Generalized Refocusing: From Hybrid Strategies to Abstract Machines*

Małgorzata Biernacka[1], Witold Charatonik[2], and Klara Zielińska[3]

1 **Institute of Computer Science, University of Wrocław, Wrocław, Poland**
mabi@cs.uni.wroc.pl
2 **Institute of Computer Science, University of Wrocław, Wrocław, Poland**
wch@cs.uni.wroc.pl
3 **Institute of Computer Science, University of Wrocław, Wrocław, Poland**
kzi@cs.uni.wroc.pl

Abstract

We present a generalization of the refocusing procedure that provides a generic method for deriving an abstract machine from a specification of a reduction semantics satisfying simple initial conditions. The proposed generalization is applicable to a class of reduction semantics encoding hybrid strategies as well as uniform strategies handled by the original refocusing method. The resulting machine is proved to correctly trace (i.e., bisimulate in smaller steps) the input reduction semantics. The procedure and the correctness proofs have been formalized in the Coq proof assistant.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases reduction semantics, abstract machines, formal verification, Coq

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.10

1 Introduction

Refocusing has been introduced by Danvy and Nielsen as a generic procedure to derive an efficient abstract machine from a given reduction semantics [7]. The method has been applied (by hand) to a number of reduction semantics both to derive new machines, and to establish the connection between existing machines and their underlying reduction semantics [3, 10]. Sieczkowski et al. later proposed an axiomatization of reduction semantics sufficient to apply the method and formalized the entire procedure in Coq [17].

However, the refocusing procedure as described previously does not account for a wide class of hybrid rewriting strategies that can be thought of as composed from several different substrategies. Notably, this class includes the normal-order strategy for full normalization in the lambda calculus which is of particular interest due to its use in type checking algorithms for dependently typed programming languages and logic systems, such as Coq [12] and Agda [14].

The problem with applying refocusing to normal order and other similar strategies has been observed by Danvy and Johannsen [6], and by García-Pérez and Nogueira [11], who offer different, partial solutions. The former attribute the problem to backward-overlapping reduction rules in an outermost strategy and propose to apply a correction in the form of "backtracking" in the decomposition procedure. The latter notice that refocusing in this

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

case becomes context-dependent, and they identify a shape invariant of the context stack they then exploit to transform the machine to an efficient form. Both of these, however, are ad-hoc handmade solutions and do not shed light on how to proceed in a general case.

In this work we propose a generic refocusing procedure that handles both uniform and hybrid rewriting strategies satisfying simple initial conditions. It subsumes Sieczkowski et al.'s formalization [17] that is applicable only to uniform strategies. The procedure is powerful enough to generate realistic abstract machines, particularly machines with environments. It can also be used to verify correctness of existing abstract machines.

Contribution. The specific contributions of our work are as follows:

- We propose a new format for specifying reduction semantics. This format generalizes the standard approach in that it is based on kinded reduction contexts and the ability to restrict composition of reduction contexts based on matching kinds.
- We propose a generalization of refocusing, i.e., an automatic procedure that leads from a specification of a reduction semantics to the corresponding abstract machine. This generalization is enabled by the format of generalized reduction semantics.
- We formalize the generalized refocusing procedure in Coq.
- We prove in Coq the correctness of the abstract machine obtained by refocusing. The correctness of the machine is defined in terms of *tracing* – a variant of bisimulation expressing the property that one system is simulated by another in smaller steps. Specifically, the Coq development contains a proof that the machine obtained by refocusing traces the input reduction semantics.

Related work. On the theoretical side, our work aims at a principled approach to specifying operational semantics, such that facilitates reasoning, analysis, and comparison of various reduction strategies and their composition. The connection made by the formal account of refocusing further enables reasoning about the corresponding abstract machines and possibly their optimizations. This toolbox comes in especially handy when we approach various reduction strategies, reduction semantics, and abstract machines in the existing literature, enabling us to disentangle the different semantic artifacts, expose their underlying structure, and relate one to another. In our work, the issue of hybrid strategies arose directly from the study of existing artifacts, such as abstract machines for full normalization in the lambda calculus that can be proved to implement the normal-order strategy (such as Crégut's abstract machine [4, 11]), or Grégoire & Leroy's strong reduction used to devise an efficient implementation model for Coq [12]; other hybrid strategies are considered in [11, 16].

On the practical side, a natural application of this work is a usable framework for automatic generation of provably correct semantic artifacts, from higher-level semantics to abstract machines. From this point of view our work is related to the existing, much more advanced tools, such as PLT Redex or the K framework, that also facilitate specification of various formats of operational semantics, and experimentation with them [9, 15]. The important difference is that we make explicit the process of producing implementation models for the given high-level specification, while making sure they are provably correct. In the process all the intermediate artifacts can be made available to the user who can further manipulate them. Thus our work can be seen to complement the work on practical tools in the quest to increase reliability of these tools.

Plan of the paper. In Section 2 we recall the basic semantic ingredients needed to describe the original refocusing procedure in its basic form applicable to uniform strategies. In

Sections 3 and 4 we present our main contributions: Section 3 contains our format for specifying reduction semantics, Section 4 – a generalization of the refocusing method to handle hybrid strategies. In Section 5 we discuss the correctness of the generated machine. In Section 6 we briefly describe the implementation, and in Section 7 we conclude.

2 Preliminaries

In this section we recall the fundamental concepts used both in the original setting and in our generalization of the refocusing procedure. We use standard lambda-calculus notions without defining them here (e.g., capture-avoiding substitution, β-reduction).

2.1 Reduction semantics

A *reduction semantics* is a kind of small-step operational semantics, where the positions in a term that can be rewritten are explicitly defined by reduction contexts, rather than implicit in inference rules [8]. More formally, a reduction semantics is a quadruple $\langle \mathcal{T}, \mathcal{C}, \rightharpoonup, V \rangle$, where $\mathcal{T}$ is a set of terms, $\mathcal{C}$ is a set of *reduction contexts*, $\rightharpoonup \subseteq \mathcal{T} \times \mathcal{T}$ is a local rewriting relation (also called *contraction*), and $V \subseteq \mathcal{T}$ is a set of *values*, i.e., terms that represent valid results of computation.

Commonly $\mathcal{C}$ is given by a grammar of contexts. By a *context* we mean a term with exactly one occurrence of 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 then refer to C as a *prefix* of $C[t]$. The $\rightharpoonup$ relation is typically given by a set of rewriting rules. Terms that can be rewritten by $\rightharpoonup$ are called *redices* and those produced by it – *contracta*. 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 *reduction relation* $\to \subseteq \mathcal{T} \times \mathcal{T}$ in the reduction semantics is defined as the compatible closure of contraction: $t_1 \to t_2$ if and only if there exist $C \in \mathcal{C}$, $t_1', t_2' \in \mathcal{T}$ such that $t_1 = C[t_1']$, $t_2 = C[t_2']$, and $t_1' \rightharpoonup t_2'$. *Evaluation* is then defined as the reflexive-transitive closure of the reduction relation (written $\to^*$), and *normal forms* as terms that cannot be reduced. We require the set of values to be a subset of normal forms. Normal forms that are not values are called *stuck terms*.

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 P of productions. All these sets must be finite. In this paper each production has the form $C \to \tau$ where $C \in N$ and τ is a term with free variables in $N \cup \{[\,]\} \cup S$, with exactly one occurrence of a variable from $N \cup \{[\,]\}$. Figure 1 contains an example of a grammar of contexts. We will use a convention that starting nonterminal symbols in grammars are underlined. Intuitively, the restriction on the form of production rules says that, if we interpret the grammar over the alphabet $\{\tau\,[\,] \mid \text{for some } k, k' \text{ there is a production } k \to \tau[k'] \text{ in } G\}$, it generates a regular set of words (as opposed, e.g., to context-free grammars). Then the automaton directly corresponding to the grammar reads the generated context in an outside-in manner, from the topmost symbol towards the hole.

By an *instance of a production* in P we mean the result of replacing in this production occurrences of variables from S with terms from the corresponding syntactic categories (where different occurrences of the same variable may be replaced with different terms). Note that there may be infinitely many instances of productions (because there may be infinitely many terms). The right-hand sides of instances of productions, with nonterminals replaced by holes, are called *elementary contexts*.

$$\begin{array}{rlrl} \text{(terms)} & t ::= x \mid \lambda x.\, t \mid t\, t & \text{(values)} & v ::= \lambda x.\, t \\ \text{(redices)} & r ::= (\lambda x.t)\, t & \text{(reduction contexts)} & \underline{C} ::= [\,] \mid C\, t \end{array}$$

$$(\beta\text{-contraction}) \quad (\lambda x.t_1)\, t_2 \rightharpoonup t_1[t_2/x]$$

Figure 1 The call-by-name reduction semantics for the lambda calculus.

$$(\lambda x.xx)s \xrightarrow{[\,]} ss \xrightarrow{([\,]\,(\lambda x.x))\,s} (\lambda y.y)(\lambda x.x)s \xrightarrow{[\,]\,s} (\lambda x.x)s \xrightarrow{[\,]} s \xrightarrow{[\,]\,(\lambda x.x)} (\lambda y.y)(\lambda x.x) \xrightarrow{[\,]} \lambda x.x$$

where $s = (\lambda x\; y.y)\; a\; (\lambda x.x)$ and a is any closed term

Figure 2 An example evaluation in the call-by-name strategy with reduction contexts indicated above reduction arrows.

Call-by-name strategy. As an example, let us consider the reduction semantics realizing the call-by-name reduction strategy for the lambda calculus. Figure 1 presents the grammar of terms, values and redices in this strategy, and the standard β-contraction relation. A reduction context is either the empty context $[\,]$ or a hole applied to a sequence of terms – as defined by the grammar, where $C\, t$ means that a context generated by C is applied to a term t. The process of evaluation prescribes that at each step we reduce the given term after we decompose it into a redex and a reduction context, unless the term is already a value or a stuck term. It can be seen that, in our setting, terms of the form $x\, t_1 \ldots t_n$ are stuck terms, as they are not proper values nor can they be further reduced. Under call by name, when we evaluate complete programs (which are closed terms), stuck terms cannot occur. Figure 2 shows an example evaluation in the call-by-name reduction semantics.

It is natural to require that a grammar has the *unique-decomposition property* meaning that for each term there exists at most one decomposition into a reduction context and a redex. A grammar with this property implicitly determines a reduction strategy understood as a function that for a given term finds a position of a redex in the term. In the following we will be interested in an explicit description of a lower-level strategy for finding redices. In the example of call-by-name semantics, this strategy prescribes that when we encounter an application, we should continue searching in the left subterm; and when we encounter a lambda abstraction (for which there is no corresponding production in the grammar), we should backtrack. Note that the decision what to do is based only on immediately available and local information (here: the topmost symbol of the processed term; in call by value also on some direct subterms being values). Strategies with this property are called *uniform*. Later we will see that in nonuniform strategies such a decision requires additional knowledge – such strategies can be thought of as composed from different substrategies.

Normal-order strategy. In this work we are interested in applying the refocusing procedure to more complex reduction semantics, such as the normal-order strategy in the lambda calculus. 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 strategy. An example evaluation in this strategy is given in Figure 3. The grammar of reduction contexts

$$(\lambda f\ x.\ f\ ((\lambda f\ x.\ x)\ f\ x))\ (\lambda x.\ g\ x) \xrightarrow{[\,]} \lambda x.\ (\lambda x.\ g\ x)\ ((\lambda f\ x.\ x)\ (\lambda x.\ g\ x)\ x) \xrightarrow{\lambda x.[\,]}$$
$$\lambda x.\ g\ ((\lambda f\ x.\ x)\ (\lambda x.\ g\ x)\ x) \xrightarrow{\lambda x.\ g\ ([\,]\ x)} \lambda x.\ g\ ((\lambda x.\ x)\ x) \xrightarrow{\lambda x.\ g\ [\,]} \lambda x.\ g\ x$$

Figure 3 An example reduction in the normal-order strategy with reduction contexts indicated above the arrows.

$$\begin{array}{rll} \text{(terms)} & t ::= \lambda x.\ t \mid x \mid t\ t, & \underline{E} ::= [\,]_E \mid \lambda x.E \mid F\ t \mid a\ E \\ \text{(neutral terms)} & a ::= x \mid a\ v & F ::= [\,]_F \mid F\ t \mid a\ E \\ \text{(normal terms)} & v ::= a \mid \lambda x.\ v & \end{array}$$

Figure 4 A grammar of reduction contexts for the normal-order strategy.

for this semantics is given in Figure 4; here the set $\mathcal{C}$ of reduction contexts is the language generated from the symbol E, and the contraction relation is β-reduction as in the case of call by name.

The grammar of contexts in Figure 4 again describes a strategy for finding redices. This time, however, this strategy is not uniform: we have two substrategies, one for each nonterminal symbol in the grammar. Each substrategy comes with its own kind of hole as introduced in Section 3.2 – 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 two presented substrategies agree on reacting to application symbols, but they differ on lambda abstractions. In the case of application symbols they say that when we encounter an application for the first time, we should continue searching in the left subterm with the F substrategy and when we backtrack from the left subterm (when the left subterm is a neutral term in the syntactic category a), we should continue searching in the right subterm with the substrategy E. In the case of lambda abstraction the E substrategy says that we should continue searching in the body of the abstraction while F says that we should backtrack.

In the following, to distinguish between different substrategies, we will use the notion of a *kind* that can be identified with a nonterminal symbol in the underlying grammar.

Uniform and hybrid strategies. The notion of hybrid strategies has been first used by Sestoft [16] informally, and recently studied by García-Pérez and Nogueira [11] who further distinguish between strategies "hybrid in style" and "hybrid in nature." Here we propose a simple definition of a hybrid strategy (that corresponds to being hybrid in style): it is a reduction strategy induced by a reduction semantics whose grammar of contexts has more than one nonterminal symbol (i.e., more than one kind of contexts). This is in contrast to uniform strategies, where the grammars have exactly one nonterminal symbol.

A uniform strategy allows free composition of elementary contexts. In the case of hybrid strategies the introduction of multiple kinds naturally leads to a restriction that when we compose two contexts, the kind of the internal one must be the same as the kind of the hole in the external one. Therefore free composition of elementary contexts may result in an invalid context when the two kinds do not match.

2.2 Abstract machines

Intuitively, abstract machines are just another form of operational semantics, only defined at a lower level of abstraction. Typically, abstract machines are devised in order to specify a semantics that is both precise and reasonably efficient. 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.

In our setting abstract machines are a kind of abstract rewriting systems of the form $\langle \mathcal{T}, \rightarrow, S, F \rangle$, where $\mathcal{T}$ is a set of configurations (states), $\rightarrow$ is a rewriting relation over $\mathcal{T}$ (the transition relation), and $S, F \subseteq \mathcal{T}$ are initial and final states, respectively. Moreover, we require that machines are deterministic, i.e., that transition relations are partial functions, and that final states are normal forms w.r.t. $\rightarrow$.

In this paper we work with abstract machines with two kinds of configurations. *Eval* configurations are of the form $\langle t, C \rangle_{\mathcal{E}}$ (or of the form $\langle t, C, k \rangle_{\mathcal{E}}$ if we work with hybrid strategies). A long-term goal of the machine in such a configuration is to evaluate the term $C[t]$; a shorter-term goal is to evaluate the term t (according to the substrategy k). The evaluation of t then starts by searching for a redex inside t. When the evaluation of t is finished returning a value v, the machine moves to a *continue* configuration of the form $\langle C, v \rangle_{\mathcal{C}}$ (or $\langle C, k, v \rangle_{\mathcal{C}}$ in the case of hybrid strategies). Again, a long-term goal of the machine in this configuration is to evaluate the term $C[v]$; a shorter-term goal is to find (by searching inside the context C) a decomposition of $C[v]$ into a new context C' and a new term t', and then to move to the configuration $\langle C', t' \rangle_{\mathcal{E}}$.

2.3 Refocusing in uniform strategies

Given a specification of a reduction semantics, a naive implementation of evaluation in this semantics would consist in repeating the following steps until the processed term is a normal form: (i) decompose the given term into a context and a redex, (ii) contract the redex, and (iii) recompose a new term by plugging the contractum in the context.

Refocusing is a mechanical procedure that optimizes this naive implementation by avoiding the reconstruction of intermediate terms in a reduction sequence. Intuitively, a considerable part of the decomposition work in step (i) simply cancels a considerable part of the recomposition work done in step (iii) of the previous iteration, and we would like to avoid this canceled work.

Refocusing (for uniform strategies) is based on the following general observation: *if we plug the contractum in the context (i.e., reconstruct) and then decompose, we obtain the same decomposition as when we continue decomposing directly from where we are.* Let us look at what happens just after contracting a redex in step (ii). There are two cases to be considered: the next redex can or cannot be found inside the contractum. For example, if we have a term $(\lambda x.(\lambda y.\, t_0)\ t_1\ t_2)\ t_3\ t_4\ \ldots\ t_n$, then it decomposes to the redex $(\lambda x.(\lambda y.\, t_0)\ t_1\ t_2)\ t_3$ and the reduction context $[\,]\ t_4\ \ldots\ t_n$ in the call-by-name strategy. But now if we perform the contraction (β-reduction), we do not need to reconstruct the term $((\lambda y.\, t_0)\ t_1\ t_2)[t_3/x]\ t_4 \ldots\ t_n$ and walk through all $t_4, \ldots, t_n$ again to find the next decomposition. It is enough to decompose the result of the contraction to the redex $((\lambda y.\, t_0)\ t_1)[t_3/x]$ and the context $[\,]\ t_2[t_3/x]$, and to plug this context into the previous one (i.e., $[\,]\ t_4\ \ldots\ t_n$).

In the second case (when the contractum does not contain a redex) we also can avoid reconstruction of the whole term, as we can just backtrack from the current reduction context,

reconstruct a part of the whole term and then decompose this part. For example, consider evaluation of the term $(\lambda x\ y.\ t_0)\ v_1\ ((\lambda x.\ t_2)\ v_3\ t_4)\ t_5\ \ldots\ t_n$. According to the left-to-right call-by-value strategy, it decomposes to the redex $(\lambda x\ y.\ t_0)\ v_1$ and the reduction context $[\,]\ ((\lambda x.\ t_2)\ v_3\ t_4)\ t_5\ \ldots\ t_n$. Now after the contraction we can reconstruct a small part of the whole term, namely $(\lambda y.\ t_0)[v_1/x]\ ((\lambda x.\ t_2)\ v_3\ t_4)$, leaving $[\,]\ t_5\ \ldots\ t_n$ as the reduction context and perform a decomposition of this part as before. So as a result we get the redex $(\lambda x.\ t_2)\ v_3$ and the reduction context $(\lambda y.\ t_0)[v_1/x]\ ([\,]\ t_4)\ t_5\ \ldots\ t_n$, and to obtain this decomposition we do not have to traverse all the terms $t_5\ \ldots\ t_n$.

These two ways of avoiding full reconstruction of intermediate terms during evaluation are the main idea behind refocusing. Specifically, a step of a machine generated by refocusing either implements the behavior described above, or it performs the contraction of a found redex. We omit a detailed presentation of the machine since it can be obtained from the one constructed in Section 4.2 by removing all kind information.

Unfortunately, the property emphasized above that underlies the original refocusing procedure no longer holds for hybrid strategies.

2.4 Refocusing in hybrid strategies

Before explaining the problems concerning refocusing in hybrid strategies, we need to discuss some aspects of the internal representation of contexts in implementations of reduction semantics. In general, reduction contexts of a reduction semantics coincide with evaluation contexts that are obtained by defunctionalizing continuations of a CPS-based interpreter implementing the semantics [1, 5]. This way one obtains a common "inside-out" representation of evaluation contexts, where a context is represented by a stack with topmost elements corresponding to the nearest surroundings of the hole and the bottommost – to the root of the context. This representation is also utilized by the refocusing procedure.

In stack representations of contexts the elements of a stack are elementary contexts introduced in Section 2.1. The meaning of such a stack is given by a *recompose* (or *plug*) function that recursively defines the result of inserting a term in the hole of a context. In the case of inside-out contexts this function is a left fold applied to an atomic recomposition function for elementary contexts.

The original refocusing procedure cannot be applied to hybrid strategies, because one of its prerequisites is that the underlying grammar has just one nonterminal symbol. As an example that backs this claim let us consider the strategy defined in Figure 5 and a configuration $\langle c\ t,\ C\rangle_{\mathcal{E}}$. In this configuration, according to E-substrategy the term $c\ t$ cannot be further decomposed and the machine should backtrack moving to the configuration $\langle C,\ c\ t\rangle_{\mathcal{C}}$. On the other hand, according to F-substrategy there might be further redices in t, so the machine should push the elementary context $c\ [\,]$ to the stack and start evaluating t, moving to the configuration $\langle t,\ c\ [\,] :: C\rangle_{\mathcal{E}}$. The choice of the substrategy depends on an occurrence of b in C. If the elementary context $b\ [\,]$ is present somewhere in the stack, the machine should choose the F-substrategy; otherwise, if $b\ [\,]$ is not present in the stack, it should choose the E-substrategy. This, however, requires additional knowledge of what is stored on the stack; a machine generated in the original refocusing procedure has only access to the very local information about the topmost symbol on the stack and the topmost symbol in the processed term (or, in a more general setting, to a bounded initial prefix of the stack and of the term), and simply does not collect this additional knowledge. It is not difficult to see that the strategy induced by this grammar is hybrid in nature – there does not exist a grammar with one nonterminal symbol generating the same set of reduction contexts.

A similar problem occurs when we deal with the normal-order strategy, where a generated machine in a state $\langle \lambda x.\ t,\ C\rangle_{\mathcal{E}}$ has to decide whether it should decompose the lambda

Strategy:

$\underline{E} ::= a\ E \mid b\ F$

$F ::= [\,] \mid a\ F \mid c\ F$

Context $C = \ldots a\ a\ [\,]$:

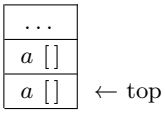

Figure 5 A small reduction strategy and a stack representation of a context.

abstraction. In the particular case of the normal-order strategy the problem can be solved ad hoc: the topmost elementary context in C always indicates the substrategy that should be used, and, in fact, this observation was used e.g. in [6] or [11]. The novelty of our approach is that it works not only in the case of normal-order strategy, but also in a much wider class of hybrid strategies. In Section 3.3 we give an example of a realistic language where the problem discussed above indeed occurs. Moreover, it cannot be solved by the ad hoc method that works for the normal-order strategy, because the choice of a substrategy depends on the occurrence of the ! symbol on the stack.

3 Reduction semantics revisited

In this section we present the first contribution of this work: a generalization of the concepts presented in Section 2.1 that enables refocusing for hybrid strategies.

3.1 Generalized reduction contexts

Our solution to the problem highlighted in Section 2.4 is based on changing the representation of evaluation contexts in generated machines. Specifically, we are going to represent contexts by sequences of instances of productions that generate the represented context. Note that this is not a big change in the representation of contexts from Section 2.4. There a context is represented as a stack (that is, a sequence) of elementary contexts, which are instances of right-hand sides of productions. Since a uniform grammar contains only one nonterminal symbol, this symbol (which is the left-hand side of each production) is implicitly present in each element of the stack. So now the only change in the representation of contexts is that we add an explicit occurrence of the left-hand side of the respective production rule to each element of the stack.

For example, consider the following derivation of the context $\lambda z.\ ((x\ y)\ [\,]_E\ z)$ in the grammar of contexts from Figure 4:

$$E \ \to\ \lambda z.\, E \ \to\ \lambda z.\, (F\ z) \ \to\ \lambda z.\ ((x\ y)\ E\ z) \ \to\ \lambda z.\ ((x\ y)\ [\,]_E\ z)\ .$$

Figure 6 shows the representation of this context. There is one subtlety here: for efficiency reasons (it is always better to pop one rather than two symbols from a stack) the last production of the form $k \to [\,]_k$ is not stored on the stack – it is better to store the nonterminal k in the internal state of the generated abstract machine.

In our generalization of refocusing the additional nonterminal symbol indicates the substrategy that should be used in a given place. Recall the example from Figure 5. A generalized stack representation of the context C with the topmost symbol $\langle E, a\ [\,]\rangle$ indicates that the E-substrategy should be used. It can be seen that this symbol is on the top of stack if and only if there is no occurrence of b in C. Similarly, $\langle F, a\ [\,]\rangle$ indicates F-substrategy;

$E,$	$\lambda z.\,[\,]_E$	
$E,$	$[\,]_F\ \ z$	
$F,$	$(x\ y)\ [\,]_E$	$\leftarrow$ top

Figure 6 The representation of the context $\lambda z.\ \big((x\ y)\ [\,]_E\ z\big)$ w.r.t. the grammar in Fig. 4.

this symbol is on top of stack iff there is an occurrence of b in C. Thus nonterminal symbols stored on stack give the additional knowledge required to choose the appropriate substrategy.

3.2 Grammars of contexts

Here we specify conditions on grammars of contexts required by the generalized refocusing procedure. We first extend the definition of a grammar from Section 2.1 by introducing holes of different kinds (recall that kinds are nonterminal symbols in the grammar) and then restrict the form of production rules.

▶ **Definition 1.** A grammar of contexts G is called *normal* if

1. for each production $k \to \tau$ in the grammar, either τ is a hole or
   - **a.** τ does not contain a hole, and
   - **b.** the unique occurrence of a nonterminal symbol in τ is a direct subterm of τ, and
2. for each nonterminal k, there is a production $k \to [\,]_k$.

Intuitively, a normal grammar generates a prefix-closed set of contexts (i.e., a prefix of a context is also a context). The restriction about direct subterms will be used by the generated abstract machine, which will be analyzing just one symbol at a time. This condition is not crucial and it is not difficult to provide a more general setting in which an abstract machine has access to a bounded initial prefix of the stack and bounded initial prefix of the processed term. In such a setting more strategies can be qualified as uniform. Note also that we introduce here holes indexed with nonterminal symbols – this will be used to keep track of which nonterminal generates given occurrence of a hole. We will refer to $[\,]_k$ as *the hole of kind* k.

We will also require that grammars have the unique-decomposition property. Since this is a semantic restriction that is difficult to check for a given grammar, in Section 4.1 we will pose additional restrictions, in the form of existence of some orders on productions, that entail this property.

3.3 Generalized reduction semantics

In our implementation of the refocusing procedure we use a more flexible definition of reduction semantics than the one given in Section 2.1. It allows to define different kinds of contractions to be used in different kinds of holes in contexts. Thus a generalized reductions semantics is a family of reduction semantics over one language that allows to reduce a term if any sub-semantics allows it.

▶ **Definition 2.** A tuple of the form $\langle \mathcal{T}, K, \{\mathcal{C}_k\}_{k\in K}, \{\rightharpoonup_k\}_{k\in K}, \{V_k\}_{k\in K}\rangle$, where

- $\mathcal{T}$ is a language,
- K is a set of reduction kinds, and
- $\langle \mathcal{T}, \mathcal{C}_k, \rightharpoonup_k, V_k\rangle$ is a reduction semantics for each $k \in K$,

$\mathcal{T} = t ::= x \mid \lambda x.\, t \mid t\ t \mid !t$

$K = \{E,\ F\}$

$(\lambda x.\, t_1)\ t_2 \rightharpoonup_E t_1[t_2/x]$

$!v \rightharpoonup_E v$

$(\lambda x.\, t)\ v \rightharpoonup_F t[v/x]$

$!t \rightharpoonup_F t$

$C \in \mathcal{C}_k \iff C[[\,]_k] \in \mathcal{L}(E) \quad \text{for } k \in K$

$\underline{E} ::= [\,]_E \mid E\ t \mid !F$

$F ::= [\,]_F \mid F\ t \mid v\ F$

$V_E = w ::= a \mid \lambda x.\, t$

$V_F = v ::= b \mid \lambda x.\, t$

$a ::= x \mid a\ t$

$b ::= x \mid b\ v$

Figure 7 Lambda calculus with the call-by-name strategy and a strictness operator.

is called a *generalized reduction semantics*. A term t_1 can then be *reduced* to a term t_2 in this semantics, written $t_1 \to t_2$, if $t_1 = C[s_1]$, $t_2 = C[s_2]$ and $s_1 \rightharpoonup_k s_2$ for some $k \in K$, $s_1,\ s_2 \in \mathcal{T}$ and $C \in \mathcal{C}_k$.

Notions given for reduction semantics generalize in a straightforward way to the generalized ones. We call elements of sets $\mathcal{C}_k$ and V_k, respectively, *(reduction) contexts with holes of kind* k and *values of kind* k. The contraction $\rightharpoonup_k$ is called *k-contraction* and terms that can be rewritten by $\rightharpoonup_k$ are called *redices of kind* k. A term t is called a *potential redex of kind* k if t is not a value of kind k and in every decomposition $t = C[t']$ of t with a nonempty context $C \in \mathcal{C}_k$, the subterm t' is a value of kind k. Intuitively, each non-value term has a (unique) decomposition into a reduction context and a potential redex.

To define families $\{\mathcal{C}_k\}_{k \in K}$ we will use normal grammars of contexts. Formally, we require that the set of context nonterminals coincides with the set K of context kinds and that $C \in \mathcal{C}_k$ if and only if $C[[\,]_k]$ is generated by the grammar.

In Figure 7 we give an example that makes use of the additional expressiveness given by generalized reductions semantics. It presents a lambda calculus with the call-by-name strategy and an extra operator ! that switches the strategy to left-to-right call by value. Here E-substrategy corresponds to call by name and F – to call by value. Note that here E-contraction is different from F-contraction.

4 Generalized refocusing

In this section we present the second main contribution of this work: a generalization of the refocusing method.

4.1 Input to the refocusing procedure

In order to run the generalized refocusing procedure, the user needs to define generalized reduction semantics together with a strategy for searching redices. Given this input, the procedure generates an abstract machine.

We now make precise the requirements on the user input. For a set of productions P, notation $P^{k_1}_{k_2}$ stands for the set of instances of productions from P of the form $k_1 \to ec[k_2]$, where ec is an elementary context. Similarly, P^k stands for instances of the form $k \to \ldots$, that is, with the nonterminal k on the left-hand side of the arrow. We say that an instance $k_1 \to ec[k_2]$ is *compatible* with t if ec is a prefix of t.

▶ **Definition 3.** An *input for generalized refocusing* is composed from

- a generalized reduction semantics $\langle \mathcal{T}, K, \{\mathcal{C}_k\}_{k\in K}, \{\rightharpoonup_k\}_{k\in K}, \{V_k\}_{k\in K}\rangle$ where the family $\{\mathcal{C}_k\}_{k\in K}$ is given by a normal grammar of contexts with the set of productions P,
- for each $k \in K$ and $t \in \mathcal{T}$, a linear strict order $<_{k,t}$ on instances of productions from P^k that are compatible with t, and
- computable functions $\Downarrow$: $\mathcal{T} \times K \to P \times \mathcal{T} \cup \{\mathtt{R}, \mathtt{V}\}$ and $\Uparrow$: $\prod_{k_1,k_2\in K} \left(P^{k_1}_{k_2} \times V_{k_2}\right) \to P \times \mathcal{T} \cup \{\mathtt{R}, \mathtt{V}\}$ (where R tags redices and V tags values)

such that all of the following conditions are satisfied:

1. All $\rightharpoonup_k$ are partial computable functions.
2. Instances of productions $k \to ec_1[k_1]$ and $k \to ec_2[k_2]$ are comparable by $<_{k,t}$ if and only if ec_1, ec_2 are different prefixes of t.
3. If $(k{\to}ec_1[k_1]) <_{k,t} (k{\to}ec_2[k_2])$ and $t = ec_2[v]$, then v is a value of kind k_2.
4. If the order $<_{k,t}$ is empty, then the value of $\Downarrow (t, k)$ is either R or V, depending on whether t is a potential redex or a value of kind k. Otherwise, if the order is nonempty, the value of $\Downarrow (t, k)$ is the greatest element $k \to ec[k']$ in this order, together with the subterm t' of t such that $t = ec[t']$.
5. If the instance $k_1{\to}ec[k_2]$ is the least element in the order $<_{k_1,ec[v]}$, then the value of $\Uparrow (k_1, k_2, (k_1{\to}ec[k_2]), v)$ is either R or V, depending on whether $ec[v]$ is a potential redex or a value of kind k. Otherwise, if $k_1{\to}ec[k_2]$ is not the least element, the value of $\Uparrow (k_1, k_2, (k_1{\to}ec[k_2]), v)$ is the predecessor $k_1{\to}ec'[k'_2]$ of $k_1{\to}ec[k_2]$ in the order $<_{k_1,ec[v]}$, together with the subterm t' of $ec[v]$ such that $ec[v] = ec'[t']$.

Note that the introduced orders describe a depth-first search order of evaluation. More precisely, suppose that $(k{\to}ec_1[k_1]) <_{k,t} \ldots <_{k,t} (k{\to}ec_n[k_n])$ are all productions in P^k compatible with t, sorted in the order $<_{k,t}$. Then for some terms $t_1, \ldots, t_n$ we have $t = ec_1[t_1] = \ldots = ec_n[t_n]$. Condition (4) says that k-substrategy starts searching for redices in term t_n. Condition (5) says that in consecutive iterations (for $i = n-1, \ldots, 1$) this strategy moves the search from t_{i+1} to t_i, and condition (3) says that it happens only after completely traversing t_{i+1} and finding it to be a value. The functions $\Downarrow$, $\Uparrow$ describe a DFS order of evaluation in a computable way. They also determine if a term is a potential redex or (after checking all possible decompositions w.r.t. a given substrategy) if it is a value of a given kind.

It is worth noting that if we want the generated abstract machine to execute each step in constant time, all of the mentioned computable functions should also work in constant time.

We give an example input for the generalized refocusing procedure in Figures 7 and 8. The former contains a generalized semantics and the latter contains an order and functions $\Downarrow$ and $\Uparrow$. The functions are presented in a relational style, with arguments on the left-hand side and results on the right-hand side of the arrows. The notation $[t_1]_{k_1}$ stands for (t_1, k_1), the notation $\left[ec\ [v]_{k_2}\right]_{k_1}$ for $(k_1, k_2, (k_1{\to}ec[k_2]), v)$ and the notation $ec\ [t_2]_{k_3}$ on a right-hand side of the arrow $\Uparrow$ involving k_1 as the leftmost argument stands for $((k_1{\to}ec[k_3]), t_2)$.

The result of $[t_1]_{k_1} \Downarrow$ prescribes how to proceed when a term t is considered by a k_1-substrategy for the first time. If it returns $ec\ [t_2]_{k_2}$, the search should be continued in the subterm t_2 by the k_2-substrategy and the current evaluation context should be extended by ec. Otherwise there is no further decomposition of t and it is determined if it is a value or a potential redex. For example, $[!t]_E \Downarrow !\ [t]_F$ prescribes that when the E-substrategy encounters a term $!t$, it has to decompose t according to the F-substrategy with the current evaluation contexts extended by $!\,[\,]$.

The result of $\left[ec_1\ [v]_{k_2}\right]_{k_1} \Uparrow$ defines how to proceed when a term $ec_1[v]$ is reconsidered by a k_1-substrategy after completing the search in the subterm v w.r.t. the k_2-substrategy and finding that it is a value of kind k_2. The options are the same as in the case of $\Downarrow$.

The order on instances of productions:
$(F \to v\ F) <_{F,(v\ t)} (F \to F\ t)$ for all values v of kind F and all terms t

Functions $\Downarrow$ and $\Uparrow$, where $k \in \{E, F\}$:

$$
\begin{array}{rr}
[x]_k \Downarrow \mathtt{V} & [\ [\lambda x.t_1]_E\ t_2]_E \Uparrow \mathtt{R} \\
[\lambda x.t]_k \Downarrow \mathtt{V} & [\ [a]_E\ t]_E \Uparrow \mathtt{V} \\
[t_1\ t_2]_k \Downarrow\ [t_1]_k\ t_2 & [!\ [v]_F\ ]_E \Uparrow \mathtt{R} \\
[!t]_E \Downarrow !\ [t]_F & [\ [v]_F\ t]_F \Uparrow v\ [t]_F \\
[!t]_F \Downarrow \mathtt{R} & [\lambda x.t\ [v]_F\ ]_F \Uparrow \mathtt{R} \\
 & [b\ [v]_F\ ]_F \Uparrow \mathtt{V}
\end{array}
$$

Figure 8 An input for generalized refocusing that formalizes the semantics from Figure 7.

4.2 Output from the refocusing procedure

The machines obtained by the generalized refocusing procedure have the form given in the definition below. The notion $ConRep_{G,k}$ in this definition is the set of generalized stack representations (cf. Section 3.1) of contexts in $\mathcal{C}_k$ w.r.t. a grammar G. To distinguish contexts and their representations we use the typewriter font for the latter. The configuration labels $\mathcal{E}$, $\mathcal{C}$ historically stand for "eval" and "continue".

▶ **Definition 4.** For a given input as in Definition 3, the *machine generated by generalized refocusing* is $\langle \mathcal{S}, \to, \{\langle t, \varepsilon, k_0\rangle_{\mathcal{E}} \in \mathcal{S}\},\ \{\langle \varepsilon, k_0, v\rangle_{\mathcal{C}} \in \mathcal{S}\}\rangle$, where

$$\mathcal{S} = \{\ \langle t, \mathtt{C}, k\rangle_{\mathcal{E}}, \langle \mathtt{C}, k, v\rangle_{\mathcal{C}} \mid t \in \mathcal{T}, k \in K, \mathtt{C} \in ConRep_{G,k}, v \in V_k\}$$

and $\to$ is defined as follows.

$$
\begin{array}{ll}
\langle t, \mathtt{C}, k\rangle_{\mathcal{E}} \to \langle \mathtt{C}, k, t\rangle_{\mathcal{C}} & \text{if } [t]_k \Downarrow \mathtt{V}, \\
\langle t_1, \mathtt{C}, k\rangle_{\mathcal{E}} \to \langle t_2, \mathtt{C}, k\rangle_{\mathcal{E}} & \text{if } [t]_k \Downarrow \mathtt{R} \wedge t_1 \rightharpoonup_k t_2, \\
\langle t_1, \mathtt{C}, k_1\rangle_{\mathcal{E}} \to \langle t_2, (k_1, ec) :: \mathtt{C}, k_2\rangle_{\mathcal{E}} & \text{if } [t_1]_{k_1} \Downarrow ec\ [t_2]_{k_2}, \\
\langle (k_1, ec) :: \mathtt{C}, k_2, v\rangle_{\mathcal{C}} \to \langle \mathtt{C}, k_1, ec[v]\rangle_{\mathcal{C}} & \text{if } \left[ec\ [v]_{k_2}\ \right]_{k_1} \Uparrow \mathtt{V}, \\
\langle (k_1, ec) :: \mathtt{C}, k_2, v\rangle_{\mathcal{C}} \to \langle t, \mathtt{C}, k_1\rangle_{\mathcal{E}} & \text{if } \left[ec\ [v]_{k_2}\ \right]_{k_1} \Uparrow \mathtt{R} \wedge ec[v] \rightharpoonup_{k_1} t, \\
\langle (k_1, ec_1) :: \mathtt{C}, k_2, v\rangle_{\mathcal{C}} \to \langle t, (k_1, ec_2) :: \mathtt{C}, k_3\rangle_{\mathcal{E}} & \text{if } \left[ec_1\ [v]_{k_2}\ \right]_{k_1} \Uparrow ec_2\ [t]_{k_3}.
\end{array}
$$

In fact, in configurations of the form $\langle \mathtt{C}, k, v\rangle_{\mathcal{C}}$ symbol k has no computational meaning and could be safely removed. We leave it here for the sake of simplicity.

Note that the construction of the machine does not depend on conditions (1)–(5) of Definition 3, it depends only on the functions $\Downarrow$, $\Uparrow$ and contractions. The conditions are needed to guarantee that the constructed machine realizes the input reduction semantics.

5 Correctness of the generated machine

In [17] the authors prove that the original refocusing procedure generates an abstract machine that is extensionally equivalent to the input semantics. Formally, a term t can be reduced (possibly in many steps) to a value v w.r.t. an input reduction semantics if and only if the

generated machine started in $\langle t, \varepsilon\rangle_{\mathcal{E}}$ evaluates to $\langle \varepsilon, v\rangle_{\mathcal{C}}$. Here we show a stronger property, namely that the generated machine exactly implements the reduction semantics given as input, possibly in smaller steps. This idea is captured in the following definition of *tracing*.

▶ **Definition 5.** An abstract rewriting system $\langle \mathcal{T}, \to\rangle$ **traces** another system $\langle \mathcal{S}, \Rightarrow\rangle$ if there exists a surjection $[\![]\!] : \mathcal{T} \to \mathcal{S}$ such that

1. if $t_1 \to t_2$, then $[\![t_1]\!] = [\![t_2]\!]$ or $[\![t_1]\!] \Rightarrow [\![t_2]\!]$,
2. if $s_1 \Rightarrow s_2$, then for each t_0 such that $[\![t_0]\!] = s_1$ there exists a sequence $t_0 \to \ldots \to t_{n+1}$, where $[\![t_0]\!] = \ldots = [\![t_n]\!]$ and $[\![t_{n+1}]\!] = s_2$, and
3. there are no infinite sequences $t_0 \to t_1 \to \ldots$, where $[\![t_n]\!] \not\Rightarrow [\![t_{n+1}]\!]$ for all n (i.e., there are no silent loops).

The definition is similar to the one used by Hardin et al. to extract reduction strategies in a calculus of closures from virtual machines [13]. Although in this paper we work only with deterministic systems, it is worth mentioning that our definition, as opposed to the one from [13], is also adequate for nondeterministic systems.

The following theorem states that each generated machine traces the input reduction semantics. It is proved in Coq, the proof can be found in the implementation in file `refocusing/refocusing_machine_facts.v`.

▶ **Theorem 6.** *Let M be the machine generated by generalized refocusing procedure from a reduction semantics with the set of terms $\mathcal{T}$ and the reduction relation $\to$. Then M traces $\langle \mathcal{T}, \to\rangle$.*

6 Implementation

Our implementation of generalized refocusing is available at the repository `http://bitbucket.org/kzi/generalized_refocusing`. The current (at the moment of writing this article) version of the code, including examples, consists of over 8000 lines of definitions and proofs. It was tested in Coq versions 8.5 and 8.6. The repository contains several examples, including a novel derivation of a machine with an environment for full β-normalization from a language with explicit substitutions (see file `lam_cl_es_no.v`). We suggest to start with simpler examples of lambda calculus with the call-by-name and call-by-value strategies in files `cbn_lam.v` and `cbv_lam.v`. A formalization of lambda calculus with the call-by-name strategy and the strictness operator from Figure 7 can be found in `cbn_strict.v` file. In three cases: MiniML, lambda calculus with the normal-order strategy, and lambda calculus with the normal-order strategy and simple explicit substitutions the examples also contain a manually defined machine and a proof of equivalence between this machine and the automatically generated one.

In the following we briefly describe the implementation of the semantics provided by a user as input to generalized refocusing, as defined in Definition 3. The user needs to define two modules: one that implements the reduction semantics (which satisfies the signature `PRE_REF_SEM`), and one that implements the lower-level reduction strategy (which satisfies the signature returned by `REF_STRATEGY`). Based on these modules, the functor `RedRefSem` produces a module defining the refocusable semantics; in particular, it automatically proves crucial decomposition lemmas required by refocusing. Then the functor `RefEvalApplyMachine` applied to the refocusable semantics module produces an abstract machine. Finally, the functor `RefEvalApplyMachine_Facts` applied to the semantics and the machine provides the proof of Theorem 6. The proof itself is an instance of `RW_TRACING` type-class defined in the file `rewriting_system/rewriting_system_tracing.v`.

We show an excerpt of the relevant signatures in Figure 9. In the semantics module, the parameters (such as terms, kinds, potential redices, or values) are provided by the user, and their types reflect their dependence on kinds. There is some technical burden caused by Coq here: the user has to define values and potential redices as types disjoint from terms, and provide injections from these types into terms. We also require the user to define elementary contexts that depend on two kinds: the first one is the nonterminal on the left-hand side of the underlying production, and the second one is the kind of the hole (this is the nonterminal occurring in the right-hand side of the production). The reduction contexts can then be represented as lists of elementary contexts that agree on kinds, as described in Section 3.1. In addition, the user has to provide proofs for the axioms ensuring that potential redices and values can have only trivial decompositions, and that values cannot be potential redices.

The implementation of the strategy consists in defining two functions: `dec_term` and `dec_context` that correspond to the $\Downarrow$ and $\Uparrow$ functions, respectively. The types of these functions make use of the `elem_dec` type of elementary decompositions. The user then needs to specify the search order on elementary contexts together with proofs that this order is well founded, transitive and that only and all immediate prefixes of a term are comparable. With these properties in place the user proceeds to proving properties required by Definition 3. The axioms at the bottom of the listing in Figure 9 correspond to these properties (due to lack of space some axioms are omitted): `elem_context_det` implements condition 3, `dec_term_term_top` implements the second part of condition 4; `dec_context_red_bot`, `dec_context_val_bot` and `dec_context_term_next` correspond to condition 5.

7 Conclusion and perspectives

We have proposed a new format for specifying reduction semantics that utilizes grammars of kinded reduction contexts. Based on this format, we have developed a generic refocusing procedure that handles a wide class of hybrid rewriting strategies satisfying simple initial conditions. The procedure is formalized in Coq proof assistant and proved to correctly translate reduction semantics to abstract machines. The original refocusing procedure, as described in [17], can be obtained from the proposed here by restricting the set of context kinds to a singleton and optimizing out dependencies on its elements.

The current formalization has several limitations that we plan to address in future work. For example, it does not account for context-sensitive reduction strategies – such as those defining control operators – where the contraction function takes into account the current reduction context (this is different from the situation described in this paper where the *decomposition* function depends on the kind of the current reduction context). Moreover, it is not possible to define layered reduction semantics that would lead to an abstract machine with several layers of stacks representing contexts (such as 2-layered semantics for the control operators shift and reset[2]). A limitation of a different nature is that refocusing is based on the assumption that the semantics is deterministic. It would be interesting to see how this restriction could be relaxed and lead to a systematic way of deriving nondeterministic abstract machines.

Acknowledgments. We would like to thank the anonymous reviewers of FSCD for their helpful and insightful comments.

```
Module Type PRE_REF_SEM.

  Parameters (term ckind : Set) (init_ckind : ckind) (redex value : ckind → Set).
  Parameter contract : ∀ {k}, redex k → option term.
  Parameters (elem_context_kinded : ckind → ckind → Set)
             (elem_plug : ∀ {k0 k1}, term → elem_context_kinded k0 k1 → term).
  Notation "ec :[ t ]" := (elem_plug t ec) (at level 0).

  Inductive context (k1 : ckind) : ckind → Set :=
  | empty : context k1 k1
  | ccons : ∀ {k2 k3} (ec : elem_context_kinded k2 k3), context k1 k2 → context k1 k3.
  Notation "c [ t ]" := (plug t c) (at level 0).

  Definition reduce k t1 t2 := ∃ {k'} (c : context k k') (r : redex k') t,
                               dec t1 k (d_red r c) ∧ contract r = Some t ∧ t2 = c[t].
  Axioms
    (redex_trivial1 : ∀ {k k'} (r : redex k) ec t, ec:[t] = r → ∃ (v : value k'), t = v)
    (value_trivial1 : ∀ {k1 k2} ec {t} (v : value k1), ec:[t]=v → ∃ (v' : value k2), t = v')
    (value_redex : ∀ {k} (v : value k) (r : redex k), value_to_term v <> redex_to_term r).
  ...
End PRE_REF_SEM.

Module Type REF_STRATEGY (S: PRE_REF_SEM).
  Import S.
  Inductive elem_dec k : Set :=
  | ed_red : redex k → elem_dec k
  | ed_dec : ∀ k', term → elem_context_kinded k k' → elem_dec k
  | ed_val : value k → elem_dec k.

  Parameter dec_term: term → ∀ k, elem_dec k.
  Parameter dec_context {k k'} (ec:elem_context_kinded k k') (v:value k') : elem_dec k.
  Parameter search_order: ∀ k, term→ elem_context_in k→ elem_context_in k → Prop.
  Notation "t |∼ ec1 « ec" := (search_order _ t ec1 ec2).

  Definition so_maximal {k} (ec : elem_context_in k) t := ∀ ec', ∼t |∼ ec « ec'.
  Definition so_minimal {k} (ec : elem_context_in k) t := ∀ ec', ∼t |∼ ec' « ec.
  Definition so_predecessor {k} ec0 ec1 t :=
    t |∼ ec0 « ec1 ∧ ∀ (ec : elem_context_in k), t |∼ ec « ec1 → ∼t |∼ ec0 « ec.

  Axioms
    (elem_context_det : ∀ {k0 k1 k2} t ec0 ec1,
                  t |∼ ec0 « ec1 → ∃ (v : value k2), t = ec1:[v])
    (dec_term_term_top : ∀ {k k'} t t' ec,
                  dec_term t k = ed_dec _ t' ec → so_maximal ec t)
    (dec_context_red_bot : ∀ {k k'} (v : value k') {r : redex k} ec,
                  dec_context ec v = ed_red r → so_minimal ec ec:[v])
    (dec_context_val_bot : ∀ {k k'} (v : value k') {v' : value k} ec,
                  dec_context ec v = ed_val v' → so_minimal ec ec:[v])
    (dec_context_term_next : ∀ {k0 k1 k2} (v : value k1) t ec0 ec1,
                  dec_context ec0 v = ed_dec _ t ec1 → so_predecessor ec1 ec0 ec0:[v])
  ...
End REF_STRATEGY.
```

Figure 9 Signatures for refocusable semantics (fragments).

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.
2. Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005.
3. Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.
4. Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007. A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming.
5. Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, Victoria, British Columbia, September 2008. ACM, ACM Press. Invited talk.
6. Olivier Danvy and Jacob Johannsen. From outermost reduction semantics to abstract machine. In Gopal Gupta and Ricardo Peña, editors, *23rd International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2013*, pages 91–98. Springer-VS, 2014. doi:10.1007/978-3-319-14125-1.
7. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.
8. Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.
9. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.
10. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Benjamin C. Pierce, editor, *Proceedings of the Thirty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 153–164. ACM Press, January 2009.
11. A García-Pérez and Pablo Nogueira. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Science of Computer Programming*, 95:176–199, 2014.
12. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, September 2002. ACM, ACM Press.
13. Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.
14. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
15. Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

16 Peter Sestoft. Demonstrating lambda calculus reduction. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 420–435. Springer-Verlag, 2002.

17 Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: a generic formalization of refocusing in Coq. In Juriaan Hage and Marco T. Morazán, editors, *The 22nd International Conference on Implementation and Application of Functional Languages (IFL 2010)*, number 6647 in Lecture Notes in Computer Science, pages 72–88, Alphen aan den Rijn, The Netherlands, September 2010. Springer-Verlag.