Deriving an Abstract Machine for Strong Call by Need

Małgorzata Biernacka
Institute of Computer Science, University of Wrocław, Poland
mabi@cs.uni.wroc.pl

Witold Charatonik
Institute of Computer Science, University of Wrocław, Poland
wch@cs.uni.wroc.pl

Abstract

Strong call by need is a reduction strategy for computing strong normal forms in the lambda calculus, where terms are fully normalized inside the bodies of lambda abstractions and open terms are allowed. As typical for a call-by-need strategy, the arguments of a function call are evaluated at most once, only when they are needed. This strategy has been introduced recently by Balabonski et al., who proved it complete with respect to full $\beta$-reduction and conservative over weak call by need.

We show a novel reduction semantics and the first abstract machine for the strong call-by-need strategy. The reduction semantics incorporates syntactic distinction between strict and non-strict let constructs and is geared towards an efficient implementation. It has been defined within the framework of generalized refocusing, i.e., a generic method that allows to go from a reduction semantics instrumented with context kinds to the corresponding abstract machine; the machine is thus correct by construction. The format of the semantics that we use makes it explicit that strong call by need is an example of a hybrid strategy with an infinite number of substrategies.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Operational semantics

Keywords and phrases abstract machines, reduction semantics

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.8

Funding This research is supported by the National Science Centre of Poland, under grant number 2014/15/B/ST6/00619.

1 Introduction

Call by need, or lazy evaluation, is a strategy to evaluate $\lambda$-terms based on two principles: first, evaluation of an expression should be delayed until its value is needed; second, the arguments of a function call should be evaluated at most once [32]. Lazy evaluation can be considered as an optimization of call-by-name evaluation where the computation of arguments is delayed but its results are not reused. A model implementation of lazy evaluation is often given in the form of an abstract machine which typically includes a store to facilitate memoization of intermediate results, as in the well-known STG machine of Peyton Jones used in the Haskell compiler [26]. On the other hand, theoretical studies of lazy evaluation stem from two canonical approaches: a store-based natural semantics of Launchbury [24, 28], and a storeless, purely syntactic account of Ariola et al. [8, 25, 17, 21]. Lazy evaluation is an example of a strategy realizing weak reduction, which is standard in functional programming languages, where all $\lambda$-abstractions are considered to be values; consequently, evaluation of a term always stops after reducing it to a $\lambda$-abstraction. In contrast, strong reduction continues to reduce inside the bodies of $\lambda$-abstractions until a full $\beta$-normal form is reached; consequently, it must reduce inside substitutions and it must be able to evaluate open terms. Strong reduction and the corresponding reduction strategies have been gaining more
attention due to the development of proof assistants based on dependent types, such as Agda or Coq, whose implementation requires full term normalization for type-checking [23, 14]. In particular, the current version of the Coq proof assistant [30] employs an abstract machine that uses a lazy strategy to fully normalize terms, but the strategy has not been studied formally. Recently, Balabonski et al. proposed a strong call-by-need reduction strategy as a theoretical foundation for implementations of strong call by need [10]. Their strategy is proved to be complete with respect to $\beta$-reduction in the $\lambda$-calculus and conservative over the weak call-by-need strategy. Even though it has good theoretical properties, it is not clear how the strategy can be efficiently implemented, because it lacks operational account.

The goal of this paper is twofold: to contribute to the study of strong call by need by presenting a novel reduction semantics and an abstract machine for the strategy described in [10], and to showcase the existing framework of generalized refocusing used to inter-derive the two, quite complex, semantic artefacts. To this end, we first give a proper formulation of reduction semantics for the strategy, one that fits the framework and thus directly enables its implementation, and which can be seen as an operationalized variant of Balabonski et al.’s semantics. Second, we derive an abstract machine from the strategy, by means of the refocusing procedure [12] which takes as input a reduction semantics satisfying mild syntactic conditions and produces a lower-level specification that is provably correct with respect to the reduction semantics. The procedure is implemented in Coq and is fully automatic, once the syntactic conditions are proved.

The reduction semantics we present is inspired by previous work on weak [9, 19] and strong [10] call-by-need strategies. In particular, it builds on the concepts from [10] and incorporates syntactic distinction between strict and non-strict let constructs from [19], and it does not introduce an explicit store. However, in contrast to [10], we avoid collecting non-local information in the process of decomposition of terms (such as traversing a term to identify all its needed variables), but rather we thread the required information throughout. We prove that our version of the reduction semantics is adequate with respect to that from [10]; the formal correspondence between the two can be found in Section 6. The reduction semantics is presented in a format recently developed in [12], based on contexts instrumented with kinds that carry extra information. This format makes it explicit that strong call by need is an example of a hybrid strategy with an infinite number of substrategies (which equals the number of nonterminal symbols in the grammar).

Since the strong call-by-need semantics is quite sophisticated, as a warm-up we show a reduction semantics and an abstract machine for weak call by need, which is much simpler and presented in the same framework of generalized refocusing.

The rest of the paper is organized as follows. In Section 2 we recall the semantic formats that we use throughout the paper. In Section 3 we show the reduction semantics for the weak call-by-need strategy due to Danvy and Zerny, and we derive the corresponding abstract machine in the framework of generalized refocusing. In Section 4 we present a novel formulation of a reduction semantics for the strong call-by-need strategy, in Section 5 we discuss the derived abstract machine for this strategy. In Section 6 we show that our semantics is equivalent to that of Balabonski et al. In Section 7 we discuss the closest related work, and we conclude in Section 8.
a variable called a hole and denoted □. For a given term \( t \) and a context \( C \), by \( C[t] \) we denote the result of plugging \( t \) into \( C \), i.e., the term obtained by substituting the hole in \( C \) with \( t \). We say that a pair \( \langle C, t \rangle \) is a decomposition of the term \( C[t] \) into the context \( C \) and the term \( t \). The set of reduction contexts in a reduction semantics is often defined in form of a grammar of contexts [18]. A contraction relation \( \rightarrow \) of a reduction semantics specifies atomic computation steps, typically given by a set of rewriting rules. Terms that can be rewritten by \( \rightarrow \) are called redices and those produced by it – contracta. In a reduction semantics, the reduction relation \( \rightarrow \) is defined as the compatible closure of the contraction: a term \( t \) reduces in one step to \( t' \) if it can be decomposed into a redex \( r \) in an evaluation context \( E \), that is \( t = E[r] \), the redex \( r \) can be rewritten in one step to \( t'' \) by one of the contraction rules, and \( t' \) is obtained by the recomposition of \( E \) and \( t'' \), that is \( t' = E[t''] \). In Section 4 we use a more general definition of reduction that accounts for more complex strategies.

When considering programs as closed terms (i.e., terms without free variables), evaluation is defined as the reflexive-transitive closure of the reduction relation (written \( \rightarrow^* \)), and values are expected results of computation, chosen from the set of all normal forms (other normal forms are often called stuck terms). All \( \lambda \)-abstractions are typically considered values and evaluation does not enter \( \lambda \)-bodies. More generally, and when we consider reduction of open terms, we often use the term normalization instead of evaluation, and normal form rather than value. In this paper, we consider open terms and we use these terms interchangeably (effectively, we treat all normal forms as values).

A grammar of contexts consists of a set \( N \) of nonterminal symbols, a starting nonterminal, a set \( S \) of variables denoting syntactic categories and a set of productions. Productions have the form \( C \rightarrow \tau \) where \( C \in N \) and \( \tau \) is a term with free variables in \( N \cup \{□\} \cup S \), with exactly one occurrence of a variable from \( N \cup \{□\} \). If the grammar of reduction contexts encoding a strategy contains just one nonterminal symbol, we call this strategy uniform; intuitively, one always proceeds in the same way when decomposing a term into a reduction context and a redex. On the other hand, strategies that require multiple nonterminals in grammars are hybrid: it is necessary to use different substrategies for finding redices, one substrategy for each nonterminal symbol.

Figure 1 contains an example of a grammar with two nonterminals \( C, E \) (where \( C \) is the starting nonterminal) and syntactic categories \( t, n \) and \( v \) of terms, neutral terms and values. Figure 2 contains an example of a grammar with one nonterminal \( E \) and syntactic categories \( t, v \) of terms and values, and \( E[x] \) is a succinct notation for the category of needy terms, i.e., terms decomposable into a variable in context (an explicit definition of needy terms will be given later in Figure 4).

<table>
<thead>
<tr>
<th>Production</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>( □_C \mid λx.C \mid E t \mid nC ) where ( t := λx.t \mid x \mid t t ), ( n := x \mid n v ), ( v := n \mid λx.v )</td>
</tr>
<tr>
<td>( E )</td>
<td>( □_E \mid E t \mid nC )</td>
</tr>
</tbody>
</table>

(\( β \)-contraction) \( (λx.t_1) t_2 \rightarrow t_1[x\rightarrow t_2] \)

\[ Figure 1 \] Normal-order reduction semantics from [12].

Figure 1 shows the normal-order strategy, which is a hybrid strategy. This strategy normalizes a term to its full \( β \)-normal form (if it exists) by first evaluating it to its weak-head normal form with the call-by-name strategy, and only then reducing subterms of the resulting weak-head normal form with the same normal-order strategy. There are two substrategies,
An Abstract Machine for Strong Call by Need

one for each nonterminal symbol in the grammar. The $E$ substrategy corresponds to call by name and reduces to weak-head normal form; the $C$ substrategy allows reduction in bodies of $\lambda$-abstractions and in arguments to neutral terms. Each substrategy comes with its own kind of hole – the subscript indicates the kind of context that can be built inside the hole. In other words, if we want to extend a reduction context with a hole of kind $k$ by plugging another context in it, this new context has to be derivable from the nonterminal $k$. The contraction rule (standard $\beta$-reduction) is here common to both substrategies, but in Section 4 we show that the contraction relation may be parameterized by kinds.

Abstract machines abound in the literature. They may serve as theoretical artefacts that facilitate reasoning about the strategy and the execution of programs, or provide the basis for further transformations and optimizations in a principled way, possibly using known, off-the-shelf techniques. Intuitively, abstract machines are just another form of operational semantics, only defined at a lower level of abstraction. They typically provide reasonably precise and efficient models of implementation. Ideally, each step of an abstract machine should be done in constant time, which usually can be achieved by rewriting only topmost symbols of machine configurations. Therefore complex operations such as finding a redex in a term or substituting a term for a variable in another term should be divided into smaller steps, making explicit the process of decomposition of terms. It is nontrivial how to achieve this atomicity when the strategy requires non-local information to proceed – one typically needs to thread some extra information in machine configurations. The strong call-by-need strategy is an example where this happens and in this paper we show how we solve this problem for finding redices. However, we do not deal with the decomposition of the contraction rules into atomic steps, which is an orthogonal issue. In fact, decomposition of contraction can also be handled by the refocusing methodology, as witnessed by previous work deriving environment-based abstract machines from calculi of closures by refocusing [13]. We leave it for future work.

3 Weak call by need

As a gentle introduction we first review the simpler weak call-by-need strategy and discuss some of the concepts we later extend to the strong case. We also summarize the main idea behind the refocusing procedure that allows to derive an abstract machine from a reduction semantics.

3.1 Reduction strategy

There is a wide range of theoretical studies of lazy evaluation in the $\lambda$-calculus, presenting different semantic formats of the weak call-by-need strategy. Here we recall one: Danvy and Zerny’s “revised storeless reduction semantics”, which is most relevant to our work and can be seen as the basis for our strong variant of the call-by-need reduction semantics. This strategy was derived in [19] from the standard call-by-need reduction for the $\lambda_{let}$ calculus common to Ariola, Felleisen, Maraist, Odersky and Wadler [8, 9, 25] as part of a bigger picture connecting the various approaches to the weak call-by-need strategy.

The strategy is presented in Figure 2. The grammar of terms extends lambda terms with two forms of let-constructors declaring denotables. A strict let expression $\text{let } x := t_0 \text{ in } t_1$ makes it syntactically explicit that the variable $x$ is needed in the let-body $t_1$ and its value is still not known. Informally, we say that the variable $x$ is needed in $t_1$ whenever the first thing to do when evaluating $t_1$ is to establish the value of $x$, i.e., when $t_1$ can be uniquely decomposed as $t_1 = E[x]$, where $E$ is a reduction context. In a non-strict version
Syntax:

(t)\( t ::= x \mid \lambda x.t \mid tt \mid \text{let } x = t \text{ in } t \mid \text{let } x := t \text{ in } E[x] \)

(\lambda\)-values: \( v ::= \lambda x.t \)

(constructs) \( A ::= \Box \mid \text{let } x = t \text{ in } A \)

(evaluation contexts) \( E ::= \Box \mid Et \mid \text{let } x = t \text{ in } E \mid \text{let } x := E \text{ in } E[x] \)

(answer contexts) \( r ::= A[v]t \mid \text{let } x := A[v] \text{ in } E[x] \mid \text{let } x = t \text{ in } E[x] \)

Contraction rules:

1. \( A[\lambda x.t]t_1 \rightarrow A[\text{let } x = t_1 \text{ in } t] \)
2. \( \text{let } x = t \text{ in } E[x] \rightarrow \text{let } x := t \text{ in } E[x] \)
3. \( \text{let } x := A[v] \text{ in } E[x] \rightarrow A[\text{let } x = v \text{ in } E[v]] \)

\& Figure 2 The revised call-by-need \( \lambda \text{let} \) calculus from [19].

\textbf{let } x = t_0 \text{ in } t_1 the variable \( x \) is possibly not needed or its value is already known. In other words, evaluation of a strict expression \( \text{let } x := t_0 \text{ in } t_1 \) starts with evaluation of \( t_0 \) while evaluation of a non-strict expression \( \text{let } x = t_0 \text{ in } t_1 \) starts with evaluation of \( t_1 \).

In Danvy and Zerny’s setting only closed terms are considered, values are arbitrary \( \lambda \)-abstractions and the answers produced by evaluation are values possibly wrapped in a number of let bindings – the answers are represented here as values plugged in answer contexts \( A \). Throughout this paper we assume the variable convention, i.e., that all the bound and free variable names are pairwise distinct. The evaluation contexts \( E \) encode the strategy; in an application we look for a redex in the operator position, and the strategy for the two let constructs described informally above is encoded in the last two context constructors.

Let us briefly discuss the contraction rules. Rule (1) implements delayed computation: we delay the evaluation of the actual parameter \( t_1 \) and instead start the computation of the body \( t \). This is because the contractum here is a non-strict let expression, where, by the third production in the grammar of evaluation contexts, a reduction context is sought in the subterm \( t \). Rule (2) states that the computation of \( x \) can no longer be delayed. Rule (3) implements memoization: when the value of the term bound to \( x \) is known, it not only replaces \( x \) in the current context, but also it is stored in the answer substitution. The let construct is no longer strict because we do not know if \( x \) will be needed again.\(^1\)

\textbf{Example 3.1.} To observe the benefits of the call-by-need strategy we should look at an evaluation of a term of the form \( fts \) where the terms \( t \) and \( s \) require some computation and \( f \) contains two formal parameters: one that occurs at least twice and the other that does not occur in the body. The simplest case is \( f = \lambda x.\lambda y.xx \) and \( t = \text{id} \text{id} \) where \( \text{id} = \lambda z. z \); a definition of \( s \) is not relevant. Figure 3 shows this evaluation. In each step the redex is underlined and the relevant contexts are marked with square brackets. In steps (i) – (ii) the computations of \( x \) and \( y \) are delayed; in step (iii) \( x \) is needed. In steps (iv) – (vii) the value of the term bound to \( x \) is computed and in step (vii) it is memoized. In step (x) \( x \) is needed again and in step (xi) the computed value is reused thus avoiding the second computation of \( t \). Step (xii) finishes the computation in the body of \( f \). The variable \( y \) is never needed, so the value of \( s \) is not computed.

\(^1\) Danvy and Zerny also include a fourth rule which is a short-cut of rules (2) and (3) in the case when \( t \) is a value. We do not consider it here since it does not constitute a different contraction step but can be seen as an optimization of the reduction sequence.
An Abstract Machine for Strong Call by Need

After contracting the redex this term is processed for the first time. The possible options are: decompose it (if not relevant), parameterize by the needed variable (we use dash symbol - in place of variable when it is not relevant), and immediately unwrap it by removing the very same context in the decomposition phase of step (iv). Consider, for example, step (iv) in Figure 3. After contracting the redex id id, the contractum let z := id in z is wrapped in the context let x := □ in let y = s in x x in the recomposition phase of step (iv) and immediately unwrapped by removing the very same context in the decomposition phase of step (v).

Refocusing optimizes this naive implementation by avoiding the reconstruction of intermediate terms in a reduction sequence. To make this optimization work as an automatic procedure, the user must provide additional input – below we describe what is required by the current implementation. Figure 4 shows the essential part of this input in our implementation of weak call by need.

First, the user must define the set of values. The implementation requires all useful normal forms to be considered values, where by “useful” we mean that such a normal form put into some reduction context can potentially lead to further reduction. Under weak call by need, values are answers and needy terms, which are used in contraction (3) in Figure 2, and can be thought of as intermediate values. Their grammar is shown explicitly in Figure 4, parameterized by the needed variable (we use dash symbol - in place of variable when it is not relevant).

Second, the user must provide two functions, effectively defining the elementary contexts of the strategy. The first of them, denoted $\triangleright$ t, takes a term t and tells what to do when this term is processed for the first time. The possible options are: decompose it (if t is...
(needy terms) \( n^x ::= x \mid n^x t \mid \text{let } y := t \text{ in } n^x \mid \text{let } y := n^x \text{ in } n^y \)

(answers) \( a ::= \lambda x. t \mid \text{let } y := t \text{ in } a \)

\[
\begin{align*}
[n] & \downarrow \mathcal{V} \\
[a] & \uparrow \mathcal{R} \\
\lambda x. t & \downarrow \mathcal{V} \\
\text{let } x := t \text{ in } [n] & \uparrow \mathcal{R} \\
t_1 t_2 & \uparrow [t_1] t_2 \\
\text{let } x := t_1 \text{ in } t_2 & \uparrow \text{let } x := t_1 \text{ in } [t_2] \\
\text{let } x := t_1 \text{ in } t_2 & \uparrow \text{let } x := [t_1] \text{ in } t_2 \\
\text{let } x := [t_1] \text{ in } t_2 & \uparrow \text{let } x := [a] \text{ in } n^x \uparrow \mathcal{R}
\end{align*}
\]

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Input to the refocusing procedure for weak call-by-need. \\
\hline
\end{tabular}
\end{figure}

The second function, denoted \( \uparrow \), tells how to process a term when we already know that it is a value. In such a case we have to take into account the context surrounding the term. Thus \( \uparrow \) takes an elementary context \( ec \) and a value \( v \) and informs how to process \( ec[v] \); the options are as in the case of \( \downarrow \).

Third, in the case when the grammar of contexts contains overlapping productions, the user must also provide an order on these productions that prescribes which of the possible decompositions of a given term should be tried first (there are overlapping production in strong call by need, but not in weak call by need).

Finally, the user must prove that the defined semantics satisfies mild syntactic conditions described in detail in [12]. An example of such a condition is that the order on productions in the grammar is well-founded, which implies that all possible decompositions of a term are checked in a finite number of steps. The complete formalization of the weak call by need strategy can be found in the repository \url{http://bitbucket.org/pl-uwr/generalized_refocusing} in the file \texttt{examples/weak_cbnd.v}.

The resulting abstract machine is presented in Figure 5. We show it in a simplified form where some parts of configurations are omitted, e.g., the information about the kind of contexts, since there is only one kind and it does not affect the strategy.

The machine uses two kinds of configurations: an \( \mathcal{E} \)-configuration represents a term in a context, and the transition for these configurations correspond to the \( \downarrow \) function. A \( \mathcal{C} \)-configuration represents a value plugged in a context, and the transitions correspond to the \( \uparrow \) function. The evaluation of a term \( t \) starts in a configuration \( \langle t, \Box \rangle_{\mathcal{E}} \). In particular, the last two \( \mathcal{E} \)-transitions show the difference in the treatment of strict and non-strict let constructors. The third \( \mathcal{C} \)-transition makes it explicit when a let constructor should be treated as strict – exactly when the let-body is a term that needs the value of \( x \). Since we admit open terms, it is possible that a let-body needs a different variable to proceed – in that case we treat such a term as an intermediate value (cf. the fourth \( \mathcal{C} \)-transition). The notation \( n^{x \mapsto v} \) in the last transition stands for the result of substitution of the value \( v \) for the needed occurrence of the variable \( x \) in the needy term \( n^x \).
An Abstract Machine for Strong Call by Need

\[ \langle x, C \rangle_E \rightarrow \langle C, x \rangle_C \]
\[ \langle \lambda x.t, C \rangle_E \rightarrow \langle C, \lambda x.t \rangle_C \]
\[ \langle t_1 t_2, C \rangle_E \rightarrow \langle t_1, \square t_2 \odot C \rangle_E \]
\[ \langle \text{let } x = t_1 \text{ in } t_2, C \rangle_E \rightarrow \langle t_2, \text{let } x = t_1 \text{ in } \square \odot C \rangle_E \]
\[ \langle \text{let } x := t_1 \text{ in } t_2, C \rangle_E \rightarrow \langle t_1, \text{let } x := \square \odot \text{in } t_2 \odot C \rangle_E \]
\[ \langle \square t \odot C, n^x \rangle_C \rightarrow \langle C, n^t \rangle_C \]
\[ \langle \square t_1 \odot C, A[\lambda x.t] \rangle_C \rightarrow \langle A[\text{let } x = t_1 \text{ in } t], C \rangle_E \]
\[ \langle \text{let } x = t \text{ in } \square \odot C, n^x \rangle_C \rightarrow \langle \text{let } x = t \text{ in } n^x, C \rangle_C \]
\[ \langle \text{let } x = t \text{ in } \square \odot C, n^x \rangle_C \rightarrow \langle C, \text{let } x = t \text{ in } n^x \rangle_C \]
\[ \langle \text{let } x := \square \text{ in } n^x \odot C, n^x \rangle_C \rightarrow \langle C, \text{let } x := n^x \text{ in } n^x \rangle_C \]
\[ \langle \text{let } x := \square \text{ in } n^x \odot C, A[v] \rangle_C \rightarrow \langle A[\text{let } x = v \text{ in } n^{x+y}], C \rangle_E \]

**Figure 5** Abstract machine for weak call by need.

This machine differs from Danvy and Zerny’s machine corresponding to the reduction semantics from Figure 2 in that it handles open terms and that it is not optimized, and not all steps can be executed in constant time. In particular, since terms of the form \( A[t] \) are coerced to terms, each time we encounter a term of this form it will be decomposed from scratch, even though we know we could directly consider \( t \) in a context where all the surrounding let bindings are on the context stack. However, the structure of the resulting machine is amenable to off-the-shelf transformations leading to more efficient variants, e.g., following Danvy and Zerny’s approach that allows to transform it into a store-based abstract machine [19].

**Generalized refocusing.** The distinction between uniform and hybrid strategies was first introduced in [22]. The refocusing procedure developed in [18] and formalized in [29] was limited to uniform strategies. Recently it has been generalized to hybrid strategies in [12] – the authors show several examples of strategies based on grammars with a few (usually two or three) nonterminal symbols, including the normal-order strategy shown in Figure 1.

Strong call by need is probably the first natural example of a reduction strategy with an unbounded number of nonterminals in the underlying grammar. In the remainder of the paper we present this strategy and the abstract machine derived by generalized refocusing.

## 4 Reduction semantics for strong call by need

We now present the reduction semantics for the strong call-by-need strategy. It is strongly inspired by the semantics from [10] in that it realizes exactly the same strategy, but is designed so that it fits in the refocusing framework, which facilitates the derivation of an abstract machine. The formal correspondence between our semantics and that of [10] is outlined in Section 6.

### Terms.
As in the case of weak call by need, the grammar of terms extends lambda terms with strict and non-strict let-constructors:

\[
\text{(terms)} \quad t ::= x \mid \lambda x. t \mid t t \mid \text{let } x = t \text{ in } t \mid \text{let } x := t \text{ in } t
\]
The reduction semantics depends crucially on the notion of frozen variables introduced in [10]. A free variable is classified as frozen if it can never be substituted, either because it is not a let-bound variable (like \( x \) in \( xt \)), or if it is bound to a term that cannot become an answer (like \( x \) in \( let \ x = zz \ in \ xt \)). If a variable is not frozen, then we call it active. The status of variables depends on the context of evaluation: for example, when we consider the term \( xt \) in the empty context, then \( x \) is frozen; but when the same term is plugged in the context \( let \ x = \lambda z. z \ in \ □ \) then \( x \) is no longer frozen. Intuitively, the latter context defrosts the variable and makes it active and substitutable.

In the process of evaluation we consider terms of particular shapes: structures \( S \), normal forms \( N \), and needy terms \( N^- \). Structures and normal forms are defined by mutual recursion. Needy terms are parameterized by the variable they need and additionally by a context kind, which will be defined later. Whether a term falls into one of these categories depends on the status of its free variables, therefore each category is further parameterized by a set of variables containing exactly the frozen variables of a term.

The grammar of structures is shown in Figure 6. Informally, a structure is a normal form formed around a frozen variable, e.g., \( let \ x = y \ in \ z (\lambda y. y) \) belongs to \( S_z \) and is a structure formed around the frozen variable \( z \). Structures here are almost equivalent to structures in [10], with the difference that here we parameterize them precisely with sets of frozen variables, while in [10] any superset of frozen variables is a good parameter.

Normal forms in \( N \) (with respect to strong call-by-need reduction) are either structures or irreducible terms built around a normal \( \lambda \)-abstraction in \( N_\lambda \) (see Figure 7). They can be seen as natural generalizations of weak call-by-need normal forms that arise when structures are introduced. The last two rules impose restrictions on the sets \( \phi \) and \( \psi \): first, \( x \) cannot be a member of \( \phi \) because otherwise contexts \( let \ x = t \ in \ □ \) and \( let \ x := s \ in \ □ \) defrost it; second, \( n \) in the last rule occurs in a position of a needy term with needed variable \( x \), so \( x \) must be a member of \( \psi \).

Finally, needy terms are terms uniquely decomposable into a reduction context and an active variable. Such terms can be seen as intermediate normal forms that arise in the normalization process when we try to establish if a given let-bound variable is needed. The grammar of needy terms is shown in Figure 8. It is important to note that the same term can be either a structure, or a needy term depending on the status of its free variables, e.g.,
An Abstract Machine for Strong Call by Need

8:10

x ∈ S_{\{x\}} (here x is frozen) and x ∈ N_{k,\emptyset} (here x is active). The latter term denotes the variable x, which is needed in the empty context (k denotes the kind of context) and active in terms of the form let x := t in x.

Reduction contexts. The strong call-by-need reduction strategy generalizes the weak call-by-need strategy – informally, it first tries to evaluate terms with the weak strategy and after reaching a weak value it attempts to normalize it further (i.e., inside a lambda abstraction or a neutral term). This pattern is analogous to the normal-order strategy, which can be adequately described using hybrid reduction contexts whose grammar defines the interconnection between the weak and the strong strategy (see Figure 1). To define strong contexts for normal order we need to use two kinds: E - for weak contexts (realizing call by name), and C - for strong contexts. When we consider call by need, we need to further instrument these kinds – they are parameterized by a set of frozen variables. Given a raw kind \( k \in \{E, C\} \) and a set of frozen variables \( \phi \) we write \( k \cdot \phi \) for the kind \( k \) parameterized by \( \phi \). The union \( \{x\} \cup \phi \) is abbreviated \( x, \phi \).

Reduction contexts are built from elementary contexts parameterized by two kinds; \( EC_{k_1}^{k_2} \) denotes an elementary context of kind \( k_1 \) whose hole is of kind \( k_2 \). Figure 9 describes all the elementary contexts. In particular, \( \lambda x.\square \) is an elementary context of raw kind \( C \) (it is only available under the strong strategy), and when reducing under the lambda the same strong strategy is used with the \( \lambda \)-bound variable treated as frozen inside the body. In both the weak and the strong variant the context \( \square t \) can be used, but its hole always forces the weak strategy to be used inside (i.e., when we decompose the left-hand-side of an application we always use the weak strategy). The context \( s \cdot \square \) enforces strong reduction of the argument of an application – in case when the operand is a structure (therefore, irreducible and not creating a \( \beta \)-redex). The condition \( \psi \subseteq \phi \) ensures that \( s \) is indeed a structure (and not a needy term) in the given context, see Example 4.1 at the end of this section.

The context let \( x := t \) in \( \square \) is used when first processing a let term, it considers the let-bound variable active inside the let-body – next we need to establish if it is needed there. The context let \( x := \square \) in \( n^x \) is available when we already know that \( x \) is needed in the let-body and it is necessary to compute the value of the term bound to the variable (using the weak strategy). Finally, the context let \( x := s \) in \( \square \) handles situations when the term bound to \( x \) does not evaluate to a weak \( \lambda \)-value but normalizes to a structure. In this case we go back to evaluating the let-body, this time with the let-bound variable treated as frozen because it will never be substituted (note that this is the second time the let-body will be decomposed but the decomposition will be different this time because the set of frozen variables has changed). Example 4.1 illustrates the last three rules.
Whereas the fourth one is added to handle open terms (structures). The first two rules are generalizations of weak call-by-need contractions from Section 2. Just like values, certain terms become redices (i.e., atomic reducible terms) given kind. Strong values $V_{\psi}$ are all strong normal forms that match the kind. Weak values $V_{\phi}$ are answers, defined as in weak call by need (i.e., as lambda values in answer contexts, $a ::= A[v]$), and structures. The set of frozen variables dictated by the kind must be a superset of the set of frozen variables of the value (here again it ensures that $s$ is indeed a structure in a given context). The strategy that we are describing works both for closed and for open terms – in the latter case we treat free variables as frozen. During normalization it happens that we produce intermediate values containing active variables – these are exactly the needy terms.

The condition $x \notin \psi$ ensures that $n^x$ is indeed a needy term, and not a structure under the given kind.

**Contraction.** Just like values, certain terms become redices (i.e., atomic reducible terms) only in a specific context. Therefore, the type of redices is parameterized by the kind.

$$
A[\lambda x.t] \xrightarrow{\phi} \text{let } x \equiv t \text{ in } t
$$

$$
\begin{align*}
n^x & \in N^k_{\psi} \\
\text{let } x := t \text{ in } n^x & \xrightarrow{\psi} \text{let } x := s \text{ in } A[v] \xrightarrow{\phi} n^x
\end{align*}
$$

The first three rules are generalizations of weak call-by-need contractions from Section 2, whereas the fourth one is added to handle open terms (structures). The first two rules

**General reduction contexts are composed from elementary ones with matching kinds:**

$$
\begin{align*}
\text{let } x & := t \text{ in } \square \in EC^k_{x,\phi} \\
\text{let } x & := \square \text{ in } n^x \in EC^k_{x,\phi} \\
\text{let } x & := s \text{ in } \square \in EC^k_{x,\phi}
\end{align*}
$$

This is a representation of contexts inside-out (the hole of the context $c$ matches the kind of the elementary context $cc$ placed inside it).

**Grammar.** The grammar of reduction contexts contains nonterminals of the form $k \cdot \phi$, where $k \in \{C, E\}$ and $\phi$ is an arbitrary (finite) set of variables, with starting nonterminal $C \cdot \emptyset$. The productions in this grammar have either the form $k \cdot \phi \rightarrow C[k' \cdot \psi]$ where $C$ is an elementary context in $EC^{k,\phi}$, or the form $k \cdot \phi \rightarrow \square$. Note that since there are no restrictions on the number of variables in terms, the grammar contains an infinite number of productions.

**Values.** Because the strong call-by-need strategy mixes weak and strong normalization, the notion of value depends on the kind of the context.

$$
\begin{align*}
n & \in N_{\phi} \quad \phi \subseteq \psi \\
n & \in V_{C \psi} \\
a & \in V_{E \psi} \\
s & \in S_{\phi} \\
s & \in V_{E \psi} \\
n^x & \in N^k_{\lambda x.\phi} \\
x & \notin \psi \\
x & \notin \psi \\
x & \notin \psi
\end{align*}
$$

Strong values $V_{C \psi}$ are all strong normal forms that match the kind. Weak values $V_{E \psi}$ are answers, defined as in weak call by need (i.e., as lambda values in answer contexts, $a ::= A[v]$), and structures. The set of frozen variables dictated by the kind must be a superset of the set of frozen variables of the value (here again it ensures that $s$ is indeed a structure in a given context). The strategy that we are describing works both for closed and for open terms – in the latter case we treat free variables as frozen. During normalization it happens that we produce intermediate values containing active variables – these are exactly the needy terms.

The first three rules are generalizations of weak call-by-need contractions from Section 2, whereas the fourth one is added to handle open terms (structures). The first two rules
encode the usual call-by-need computation steps and they coincide with the contractions in [10], and the last two can be seen as “administrative” reductions that are responsible for marking/unmarking strict \textit{lets}. The first rule creates a new let-binding whenever a lambda abstraction is applied to an argument (this binding will be processed lazily). The second rule is triggered in a situation when a needed let-bound variable has a value ready to be used in the let-body; the value is then substituted for the variable and the variable ceases to be needed. The third rule consists in marking the let-bound variable as needed when we know that the let-body is needy of this variable. The last rule is used to convert a strict let to an answer – when the variable is not really needed (it cannot happen if we start reducing with a term not containing strict lets).

\textbf{Reduction.} The reduction relation is defined as usual, with the restriction that the kind of contraction must match the kind of the hole in the reduction context: a term \( t \) reduces in one step to \( t' \) if it can be decomposed as \( t = \mathcal{E}[r] \) for some redex \( r \) and an evaluation context \( \mathcal{E} \in C_{k,\phi} \), the redex \( r \) can be rewritten in one step to \( t'' \) by contraction \( \mathcal{C} \cdot \phi \), and \( t' \) is obtained by the recomposition of \( \mathcal{E} \) and \( t'' \), that is \( t' = \mathcal{E}[t''] \).

\textbf{Example 4.1.} Consider an evaluation of a term of the form \texttt{let x = y y in x t}, where \( y \) is the only free (and frozen) variable, with the strong strategy \( \mathcal{C} \cdot \{ y \} \). We first decompose the term to the context \texttt{let x = y y in □ ∈ EC_{C}(y)} and the let-body \texttt{x t}. Here \( x \in N_X \) is a needy term, so the input term is rewritten to \texttt{let x := y y in x t}, using the (\texttt{ls}) rule and contraction \( \mathcal{C} \cdot \{ y \} \). Since \( y y \) is a structure in \( S_{\{y\}} \) and \( \{ y \} \subseteq \{ x, y \} \), the obtained term can be decomposed using the last rule in Fig. 9 to the context \texttt{let x := y y in □ ∈ EC_{C}(x,y)} and the term \texttt{x t}, which we now visit for the second time. Note that the inclusion \( \psi \subseteq \phi \) from Fig. 9 forces us to change the evaluation strategy here by adding \( x \) to the parameter set, so \( x \) becomes frozen. Since \( x \in S_{\{x\}} \) and \( \{ x \} \subseteq \{ x, y \} \), the third rule in Fig. 9 prescribes now to evaluate \( t \) using the strong strategy \( \mathcal{C} \cdot \{ x, y \} \).

\section{An abstract machine for strong call by need}

In this section we present an abstract machine for strong call by need derived in the framework of generalized refocusing. The Coq development can be found in the repository \url{http://bitbucket.org/pl-uwr/generalized_refocusing} in the file \url{examples/strong_cbnd.v}. The machine (with some technical clutter removed) is presented in Figure 10. We assume implicit α-renaming of bound variables in order to avoid name clashes (cf. side conditions in transitions (3),(8),(21)).

The machine has two kinds of configurations. An \( \mathcal{E} \)-configuration of the form \( \langle t, C, k, \phi \rangle_\mathcal{E} \) consists of a term, a surrounding context, and the kind of the context hole represented by the last two components of the configuration. In turn, a \( \mathcal{C} \)-configuration of the form \( \langle C, v, k, \phi \rangle_\mathcal{C} \) consists of a context, a value plugged in this context, and the kind of the context hole. Because both contexts and values are parameterized by kinds, a configuration is considered correct if the last two components match the kind of the context component (in both \( \mathcal{E} \) and \( \mathcal{C} \)-configurations) and of the value component (in \( \mathcal{C} \)-configurations). If we start the machine with a correct configuration, then all the configurations in the machine run are correct by construction.

The values computed by the machine (and used in \( \mathcal{C} \)-configurations) coincide with those in the reduction semantics. In Figure 10 we use notation \( n \) for arbitrary normal forms, \( s \) for structures, \( n^\lambda \) for lambda values, \( n^\pi \) and \( n^y \) for needy terms. Sometimes it is not obvious to
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$(t_1 t_2, C, k, \phi)_E$</td>
<td>$(t_1, \Box t_2 \circ C, k, \phi)_E$</td>
</tr>
<tr>
<td>3</td>
<td>$(\text{let } x = t_1 \text{ in } t_2, C, k, \phi)_E$</td>
<td>$(t_2, \text{let } x = t \text{ in } t \circ C, k, \phi)_E$ if $x \notin \phi$</td>
</tr>
<tr>
<td>4</td>
<td>$(\text{let } x := t \text{ in } n^x, C, k, \phi)_E$</td>
<td>$(t, \text{let } x := \Box \text{ in } n^x \circ C, k, \phi)_E$</td>
</tr>
<tr>
<td>5</td>
<td>$(x, C, k, \phi)_E$</td>
<td>$(C, \text{str}(x), k, \phi)_C$ if $x \in \phi$</td>
</tr>
<tr>
<td>6</td>
<td>$(x, C, k, \phi)_E$</td>
<td>$(C, \text{nd}(x), k, \phi)_C$ if $x \notin \phi$</td>
</tr>
<tr>
<td>7</td>
<td>$(\lambda x. t, C, E, \phi)_E$</td>
<td>$(C, \text{ans}(\lambda x. t), E, \phi)_C$</td>
</tr>
<tr>
<td>8</td>
<td>$(\lambda x. t, C, \phi)_E$</td>
<td>$(t, \lambda x. \Box \circ C, C, x \cdot \phi)_E$ if $x \notin \phi$</td>
</tr>
<tr>
<td>9</td>
<td>$(\lambda x. \Box \circ C, n, C, \phi)_C$</td>
<td>$(C, \lambda x. n, C, \phi \setminus {x})_C$</td>
</tr>
<tr>
<td>10</td>
<td>$(\Box t \circ C, \phi)_E$</td>
<td>$(A[\lambda x. r], C, k, \phi)_E$</td>
</tr>
<tr>
<td>11</td>
<td>$(\Box t \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(n^y), E, \phi)_C$ if $y \notin \phi$</td>
</tr>
<tr>
<td>12</td>
<td>$(\Box t \circ C, \phi)_E$</td>
<td>$(t, s \circ C, C, \phi)_E$</td>
</tr>
<tr>
<td>13</td>
<td>$(s \circ C, n, C, \phi)_C$</td>
<td>$(C, \text{str}(s n), k, \phi)_C$</td>
</tr>
<tr>
<td>14</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{ans}(\text{let } x = t \text{ in } A[v]), k, \phi)_C$</td>
</tr>
<tr>
<td>15</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(n^y), k', \phi)_C$ if $x \notin \phi$</td>
</tr>
<tr>
<td>16</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{nd}((\text{let } x = t \text{ in } n^y), k, \phi)_C$ if $y \neq x, y \notin \phi$</td>
</tr>
<tr>
<td>17</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{str}(\text{let } x = t \text{ in } s), k, \phi)_C$</td>
</tr>
<tr>
<td>18</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{str}(\text{let } x = t \text{ in } n^y), k, \phi)_C$</td>
</tr>
<tr>
<td>19</td>
<td>$(\text{let } x = t \text{ in } \Box \circ C, \phi)_E$</td>
<td>$(C, \text{ans}(A[v]), E, \phi)_C$</td>
</tr>
<tr>
<td>20</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(\text{let } x := n^y \text{ in } n^x), k, \phi)_C$ if $y \notin \phi$</td>
</tr>
<tr>
<td>21</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(\text{let } x := t \text{ in } n^y), k, \phi)_C$ if $y \notin \phi$</td>
</tr>
<tr>
<td>22</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(\text{let } x := t \text{ in } n^y), k, \phi)_C$ if $y \notin \phi$</td>
</tr>
<tr>
<td>23</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{str}(\text{let } x := t \text{ in } s'), k, \phi)_C$</td>
</tr>
<tr>
<td>24</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{str}(\text{let } x := t \text{ in } s'), k, \phi)_C$</td>
</tr>
<tr>
<td>25</td>
<td>$(\text{let } x := t \text{ in } n^x \circ C, \phi)_E$</td>
<td>$(C, \text{nd}(\text{let } x := t \text{ in } n^y), k, \phi)_C$</td>
</tr>
<tr>
<td>26</td>
<td>$(\Box, n, C, \phi)_E$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

**Figure 10** An abstract machine for strong call by need.
which category a given value falls; for example \( x \) in the configuration \( \langle C, x, k, \phi \rangle_C \) is either a structure or a needy term, depending on whether \( x \in \phi \). To ease reading the transition rules, we attach tags to the different value constructors in the machine:

\[
n ::= \text{ans}(A[\lambda x.t]) \mid \text{str}(s) \mid \text{nd}(n^\lambda) \mid \text{lnf}(n^\lambda)
\]

writing respectively \( \langle C, \text{str}(x), k, \phi \rangle_C \) and \( \langle C, \text{nd}(x), k, \phi \rangle_C \) instead of \( \langle C, x, k, \phi \rangle_C \).

An \( E \)-configuration either decomposes a term by pushing a new elementary context on the existing context (in such a way that their kinds match) and proceeds with evaluation of a subterm, or it calls a \( C \)-configuration if the term is a value. In particular, transition (4) prescribes that if a term is a strict let, then we need to evaluate (to weak value) the term \( t_1 \) bound to the variable. If the term happens to be a variable, it is a value but its status depends on the kind, more specifically on the set of frozen variables \( \phi \) in the configuration: if the variable is not in this set, then it will be treated as active (cf. transitions (5) and (6)). A lambda abstraction is a value in a \( E \)-hole but it is further decomposed in a \( C \)-hole.

A \( C \)-configuration dispatches on the context when a (matching) value is plugged in its hole. For example, transition (10) encodes \( \beta \)-contraction, transition (15) encodes the \( \lambda s \)-contraction, transition (19) encodes the \( \lambda sv \)-contraction, and transition (23) encodes the \( \lambda ns \)-contraction.

The machine correctly realizes the strong call-by-need strategy, it decomposes terms based on local information given in a configuration, but it still could be optimized in various directions, in particular, using insights from existing work on abstract machines for weak lazy evaluation [3, 19]. It is future work to check which of the transformations discussed there generalize to the strong case, in particular how to systematically obtain an efficient store-based machine.

## 6 Correctness

In this section we show how our reduction semantics relates to the Balabonski et al.’s. To this end, we need to introduce some of the notions they use in their work. In the following, we refer to their language as \( \Lambda \), to our language as \( \Lambda_s \), and to standard lambda calculus as \( \Lambda_\beta \).

Balabonski et al. do not distinguish between strict and non-strict let syntactically, they only have one form of let-construct. In order to discover the difference, they have to traverse the term to check if the let-bound variable is really needed, i.e., if it is in the set of non-garbage variables of the body. Non-garbage variables are defined as follows:

\[
\begin{align*}
\text{ngv}(x) &= \{ x \} \\
\text{ngv}(\lambda x.t) &= \text{ngv}(t) \setminus \{ x \} \\
\text{ngv}(t_1 t_2) &= \text{ngv}(t_1) \cup \text{ngv}(t_2) \\
\text{ngv}(t_2[x \leftarrow t_1]) &= \text{ngv}(t_2) \setminus \{ x \} \cup \begin{cases} 
\text{ngv}(t_1), & \text{if } x \in \text{ngv}(t_2) \\
\emptyset, & \text{otherwise}
\end{cases}
\end{align*}
\]

where the notation \( t_2[x \leftarrow t_1] \) denotes an explicit substitution of \( t_1 \) for \( x \) in \( t_2 \).

In contrast, our intermediate language makes this distinction effective just as soon as decomposition of the term reveals it.
We define an erasure operation $|-| : \Lambda_s \to \Lambda$:

\[
\begin{align*}
|x| &= x \\
|\lambda x. t| &= \lambda x.|t| \\
|t_1 \cdot t_2| &= |t_1| \cdot |t_2| \\
|\mathtt{let} \ x = t_1 \ \mathtt{in} \ t_2| &= |t_2|[x\backslash t_1] \\
|\mathtt{let} \ x := t_1 \ \mathtt{in} \ t_2| &= |t_2|[x\backslash t_1]
\end{align*}
\]

This operation gives us a translation from $\Lambda_s$ to $\Lambda$. A translation from $\Lambda$ to $\Lambda_s$ is trivial: every term in $\Lambda$ is a term in $\Lambda_s$ (modulo the notation; $t_2[x\backslash t_1]$ is represented as $\mathtt{let} \ x = t_1 \ \mathtt{in} \ t_2$ in $\Lambda$).

We can show that the set of frozen variables of $\Lambda_s$-normal forms coincides with the non-garbage variables. All lemmas below have routine inductive proofs, which we omit here.

\begin{lemma}
For all $\Lambda_s$-normal forms $n \in N_\phi$ and for all variables $x \in \phi$, $x \notin \ngv(|n|)$.
\end{lemma}

The following two lemmas give a correspondence between normal forms in $\Lambda$ and $\Lambda_s$.

\begin{lemma}
For all $\Lambda_s$-normal forms $n \in N_\phi$, $|n| \notin \ngv$ (it is a $\Lambda$-normal term).
\end{lemma}

\begin{lemma}
For all $\Lambda$-normal forms $n \in \ngv$, there exist $n_0$ and $\phi \subseteq \psi$ such that $|n_0| = n$ and $n_0 \in N_\phi$.
\end{lemma}

The next lemma states that needy terms correspond to $\Lambda$-contexts with holes filled with a designated variable. The notation $C[x]$ comes from [10] and denotes the variable $x$ plugged in the context $C$ that does not capture the variable (there are no abstractions or explicit substitutions that bind $x$ in $C$). Similarly, notation $E_\phi$ (respectively, $E_\phi$) is introduced in [10] and denotes evaluation contexts (respectively, inert evaluation contexts) under the set of frozen variables $\phi$, both defined in [10].

\begin{lemma}
For every needy term $n^x \in N_{E_\phi}^x$ ($n^x \in N_{E_\phi}$, resp.) there exists a $\Lambda$-context $C \in E_\phi$ ($C \in E_\phi$, resp.) such that $|n^x| = C[x]$.
\end{lemma}

In order to relate contexts, we need to introduce the conversion operator that transforms $\Lambda_s$-elementary contexts into $\Lambda$-contexts.

\[
\begin{align*}
|\lambda x. \Box|_c &= \lambda x. \Box \\
|\Box t|_c &= \Box |t| \\
|s \Box|_c &= |s| \Box \\
|\mathtt{let} \ x = t \ \mathtt{in} \ \Box|_c &= \Box[x\backslash t] \\
|\mathtt{let} \ x := \Box \ \mathtt{in} \ n^x|_c &= C[x][x\backslash \Box] \quad \text{where } C[x] = |n^x| \\
|\mathtt{let} \ x := s \ \mathtt{in} \ \Box|_c &= \Box[x\backslash s]
\end{align*}
\]

Based on this definition and the composition of contexts in $\Lambda$, we can translate between $\Lambda_s$-contexts and $\Lambda$-contexts. Here $EC_{k,\phi}$ is the union $\bigcup_{k,\phi} EC_{k,\phi}$.

\begin{lemma}
For every elementary context $cc \in EC_{k,\phi}$ there is a $\Lambda$-context $|cc|_c \in E_\phi$, and for every elementary context $cc \in EC_{\infty,\phi}$ there is a $\Lambda$-context $|cc|_c \in E_\phi$ and such that

\[
\forall t, |cc[t]| = |cc|_c[t].
\]

\begin{lemma}
\end{lemma}
Lemma 6.6. For every $\Lambda$-context $C \in E_\psi$ (or $C \in E_\psi^c$, resp.) there is a $\Lambda_s$-context $c \in C^\phi_\psi$ (or $c \in C^\phi_\psi^c$, resp.) such that $\psi \subseteq \phi$ and $|c|_c = C$.

Having the ability to translate between both terms and contexts in the two languages, we can prove that $\Lambda_s$ correctly simulates reductions in $\Lambda$. The reduction in $\Lambda_s$ possibly uses more steps than the reduction in $\Lambda$ because of switching between strict and non-strict versions of let constructs.

Definition 6.7. We say that a term $t \in \Lambda_s$ is proper if all its subterms of the form $\text{let } x := t_1 \text{ in } t_2$ are such that $t_2 = n^x$ for some needy term $n^x$, i.e., that strict let’s are correctly marked.

Proposition 6.8. For all $\Lambda$-terms $t_1$ and $t_2$, if $t_1 \rightarrow^N_\Lambda t_2$ then there exists a proper $t_s \in \Lambda_s$ such that $t_1 \rightarrow^*_\Lambda t_s$ and $|t_s| = t_2$. Moreover, for all proper $t_s \in \Lambda_s, t \in \Lambda$, if $|t_s| = t$ then $t \rightarrow^*_\Lambda t_s$.

Proposition 6.9. For all $t_1, t_2 \in \Lambda_s$, if $t_1 \rightarrow^N_\Lambda t_2$ then $|t_1| = |t_2|$ or $|t_1| \rightarrow^N_\Lambda |t_2|$. There is no infinite sequence $t_1 \rightarrow^N_\Lambda t_2 \rightarrow^N_\Lambda \ldots \rightarrow^N_\Lambda t_n \rightarrow^N_\Lambda \ldots$ such that $|t_1| = |t_n|$ for all $n$.

As a consequence of the correct simulation result we obtain that $\Lambda_s$ achieves the same normal forms as $\Lambda$.

Lemma 6.10. For all $\phi$ and for all $t, r \in \Lambda$ such that $t \rightarrow^*_\Lambda r$ and $r \in N_\psi$,

$$t \rightarrow^*_\Lambda r_s$$

for some $r_s \in N_\psi$ with $\psi \subseteq \phi$, and such that $|r_s| = r$.

Lemma 6.11. For all $\phi$, and for all $t \in \Lambda, r_s \in \Lambda_s$ such that $t \rightarrow^*_\Lambda r_s$ and $r_s \in N_\phi$,

$$t \rightarrow^*_\Lambda |r_s|$$ holds and $|r_s| \in N_\phi$.

Now the completeness and conservativity results propagate from [10] to our setting. Here $(\cdot)^\circ$ denotes the unfolding function defined in [10], which is a translation from $\Lambda$ to $\Lambda_\beta$. The completeness result expresses that whenever a pure lambda term reduces to a normal form, the same normal form can be reached by the strong call-by-need strategy followed by unfolding the substitutions wrapping the obtained value.

Corollary 6.12 (Completeness). For all lambda terms $t \in \Lambda_\beta$, if $t$ reduces to a normal form $r$ in $\Lambda_\beta$ (in symbols, $t \rightarrow^*_\Lambda r$), then there exists a normal form $r_s$ in $\Lambda_s$ such that

$$t \rightarrow^*_\Lambda r_s$$

and $|r_s|^\circ = r$.

The conservativity result expresses that any strong call-by-need reduction has a weak call-by-need reduction as a prefix.

Corollary 6.13 (Conservativity). For all lambda terms $t \in \Lambda_s$, if

$$t = t_0 \rightarrow^N_\Lambda t_1 \rightarrow^N_\Lambda t_2 \rightarrow^N_\Lambda \ldots \rightarrow^N_\Lambda t_n$$

is a sequence of reductions ending with a normal form $t_n$, then there exists an $i \leq n$ such that $t_0 \rightarrow \ldots \rightarrow t_i$ is a reduction with weak call-by-need strategy.
Related work

Operational accounts for lazy evaluation are numerous, coming both from the practical and the theoretical considerations. The common implementation models include a canonical store-based abstract machine, where the store component is used to memoize the computed values and facilitate their reuse [26], as well as graph reduction machines devised on a principle of sharing of subgraphs representing argument terms [31]. On the other hand, theoretical investigations of call by need focus on establishing equational reasoning principles for lazy evaluation and various calculi have been developed for this purpose, with the canonical store-based natural semantics [24], and a storeless calculus based on let-constructs [8, 25]. These two worlds cross-fertilize, and there exist machines derived in an ad hoc manner from calculi, e.g., Sestoft’s machine obtained from Launchbury’s semantics [28].

A more principled approach to interderiving semantic artefacts has been advocated by Danvy and his collaborators; it consists in mechanizing derivations by identifying common transformation patterns that can be applied generically and provide guidance in the derivation process. In particular, they have used the functional correspondence to connect evaluators with abstract machines – including weak call by need [6, 7, 27], and the syntactic correspondence (based on refocusing) to connect reduction semantics and abstract machines [13].

Specific to call by need is an operational account of Danvy and Zerny who unify the various operational artefacts for lazy evaluation and provide a systematic approach to go from the canonical let-calculus of Ariola et al. through a series of refined reduction semantics and the corresponding abstract machines to a canonical store-based abstract machine (lazy Krivine machine) [19].

Strong reduction has been less studied operationally. The prominent examples of abstract machines for strong normalization include Crégut’s abstract machine extending the Krivine machine [15, 16], which has later been derived more systematically by Nogueira and Garcia-Perez from the normal-order reduction strategy [22]; another variant has been devised using Linear Substitution Calculus [2] and useful sharing [1]. A different approach has been taken by Gregoire and Leroy whose normalization strategy uses call by value rather than call by name as a substrategy and was motivated by Coq implementation [23], and has later been refined as an instance of normalization by evaluation [14]. Efficient abstract machines for strong call by value have been recently devised by Accattoli et al. using Fireball Calculus [4, 5].

The only work on strong call by need seems to be Balabonski et al.’s strategy which we build on [10], recently extended to the Calculus of Inductive Constructions [11], focused on ensuring completeness of the strong call-by-need strategy.

The framework we work with is based on generalized refocusing that generalizes the syntactic correspondence used for weak strategies in that it allows for more complex strategies to be expressed (ones that can be seen as compositions of several substrategies), and the corresponding abstract machines to be derived [12].

Conclusion

We presented a systematic approach to defining reduction semantics for the call-by-need strategies, both weak and strong, in the framework of generalized refocusing. Within this framework we derived the corresponding abstract machines that are correct by construction. We observed that this approach is very effective – the requirements posed by the framework provide just enough structure and constraints to guide us in the process; in contrast, devising a machine from scratch would be much more a trial-and-error process, and quite tedious.
8:18 An Abstract Machine for Strong Call by Need

The derived machine is not optimized and thus not as effective as it might be. Our approach opens the possibility to systematically transform and optimize it. We leave it to the future work to further connect it to other semantic formats and models of implementation (in particular, a store-based abstract machine), in the spirit of the derivational approach of Danvy and Zerny [19], and to analyse its complexity as in the line of work of Accattoli [3].

References


An Abstract Machine for Strong Call by Need

