, S \rangle_C) = \Phi_t(t) + \Phi_S(S) + \Phi_H(K) \]

\[ \Phi_K(\langle S, v \rangle_C) = \Phi_v(v) + \Phi_S(S) + \Phi_H(K) - 9 \cdot [\langle S, v \rangle_C \rightarrow] \]

\[ \Phi_K(\langle t^2, \ell, S, v \rangle_M) = 2 + \Phi_S(S) + \Phi_H(K) \]

\[ \Phi_K(\langle S, t \rangle_S) = \Phi_S(S) + \Phi_H(K) \]

**Lemma 6.** If substitution delimiter \( \langle v \rangle \) occurs somewhere in a reachable configuration then it is of the form \( \langle v^\ell \rangle \).

**Proof.** The only transitions introducing substitution delimiters are (5) and (9) which ensure location annotation.
Lemma 7. For any \( t, x, v \) we have \( \Phi(t) = \Phi(t[x := \langle v \rangle]) \).

Proof. The only constructors that can be replaced by a substitution are variables and \( \Phi_t(x) = \Phi_t(\langle v \rangle) \).

Lemma 8 (decrease). If \( K \) is a reachable configuration and \( K \overset{7}{\rightarrow} K' \) then \( \Phi(K) > \Phi(K') \).

Proof (of Lemma 8). The proof goes by case analysis on machine transitions.

1. \( \Phi(t_1 t_2 S) = 6 + \Phi(t_1) + \Phi(t_2) + \Phi(S) + \Phi_H(K) \)
   \[ \geq \Phi(t_1) + \Phi(t_2) + \Phi(S) + \Phi_H(K) \]
   \[ = \Phi(\langle t_2, t_1 \rangle) + \Phi(S) + \Phi_H(K) \]

2. \( \Phi(\langle x. t, S \rangle) = 4 + \Phi(t) + \Phi(S) + \Phi_H(K) \)
   \[ > 3 + \Phi(t) + \Phi(S) + \Phi_H(K) - 9 \cdot |\langle S, x, t \rangle| \]
   \[ = \Phi(\langle S, x, t \rangle) \]

3. Here, by Lemma 6, term \( t \) is of the form either \( x \) or \( \langle v^f \rangle \), so \( \text{strip}(t) \) is either \( V(x) \) or \( v^f \) and thus
   \[ \Phi(\langle t, S \rangle) = 4 + \Phi(S) + \Phi_H(K) \]
   \[ > 3 + \Phi(S) + \Phi_H(K) \]
   \[ \geq \Phi(\text{strip}(t)) + \Phi(S) + \Phi_H(K) - 9 \cdot |\langle S, v \rangle| \]
   \[ = \Phi(\langle S, \text{strip}(t) \rangle) \]

4. \( \Phi(\langle t \rangle) = \Phi(t) + 5 + \Phi(S) + \Phi_H(K) \)
   \[ > \Phi(t) + 4 + \Phi(S) + \Phi_H(K) \]
   \[ = \Phi(\langle t \rangle) \]

5. \( \Phi(\langle v^f : S, x, t \rangle) = \Phi(v^f) + \Phi(S) + \Phi_H(K) - 9 \)
   \[ > \Phi(v^f) + \Phi(S) + \Phi_H(K) - 9 \]
   \[ = \Phi(v^f) + \Phi(S) + \Phi_H(K) + 1 \]

6. This is an easy case: 4 occurring in \( \Phi_S(\langle v \rangle) \) is greater than 3 occurring in \( \Phi(v_1 v_2) \).
(9) \( \Phi_K((S, [x, t]_C) = \Phi_v([x, t]) + \Phi_S(S) + \Phi_H(K) - 0 \\
= 3 + \Phi_v(t) + \Phi_S(S) + \Phi_H(K) \\
> \Phi_v(t) + 1 + \Phi_S(S) + \Phi_v(V(x^*)) + \Phi_H(K) \\
\text{Lemma 7} \Rightarrow \Phi_v(t[x := \langle V(x^*) \rangle]) + \Phi_S(\lambda x^* . \square :: S) + \Phi_H(K') \\
= \Phi_K((t[x := \langle V(x^*) \rangle], \lambda x^* . \square :: S)_C) \\
(10) \text{This is an easy case.} \\
(11) \text{This is an easy case.} \\
(12) \Phi_K((S, v^\ell)_C) = 3 + \Phi_S(S) + \Phi_H(K) \\
> 2 + \Phi_S(S) + \Phi_H(K) \\
= \Phi_K((H(\ell), \ell, S, v)_M) \\
(13) \Phi_K((v^\ell, \ell, S, v)_M) = 2 + \Phi_S(S) + \Phi_H(K) \\
> \Phi_S(S) + \Phi_H(K) \\
= \Phi_K((S, t)_S) \\
(14) \Phi_K((\bullet, \ell, S, v)_M) = 2 + \Phi_S(S) + \Phi_H(K) \\
> 1 + \Phi_S(S) + \Phi_H(K) \\
= \Phi_v(v) + 1 + \Phi_S(S) + \Phi_H(K') \\
= \Phi_K((\bullet[\ell] :: S, v)_C) \\
(15) \text{The pair } (\ell, v) \text{ is not counted before the transition because } \bullet[\ell] \text{ is on the stack, and it is not counted after the transition because } H(\ell) \neq \bullet. \\
\Phi_K((\bullet[\ell] :: S, t)_S) = 1 + \Phi_S(S) + \Phi_H(K) \\
> \Phi_S(S) + \Phi_H(K) \\
= \Phi_K((S, t)_S) \\
(16) \text{This is an easy case: 2 occurring in } \Phi_S(v \square :: S) \text{ is greater than 1 occurring in } \Phi_S(\square t :: S). \text{ The value counted on the stack before the transition is counted in the configuration after the transition.} \\
(17) \text{This is an easy case.} \\
(18) \text{This is an easy case.} \\

\textbf{Lemma 9 (subterm).} If \([x, t]\) is a reachable value of the machine starting from term } t_0 \text{ then } \Phi_v([x, t]) < \Phi_v(t_0). \\

\textbf{Proof.} Both machines never perform reductions in bodies of abstractions which will be invoked later. In environment-based machine for all values \([x, t, E]\) terms \(\lambda x. t\) are always subterms of } t_0. \text{ By translation in substitution-based machine these abstractions may be modified only by replacing source variables with values under substitution delimiters and by Lemma 7 it does not change the potential. Function } \Phi_v \text{ assigns 4 to abstraction constructor which is greater than 3 assigned by } \Phi_v.
**Lemma 10 (increase).** If $K$ is a reachable configuration from $t_0$ and $K \xrightarrow{7} K'$ then $\Phi_K(K') - \Phi_K(K) < \Phi_t(t_0)$.

**Proof.**

\[
\Phi_K([\square v :: S, [x, t])_{\mathcal{C}} + \Phi_t(t_0) = \Phi_v([x, t]) + \Phi_S(\square v :: S) + \Phi_H(K) - 0 + \Phi_t(t_0)
\]

\[
= \Phi_t(t_0) + 3 + \Phi_S(\square v :: S) + \Phi_H(K) - 0
\]

by Lemma 9

\[
\Phi_v([x, t]) + \Phi_S(\square v :: S) + \Phi_H(K') - 9 \cdot [\square v :: S, [x, t])_{\mathcal{C}} \xrightarrow{5}
\]

\[
= \Phi_K([\square v :: S, [x, t])_{\mathcal{C}}
\]

Example changes of potential are depicted in Figures 6 and 7 using Matplotlib [22].

![Figure 6: Plot of potential for execution of $c_0$ dub $I$ performing 217 transitions of which 8 are $\beta$-transitions (5)](image)

**Lemma 11.** Let $\rho$ be a sequence of consecutive machine transitions starting from term $t_0$, $|\rho|$ be number of steps in $\rho$ and $|\rho|_{(7)}$ be number of steps (7) in $\rho$. Then $|\rho| \leq (|\rho|_{(7)} + 1) \cdot \Phi_t(t_0)$.

**Proof.** This is an immediate consequence of Lemmas 8 and 10.
7.3 Completeness

The upper bound on a length of machine trace from Lemma 11 leads to the completeness of the machine.

**Proposition 2.** If $K$ is a reachable configuration, term $\llbracket K \rrbracket_K$ is closed and $\llbracket K \rrbracket_K \xrightarrow{R} t'$, then there exists $K'$ such that $K \xrightarrow{R} K'$ and $t' \xrightarrow{\beta_w} \alpha \llbracket K' \rrbracket_K$.

**Proof.** By Lemma 1 and inspection of all transitions, configurations that decode to a term not in normal form are non-terminal. Since the decoding of the given configuration $\beta$-reduces, by Proposition 1 the machine makes steps until (5) or (13) is performed. By Lemma 8 the number of consecutive transitions $\neq (7)$ is bounded by the potential. Since transition (7) must be followed by (5) in one or two steps, (5) or (13) must be reached eventually. In both cases Proposition 1 guarantees that, up to $\alpha$-conversion, the same term is reached or bypassed.

**Theorem 2 (Completeness).** If $t_0$ reduces in many steps to a normal form $t$ (i.e., $t_0 \xrightarrow{R} t$ and $t \not\xrightarrow{R} \beta_w$), then machine starting from $t_0$ computes $t$ (i.e., there exists $t'$ such that $(t_0, \bullet) \xrightarrow{E} (\bullet, t')$ and $t = \alpha t'$).

**Proof.** By Lemma 11 the machine reaches a terminal configuration in finite number of steps. By Lemma 1 the terminal configuration decodes to a term in normal form and by Proposition 2 this normal form is $\alpha$-equivalent to $t$. 
7.4 Implementation of environments

The environment-based machine performs insert (in rules (5) and (9)) and lookup (in rule (3)) operations on environments. When environments are implemented as lists, then the cost of lookup is proportional to the size of the list.

Every environment is paired with a subterm of the initial term constituting a closure. The machine maintains an invariant that the size of any environment is equal to the number of lambda abstractions under which the paired subterm is located in the original term.

Accattoli and Barras in [2] outline two realizations of local environments which improve the asymptotic cost of operations. One of them uses balanced trees, which are a natural choice for named representation of lambda expressions. The other uses random-access lists – this requires precomputing de Bruijn indices of variables in the original term. Both these data structures are persistent, both improve the cost of lookup to logarithmic in the size of the initial term.

We also note that if identifiers are strings of symbols from a finite alphabet (as binary numbers are strings of bits) then tries can be applied as dictionaries making lookup and update costs proportional to the length of the involved identifier.

7.5 Overall complexity

We treat arithmetic operations and operations on identifiers as realizable in constant time. In fact, they involve extra logarithmic cost but it does not affect the polynomial complexity. As described in Subsection 3.4, heap operations can be implemented within constant time.

Let $n$ be the number of $\beta$-reductions performed in a derivation from term $t_0$. It can be simulated on both machines with $O((1 + n) \cdot |t_0|)$ transitions.

The environment-based machine has three transitions ((3), (5) and (9)) whose cost is related to the environment; all the other transitions have constant cost. Therefore the overall cost of the execution is $O((1 + n) \cdot |t_0| \cdot E(|t_0|))$ where $E(|t_0|)$ is the cost of an operation on an environment of size $|t_0|$ and this cost can be considered as logarithmic because an environment maps variables occurring in the input term to values and number of such variables is bounded by $|t_0|$.

In the substitution-based machine transitions (5) and (9) have cost proportional to $|t_0|$ while the cost of all the other transitions is constant. Therefore the overall cost of the execution is $O((1 + n) \cdot |t_0|^2)$. However, in the weak case, i.e., before going under $\lambda$ with transition (9), the machine performs $O((1 + n) \cdot |t_0|)$ transitions with unitary cost and $O(n)$ transitions (5) of cost $O(|t_0|)$. Therefore the cost of the weak part of the execution is bilinear, i.e., $O((1 + n) \cdot |t_0|)$.

Accattoli and Dal Lago, in [5,18], present a polynomial simulation of Turing Machines relevant for any strategy reducing weak redexes first which is also true for rrCbV. We have shown a polynomial simulation of rrCbV strategy. Therefore reasonable machines and the rrCbV strategy can simulate each other with a polynomial overhead making the latter also a reasonable machine for time.
Call-by-value strategies as described in the preliminaries perform the same number of $\beta$-reductions to achieve normal form so this result generalizes to all of them:

**Theorem 3.** The number of steps performed by a strong call-by-value strategy is a reasonable measure for time.

8 Conclusion and future work

We presented an abstract machine that realizes the strong CbV strategy (in its right-to-left variant) and we proved its reasonability that makes it a sufficiently good implementation model. The machine uses a form of memoization to store computed normal forms and reuse them when needed. A derivation from an evaluator using memothunks also in weak normalization would lead to a machine performing some kind of strong call-by-need strategy but its study and comparison with other call-by-need machines are beyond the scope of this paper.

References

1. Accattoli, B.: A fresh look at the lambda-calculus (invited talk). In: 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019. LIPIcs, vol. 131, pp. 1:1–1:20 (2019)
2. Accattoli, B., Barras, B.: Environments and the complexity of abstract machines. In: Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP'17). pp. 4–16 (2017)
3. Accattoli, B., Coen, C.S.: On the relative usefulness of fireballs. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015. pp. 141–155 (2015)
4. Accattoli, B., Condoluci, A., Guerrieri, G., Coen, C.S.: Crumbling abstract machines. In: Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019. pp. 4:1–4:15 (2019)
5. Accattoli, B., Dal Lago, U.: On the invariance of the unitary cost model for head reduction. In: 23rd International Conference on Rewriting Techniques and Applications, RTA 2012. LIPIcs, vol. 15, pp. 22–37 (2012)
6. Accattoli, B., Lago, U.D.: (Leftmost-outermost) beta reduction is invariant, indeed. Logical Methods in Computer Science **12** (2016)
7. Accattoli, B., Guerrieri, G.: Open call-by-value. In: 14th Asian Symposium, APLAS 2016. Proceedings. LNCS, vol. 10017, pp. 206–226 (2016)
8. Accattoli, B., Guerrieri, G.: Abstract machines for open call-by-value. Sci. Comput. Program. **184** (2019)
9. Aehlig, K., Joachimski, F.: Operational aspects of untyped normalization by evaluation. Mathematical Structures in Computer Science **14**, 587–611 (2004)
10. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From interpreter to compiler and virtual machine: a functional derivation. Tech. Rep. BRICS RS-03-14, DAIMI, Aarhus University, Aarhus, Denmark (Mar 2003)
11. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the Fifth ACM-SIGPLAN Conference, PPDP’03. pp. 8–19 (2003)
12. Balabonski, T., Barenbaum, P., Bonelli, E., Kesner, D.: Foundations of strong call by need. PACMPL 1(ICFP), 20:1–20:29 (2017)
13. Biernacka, M., Biernacki, D., Charatonik, W., Drab, T.: An abstract machine for strong call by value. In: Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Proceedings. LNCS, vol. 12470, pp. 147–166 (2020)
14. Biernacka, M., Charatonik, W.: Deriving an abstract machine for strong call by need. In: 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019. LIPIcs, vol. 131, pp. 8:1–8:20 (2019)
15. Biernacka, M., Charatonik, W., Zielinska, K.: Generalized refocusing: From hybrid strategies to abstract machines. In: 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017. pp. 10:1–10:17 (2017)
16. Condoluci, A., Accattoli, B., Coen, C.S.: Sharing equality is linear. In: Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019. pp. 9:1–9:14 (2019)
17. Crégut, P.: Strongly reducing variants of the Krivine abstract machine. Higher-Order and Symbolic Computation 20(3), 209–230 (2007)
18. Dal Lago, U., Accattoli, B.: Encoding Turing machines into the deterministic lambda-calculus. CoRR abs/1711.10078 (2017)
19. Filinski, A., Rohde, H.K.: Denotational aspects of untyped normalization by evaluation. Theoretical Informatics and Applications 39(3), 423–453 (2005)
20. García-Pérez, A., Nogueira, P.: On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. Science of Computer Programming 95, 176–199 (2014)
21. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: International Conference on Functional Programming. pp. 235–246. SIGPLAN Notices 37(9) (2002)
22. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in Science & Engineering 9(3), 90–95 (2007)
23. Lago, U.D., Martini, S.: The weak lambda calculus as a reasonable machine. Theor. Comput. Sci. 398(1-3), 32–50 (2008)
24. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system, release 4.10. INRIA, Rocquencourt, France (Feb 2020)
25. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
26. Sestoft, P.: Demonstrating lambda calculus reduction. In: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 420–435. No. 2566 in Lecture Notes in Computer Science (2002)
27. Slot, C.F., van Emde Boas, P.: On tape versus core; an application of space efficient perfect hash functions to the invariance of space. In: Proceedings of the 16th Annual ACM Symposium on Theory of Computing. pp. 391–400. ACM (1984)
28. https://www.ii.uni.wroc.pl/~tdr/pub/rscbv.zip