# Generalized Refocusing: a Formalization in Coq

Klara Zielińska and Małgorzata Biernacka

*Institute of Computer Science*
*University of Wrocław*
*Wrocław, Poland*
{kzi,mabi}@cs.uni.wroc.pl

Abstract

We report on 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 implement the reduction strategy of the input semantics. The procedure and the correctness proofs have been formalized in the Coq proof system.

*Keywords:* reduction semantics, abstract machines, formal verification, Coq

## 1 Introduction

Refocusing has been introduced by Danvy and Nielsen as a generic procedure to derive an efficient abstract machine implementation from a given reduction semantics [5]. 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 [2,6]. Sieczkowski et al. then proposed an axiomatization of reduction semantics sufficient to apply the method and formalized the entire procedure in Coq [11].

However, the refocusing procedure as described previously does not account for a large class of hybrid rewriting strategies, notably including the normal-order strategy for full normalization in the lambda calculus. This strategy is of particular interest due to its use in type checking algorithms for dependently typed programming languages and logic systems, such as Coq [8].

The problem with applying refocusing to normal order and other similar strategies has been observed by Danvy and Johannsen [4], and by García-Pérez and Nogueira [7], 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 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 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 any rewriting strategy satisfying certain natural conditions and that is a generalization of Sieczkowski et al.'s formalization. The work is still in progress and in this paper we present its main contributions. The most significant one is a Coq formalization of generalized refocusing that is applicable to hybrid strategies such as normal order by utilizing a novel approach to representing evaluation contexts. We also use a variant of a bisimulation to closely relate abstract machines and reduction semantics.

Our implementation of refocusing is available at the repository https://github.com/klara-zielinska/refocusing2. It is powerful enough to generate efficient and realistic abstract machines. The repository contains several examples, including a novel derivation of a machine with an environment for full $\beta$-normalization from a language with explicit substitutions. This work also enables us to verify the correctness of other existing abstract machines.

The rest of this paper is structured as follows. In Section 2 we sketch the idea of the original refocusing procedure that is applicable to uniform strategies. In Section 3 we present our contribution: a generalization of reduction contexts representation, of reduction semantics and of the refocusing method to handle hybrid strategies, and we illustrate it with the normal-order strategy in the lambda calculus. In Section 4 we discuss the correctness of the generated machine and in Section 5 we describe the implementation.

## 2 Preliminaries

### *2.1 Reduction semantics*

A *reduction semantics* is a kind of small-step operational semantics with an explicit representation of reduction contexts. Usually, it is defined by a set of reduction contexts $\mathcal{C}$ given by a grammar of contexts, and a binary, local rewriting relation $\rightharpoonup$, also called *contraction*, over terms of the considered language. A computation step in such semantics, also called a *reduction*, is defined as rewriting in a reduction context, i.e., we can perform a reduction $C[t_1] \to C[t_2]$ iff $t_1 \rightharpoonup t_2$ and $C \in \mathcal{C}$, where $C[t_1]$ is a *decomposition* of the program into a context $C$ and a redex $t_1$.

Reduction contexts can be seen as defunctionalized continuations of one-step reduction functions and thus as a representation of "the rest of the reduction" in a reduction sequence. They coincide with evaluation contexts that in turn can be obtained by defunctionalizing continuations of an interpreter in the CPS form, as demonstrated by Danvy et al. [1,3]. This way one obtains the common "inside-out" representation of contexts. The meaning of this representation is given by a *recompose* function that recursively defines the result of inserting a term in the hole of a context. In the implementation, it is often convenient to treat reduction contexts as stacks of elementary contexts (context frames) where we can define *recompose* as a left fold using an atomic recomposition function for context frames.

In a uniform strategy, such as call-by-name or call-by-value evaluation in the lambda calculus, reduction contexts can be adequately represented using one syn-

$$\underline{E} ::= [\,]_E \mid \lambda x.E \mid F\ t \mid a\ E$$
$$F ::= [\,]_F \mid F\ t \mid a\ E$$

where
$t ::= \lambda x.\ t \mid x \mid t\ t,$
$a ::= x \mid a\ v, \quad v ::= a \mid \lambda x.\ v$

Figure 1. A grammar of reduction contexts for the normal-order strategy, where the starting symbol $E$ is underlined

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

Figure 2. An example reduction in the normal-order strategy with reduction contexts indicated above the arrows

tactic category (in other words, they can be defined by a grammar with one non-terminal context symbol) and they can be freely composed.

In this paper we are especially interested in a reduction semantics implementing the normal-order strategy in the lambda calculus. This strategy normalizes a term to its $\beta$-normal form (if it exists) by first evaluating it to its weak head normal form with the call-by-name strategy (i.e., leftmost outermost $\beta$-reduction), and only then reducing subterms of the resulting weak-head normal form with the same strategy. The grammar of reduction contexts for this semantics is the one given in Figure 1 in a variant with labeled holes (labels will be explained in Section 3.2). In this case the set $\mathcal{C}$ of reduction contexts is the language generated from the symbol $E$, and the contraction is just $\beta$-reduction, i.e., $(\lambda x.t_1)\ t_2 \rightharpoonup t_1[t_2/x]$ (capture-avoiding).

### *2.2 Refocusing*

Given a specification of a reduction semantics, a naive implementation of evaluation in this semantics consists in repeating the following steps: 1) decompose a given term into a context and a redex, 2) contract the redex, 3) recompose a new term by plugging the result of the contraction in the context.

Refocusing is a mechanical procedure that optimizes this naive implementation by avoiding the reconstruction of the intermediate terms in a reduction sequence. It builds on the following property of reduction semantics: if we plug a term in a context (i.e., reconstruct) and then decompose, we obtain the same decomposition as when we continue decomposing directly from where we are. Unfortunately, this property no longer holds if we try to apply refocusing as is to hybrid strategies.

The original refocusing procedure, like in [11], can be obtained from the proposed here generalized one by restricting the sets $K$ of context kinds (that will be introduced later) to singletons and optimizing out dependencies on elements of $K$.

### *2.3 Abstract machines*

In our setting abstract machines are abstract rewriting systems $(\mathcal{T}, \rightarrow)$, where $\mathcal{T}$ is a set of configurations (states), $\rightarrow$ is a computable function, and initial and final states are explicitly specified. They can be thought of as models of implementation for reduction in the source calculus. Ideally, we would like $\rightarrow$ to be a *small function*, i.e., to be computable in constant time.

# 3 Refocusing for hybrid strategies

In this section we present the contribution of this work: a generalization of the concepts presented in Section 2 that enables refocusing for hybrid strategies.

## *3.1 Generalized reduction contexts*

Hybrid strategies can be thought of as strategies where the composition of elementary contexts may result in a context that is not a valid reduction context for the strategy. In terms of stacks of context frames it means that not every sequence of elementary contexts represents a valid reduction context. For example, if we put the frame $\lambda x.[\,]$ on top of a stack that has the frame $[\,]\,t$ on its top, the resulting stack does not represent a valid normal-order reduction context.

More precisely—assuming that we have a definition of elementary contexts—uniform strategies can be defined as those for which there exists a set of elementary contexts $EC$ such that $EC^*$ is the set of all reduction contexts (where $EC^*$ means the smallest set containing $EC$ and closed under composition). In this approach, hybrid strategies can be defined as those that are not uniform, yet whose reduction contexts can be defined by a (regular [1]) grammar of contexts. This also implies that the grammar requires more then one syntactic category (non-terminal symbol). [2]

Note that switching the starting symbol in a grammar of contexts may, and usually does, change the strategy (cf. Figure 1, the $E$ symbol allows to go under lambdas, while $F$ prevents it). Let us call a strategy obtained by changing the symbol a *substrategy* of a given reduction semantics.

Because not all elements of $EC^*$ are reduction contexts in hybrid strategies, the original refocusing procedure cannot be directly applied to them. One problem is shown in Figure 3 (for simplicity, the grammar is very abstract). In a machine generated by the procedure a state is composed from a current evaluation context stored as a stack (as in the figure) and a representation of a term that is plugged in the context. Thus, for a state containing the presented context and a redex any such machine cannot decide in a small step if it can contract the redex, which depends on an occurrence of $b$ in the stack (refocusing provides no facility to scan the stack in multiple steps). The second problem is that by looking at such a stack the machine cannot immediately determine which substrategy it should use to proceed.

Our solution is to add context non-terminal symbols to the stack representations of contexts, so that they remember not only contexts themselves, but also their derivations in the grammar. An example of such a representation for a normal-order context is given in Figure 4. To make such an approach valid, we additionally require (1) that in each production $k \to \pi$ from a grammar of contexts $\pi$ is a hole or it does not contain a hole; (2) that reduction contexts are given by *deterministic grammars of contexts*, i.e., for each context $C$ and context non-terminal $k$, there must exist at most one nonterminal $k'$ such that we can generate $C\,[k']$ from $k$ (generate in the

[1] We are not aware of a proper formalization of grammars of contexts, but they coincide with grammars on words, thus they can be divided into similar classes. Nevertheless, we have never seen grammars of contexts that do not coincide with regular grammars on words.

[2] The notion of hybrid strategies has been first used by Sestoft [10] informally, and recently studied by García-Pérez and Nogueira [7] who further distinguish between strategies "hybrid in style" and "hybrid in nature". It seems that our definition generalizes the latter.

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

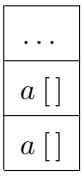


Figure 3. A malicious grammar of contexts ($E$ is the starting symbol) and a stack representation of some context with many $a$s around the hole, i.e., $\ldots aaaa[\,]$

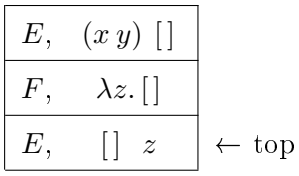


Figure 4. The representation of the context $(x\ y)\ (\lambda z.\,[\,]_F\ z)$ from the grammar in Fig. 1 (the uppermost $E$ is the starting symbol and the last two elements: $F$, $[\,]_F$ are omitted)

sense of generating from a grammar).

This approach allows to check if we can reduce in a given context, or if we can extend it with another context frame (e.g., the context in Figure 4 cannot be extended with $\lambda z.\,[\,]$) in constant time. Note that storing only the last non-terminal in a derivation is not sufficient, as after popping a frame we cannot determine this symbol for the resulting context without scanning it (cf. Figure 3 with a context that contains two $b$ symbols).

### *3.2 Generalized reduction semantics*

In the generalized case a reduction semantics is defined similarly to the standard one, but instead of a single set of reduction contexts and a single contraction function we have a family $\{\mathcal{C}_k\}_{k\in K}$ of sets of reduction contexts and a family $\{\rightharpoonup_k\}_{k\in K}$ of contractions. Let us call the elements of the set $K$ *context kinds*. In this setting we can perform a reduction $C[t_1] \to C[t_2]$ iff $t_1 \rightharpoonup_k t_2$ and $C \in \mathcal{C}_k$ for some kind $k$. This approach allows us to define different contractions for different reduction contexts.

To define a family $\{\mathcal{C}_k\}_{k\in K}$ we can give a grammar of contexts independently for each $k$, but this is not appropriate for our refocusing procedure. An alternative is to give one grammar for the family with holes indexed by $K$, so that a context is a member of $\mathcal{C}_k$ if it is generated by the grammar and if it contains the hole $[\,]_k$.

### *3.3 Generalized refocusing*

Now we briefly describe the preconditions needed by our refocusing procedure.

As input to our refocusing procedure we require a generalized reduction semantics where $\{\mathcal{C}_k\}_{k\in K}$ is given by a *normalized grammar of reduction contexts*, i.e., a grammar where (1) $K$ is the set of all context non-terminals, (2) for each $k \in K$ there is a production $k \to [\,]_k$, and (3) it satisfies the two conditions from Section 3.1. (In Figure 1 we introduce holes with indexes. These indexes are introduced to satisfy the above conditions and make our procedure applicable.) We believe that generalized reduction semantics with ordinary (regular) grammars of contexts can be always normalized to such form, however, we have not proved it formally.

For each kind $k \in K$, we also need to specify a *set of values* $\mathcal{V}_k$ denoting terms that are irreducible in $[\,]_k$ and represent proper results of computation; and a set of potential redexes $\mathcal{R}_k$ that can occur in $[\,]_k$. The latter set includes both proper redexes that can be contracted by $\rightharpoonup_k$ as well as *stuck terms* (i.e., $k$-irreducible non-value terms). A potential redex $r \in \mathcal{R}_k$ can be decomposed only to values, i.e., if $r = C[t]$ and $C$ is a non-empty reduction context w.r.t. $k$-substrategy with

a hole $[\,]_l$, then $t \in \mathcal{V}_l$. We set this requirement because our refocusing procedure needs that there is at most one redex in a term that can be reduced.

Another requirement is that the contractions in the semantics must be at least partial computable functions from potential redexes to terms. However, to generate an abstract machine satisfying the definition from Section 2 these functions must be small, otherwise the machine will operate in big steps.

To apply our refocusing procedure we also need to give a well-founded linear order in which the generated machine ought to search for a redex in a term. This is done by setting a family of orders on elementary contexts indexed by kinds and terms such that elementary contexts are comparable, i.e., $ec_1 \leq_{k,t} ec_2$ or $ec_1 \geq_{k,t} ec_2$ iff $k \to ec_1[k_1]$ and $k \to ec_2[k_2]$ are productions in the grammar of contexts, and $ec_1$, $ec_2$ are prefixes of $t$, i.e., $t = ec_1[t_1] = ec_2[t_2]$. The order determines that if $ec_1 <_{k,t} ec_2$ and $t = ec_1[t_1] = ec_2[t_2]$, then searching $t$ w.r.t. the $k$-substrategy should check the subterm $t_2$ before going to $t_1$. The introduced orders are called a *DFS evaluation strategy* in our refocusing as they describe a depth-first search order of evaluation that turns back from a term only if it has checked all possible subterms and it has found that the term is a value. This is captured by another condition, i.e., if $ec_1 <_{k,t} ec_2$, then $t = ec_2[t_2]$ and $t_2 \in \mathcal{V}_k$.

The operations of taking the greatest element and the predecessor for each of these orders have to be computable partial functions. We also need a computable operation that determines if a term is a potential redex or a value after scanning all subterms. If we want to generate a machine that operates in small steps all these functions need to be small. In our refocusing all these operations are given at once in terms of elementary decompositions, which will be described in the example below.

As an example of an input for the refocusing, consider the normal-order strategy in the lambda calculus. The reduction semantics is defined as follows: the grammar of contexts is defined in Figure 1, the contraction function is defined as $(\lambda x.t_1)\, t_2 \rightharpoonup_k t_1[t_2/x]$ for $k = E, F$, the values $\mathcal{V}_E$ are $\beta$-normal forms and $\mathcal{V}_F$ are neutral terms (cf. Figure 1) plus all lambda abstractions. The elementary decompositions are given by functions $\Downarrow$ and $\Uparrow$ (in the relational form):

$$
\begin{array}{rlrl}
[t_1\, t_2]_k & \Downarrow [t_1]_F\ t_2 & [[a]\, t]_k & \Uparrow a\, [t]_E \\
[x]_k & \Downarrow \mathit{Value}_k(x) & [a\, [v]]_k & \Uparrow \mathit{Value}_k(a\, v) \\
[\lambda x.t]_E & \Downarrow \lambda x.\, [t]_E & [\lambda x.[v]]_E & \Uparrow \mathit{Value}_E(\lambda x.v) \\
[\lambda x.t]_F & \Downarrow \mathit{Value}_F(\lambda x.t) & [[\lambda x.t_1]\, t_2]_k & \Uparrow \mathit{Redex}_k((\lambda x.t_1)\, t_2)
\end{array}
$$

Each of the functions either returns a new decomposition, or recognizes the processed term as a redex or as a value of the appropriate kind. The function $[t]_k \Downarrow$ defines how to proceed when a term $t$ is considered in a context with a hole $[\,]_k$ for the first time. If it returns $ec[t']_l$, then $ec$ is the greatest element in $\leq_{k,t}$, otherwise there is no further decomposition and it is determined if $t$ is a value or a potential redex. The $l$ symbol is the kind of the hole of the current context extended by $ec$. For example, $[t_1\, t_2]_k \Downarrow [t_1]_F\ t_2$ prescribes that when we encounter an application $t_1\, t_2$, we have to decompose $t_1$ according to the $F$-strategy with the current contexts extended by $[\,]\, t_2$. Then $[ec[v]]_k \Uparrow$ defines how to proceed when a term $ec[v]$ is reconsidered in a context with a hole $[\,]_k$ after finding out that $v$ is a value of an appropriate

kind (this kind is determined by $k$ and $ec$, because the grammars of contexts are deterministic). If $\Uparrow$ returns $ec'[t]_l$, then $ec'$ is a predecessor of $ec$ w.r.t. $\leq_{k,ec[v]}$.

The machine obtained by the generalized refocusing has the following form. Each configuration is a tuple $\langle t, \mathtt{C}, k\rangle_e$ or $\langle \mathtt{C}, v\rangle_a$ (historically $e$ stands for "eval" and $a$ for "apply"), where $t$ is a term, $k$ is the kind of the hole in the current context, $\mathtt{C}$ is the stack representing the rest of the context (cf. Figure 4), and $v$ is a value from $\mathcal{V}_k$. The symbol $k$ has no computational meaning in $\langle\,\rangle_a$ configurations, thus it is omitted, yet it can be inferred from $\mathtt{C}$ because grammars of contexts are deterministic. The initial configurations are of the form $\langle t, \varepsilon, k_{\text{init}}\rangle_e$ and the final configurations are of the form $\langle \varepsilon, v\rangle_a$, where $k_{\text{init}}$ is the starting symbol in the grammar of contexts. The transitions are as follows:

$$
\begin{array}{ll}
\langle t, \mathtt{C}, k\rangle_e \to \langle \mathtt{C}, t\rangle_a & \text{if } [t]_k \Downarrow Value_k(t),\\
\langle t, \mathtt{C}, k\rangle_e \to \langle t', \mathtt{C}, k\rangle_e & \text{if } [t]_k \Downarrow Redex_k(t) \wedge t \rightharpoonup_k t',\\
\langle t, \mathtt{C}, k\rangle_e \to \langle t', (k, ec) :: \mathtt{C}, k'\rangle_e & \text{if } [t]_k \Downarrow ec\,[t']_{k'},\\
\langle (k, ec) :: \mathtt{C}, v\rangle_a \to \langle \mathtt{C}, ec[v]\rangle_a & \text{if } [ec[v]]_k \Uparrow Value_k(ec[v]),\\
\langle (k, ec) :: \mathtt{C}, v\rangle_a \to \langle t, k, \mathtt{C}\rangle_e & \text{if } [ec[v]]_k \Uparrow Redex_k(ec[v]) \wedge ec[v] \rightharpoonup_k t,\\
\langle (k, ec) :: \mathtt{C}, v\rangle_a \to \langle t, (k, ec') :: \mathtt{C}, k'\rangle_e & \text{if } [ec[v]]_k \Uparrow ec'\,[t]_{k'}\,.
\end{array}
$$

Note that the machine depends only on $\Downarrow$ and $\Uparrow$, but the introduced requirements are needed to guarantee that such a machine realizes the input reduction semantics.

The transitions of the machine generated for the normal-order reduction semantics described above are as follows:

$$
\begin{array}{ll}
\langle x, \mathtt{C}, k\rangle_e \to \langle \mathtt{C}, x\rangle_a & \langle (k, [\,]\ t_2) :: \mathtt{C}, \lambda x.t_1\rangle_a \to \langle t_1[t_2/x], \mathtt{C}, k\rangle_e\\
\langle t_1\ t_2, \mathtt{C}, k\rangle_e \to \langle t_2, (k, [\,]\ t_2) :: \mathtt{C}, F\rangle_e & \langle (k, [\,]\ t) :: \mathtt{C}, a\rangle_a \to \langle t, (k, a\ [\,]) :: \mathtt{C}, E\rangle_e\\
\langle \lambda x.t, \mathtt{C}, E\rangle_e \to \langle t, (E, \lambda x.[\,]) :: \mathtt{C}, E\rangle_e & \langle (k, a\ [\,]) :: \mathtt{C}, v\rangle_a \to \langle \mathtt{C}, a\ v\rangle_a\\
\langle \lambda x.t, \mathtt{C}, F\rangle_e \to \langle \mathtt{C}, \lambda x.t\rangle_a & \langle (E, \lambda x.[\,]) :: \mathtt{C}, v\rangle_a \to \langle \mathtt{C}, \lambda x.v\rangle_a
\end{array}
$$

where $k \in \{E, F\}$. This example is shown in the file `refocusing_examples/lam_no.v`.

## 4 Correctness of the generated machine

The refocusing procedure generates an abstract machine that provides not only extensionally equivalent semantics, but one that can be proved to exactly implement the reduction semantics given as input, possibly in smaller steps.

This idea is captured in the following definition of *tracing*. In the implementation each generated machine is proved to trace the input reduction semantics.

**Definition 4.1** *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*

(i) *if $t_1 \to t_2$, then $[\![t_1]\!] = [\![t_2]\!]$ or $[\![t_1]\!] \Rightarrow [\![t_2]\!]$,*

(ii) *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*

(iii) *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 [9]. Our definition is more general in that it is adequate for non-deterministic systems.

## 5 Implementation

Our work is implemented as a library in Coq 8.5. A comprehensive instruction on how to use the library is provided in the repository in the `instruction` folder. A brief instruction is provided in the section *Quick start* of the instruction file.

One significant note is that grammars of contexts are implemented in the form of deterministic total automatons on elementary contexts. This enforces introducing sink (dead) non-terminals. An alternative that we consider is to make the type of elementary contexts dependent on two kinds, where the first describes the left-hand side of a production, and the second when plugged in the elementary contexts forms the right-hand side. This should remove the need for sinks.

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

## 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 and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

[3] Olivier Danvy. Defunctionalized interpreters for programming languages. *SIGPLAN Notices*, 43(9):131–142, September 2008.

[4] Olivier Danvy and Jacob Johannsen. From outermost reduction semantics to abstract machine. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 91–108, Madrid, Spain, September 2013. Springer.

[5] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Dept. of Computer Science, Aarhus University, 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.

[6] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Benjamin C. Pierce, editor, *Principles of Programming Languages (POPL'09)*, pages 153–164. ACM Press, January 2009.

[7] A. García-Pérez and P. Nogueira. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Science of Computer Programming*, 95, Part 2:176 – 199, 2014.

[8] 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 Press.

[9] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[10] 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.

[11] 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. Springer-Verlag, September 2010.