Automating Derivations of Abstract Machines from Reduction Semantics:
A Generic Formalization of Refocusing in Coq

Filip Sieczkowski*, Małgorzata Biernacka, and Dariusz Biernacki

Institute of Computer Science, University of Wrocław

Abstract. We present a generic formalization of the refocusing transformation for functional languages in the Coq proof assistant. The refocusing technique, due to Danvy and Nielsen, allows for mechanical transformation of an evaluator implementing a reduction semantics into an equivalent abstract machine via a succession of simple program transformations. So far, refocusing has been used only as an informal procedure: the conditions required of a reduction semantics have not been formally captured, and the transformation has not been formally proved correct. The aim of this work is to formalize and prove correct the refocusing technique. To this end, we first propose an axiomatization of reduction semantics that is sufficient to automatically apply the refocusing method. Next, we prove that any reduction semantics conforming to this axiomatization can be automatically transformed into an abstract machine equivalent to it. The article is accompanied by a Coq development that contains the formalization of the refocusing method and a number of case studies that serve both as an illustration of the method and as a sanity check on the axiomatization.

1 Introduction

Refocusing has been introduced by Danvy and Nielsen [12] as a method for optimizing functions that directly implement the transitive closure of the following three steps: (1) decomposition of a term in order to locate a reduction site, (2) contraction of a redex, (3) recomposition of the entire term. Such an implementation induces an overhead due to the recomposition of a term that will immediately be decomposed in the next iteration; in such cases, refocusing can be applied to eliminate the overhead and produce more efficient functions.

In particular, Danvy and Nielsen showed how to mechanically derive an abstract machine from an evaluator implementing a reduction semantics (i.e., a small-step operational semantics with explicit representation of reduction contexts).

The original derivation method was applied to substitution-based reduction semantics and accounted for local contractions. It has later been used by Biernacka and Danvy to derive abstract machines for context-sensitive reduction

* Author's current affiliation: IT University of Copenhagen

semantics [3], and it has been extended to a syntactic correspondence in order to facilitate derivations of environment-based machines from reduction semantics using explicit substitutions [2]. The refocusing method has been applied since for a variety of languages [4,7,8,9,11,13,14,16]. This transformation can serve not only to derive new abstract machines, but also as a tool for interderiving different semantic specifications of the same language that are often designed independently from each other. For example, Danvy and Biernacka have shown the underlying reduction semantics of several well-known abstract machines and confirmed their correctness by applying refocusing [2].

The goal of this work is to formalize the refocusing transformation and prove it correct in the Coq proof assistant. In the article introducing the vanilla version of refocusing, Danvy and Nielsen define a set of conditions on a reduction semantics sufficient for constructing a refocused evaluation function, and they sketch a correctness proof of this function. However, they focus on the final efficient definition of an evaluation function and their representation of reduction semantics is not adequate for a formalization on a computer. In contrast, we formalize refocusing as a succession of simple intensional transformations of the evaluation relation induced by reduction semantics and we formally prove the correctness of all steps of the transformation.

To this end, we first give an axiomatization of reduction semantics that is sufficient to automatically apply the refocusing method. Next, we prove that any reduction semantics conforming to this axiomatization can be automatically transformed into an abstract machine equivalent to it. We formalize each intermediate step of the derivation and we state and prove its correctness. Our work is based on preliminary results by Biernacka and Biernacki [1] which we extend to a general framework.

Apart from the formalization of the basic refocusing transformation of Danvy and Nielsen, we also consider its variant used by Biernacka and Danvy for context-sensitive reduction semantics (useful, e.g., for expressing languages with control operators such as `call/cc`), and also a syntactic correspondence that for variants of calculi of closures leads to abstract machines with environments (rather than with meta-level substitutions) [2,3].

The formalization is carried out in the Coq proof assistant.[1] It includes a number of case studies: the language of arithmetic expressions, the lambda calculi (both pure and with `call/cc`) under call-by-value (accounting for Felleisen's CEK machine [15]) and call-by-name (accounting for Krivine's abstract machine [17]), as well as Mini-ML. These case studies serve both as an illustration of how to use the formalization in practice and as a sanity check on the axiomatization. However, due to space constraints, these examples are left out of this article—the reader is welcome to consult the Coq development.

[1] The Coq development accompanying this article can be found at `<http://fsieczkowski.com>`. Sections 2 and 3 refer to the subdirectory `substitutions`, Section 4—to the subdirectory `environments`, and Section 5—to the subdirectory `substitutions_cs` of the Coq development.

The implementation makes essential use of the Coq module system [5] that is based on the module system known from the ML family of languages. In Coq, however, a module type (i.e., a signature in SML parlance) may contain not only data declarations, but also logical axioms that capture extra properties of the data. In consequence, in an implementation of any module of such a signature one must provide proofs of the required properties. In our formalization, we first gather all the properties characterizing a reduction semantics into a module type, and similarly we define further signatures describing each of the intermediate semantic artefacts of the refocusing transformation. Then we define a series of functors each implementing one step of the derivation (i.e., the transformation from one module to the next). The formalization is engineered as a generic framework, so one can use it to transform one's own reduction semantics into an abstract machine. To this end, one has to specify the reduction semantics in the format prescribed by the signature, and then to apply the sequence of functors in order to obtain the abstract machine that is extensionally equivalent to the initial semantics.

The rest of this article is structured as follows. In Section 2 we define an axiomatization for the substitution-based reduction semantics amenable to refocusing. In Section 3 we give a brief summary of the refocusing method starting with the semantics given in Section 2 and we show the resulting abstract machine semantics in two versions: an *eval/continue* abstract machine and an *eval* abstract machine.[2] In Section 4 we formalize the extension of refocusing for a language with closures. In Section 5 we turn to context-sensitive reduction semantics and sketch the formalization for this extension. We conclude in Section 6.

2 An axiomatization of a substitution-based reduction semantics

In this section we describe an axiomatization of a generic reduction semantics that defines sufficient conditions for the semantics to be automatically refocused. The description is similar to the one given by Danvy and Nielsen in [12], though it differs in several points, e.g., we require potential redexes to be explicitly provided, whereas Danvy and Nielsen specify them by their properties. The differences we introduce arise from the need to completely formalize the language and its reduction semantics in a proof assistant. We use the call-by-value lambda calculus (λ_v) as a running example in this section and in the next.

[2] We distinguish between eval/continue abstract machines, e.g., the CK abstract machine [15], that make transitions between two kinds of configurations: one focused on the term under evaluation and one focused on the context of the evaluation, and eval abstract machines, e.g., the Krivine Abstract Machine [17], that operate on configurations of one kind.

2.1 Syntactic categories

We begin by specifying the syntactic categories used throughout the formalization: terms, values, potential redexes, and context frames, which we denote with t, v, r and f, respectively. All these sets are declared as parameters of the language signature and have to be provided by the user in the implementation of that signature. Both the set of values and the set of potential redexes should be disjoint subsets of the set of terms. As traditional, values are terms irreducible in a given strategy (i.e., results of evaluation), and potential redexes can be thought of as minimal non-value terms.

Further, we introduce the syntactic category of reduction contexts (denoted as E). A reduction context is defined as a list of context frames and is interpreted similarly to the standard inside-out reduction context (i.e., as a stack). The composition of two reduction contexts is denoted as $E_1 \circ E_2$, while a context extended with a single frame is denoted as $f :: E$. The meaning of reduction contexts is usually specified by a *plug* function[3], which describes the effect of plugging a term in the context. Since reduction contexts are defined constructs in our approach, we can specify *plug* as a (left) folding of an *atomic plug* function over a reduction context, where *atomic plug* describes the effect of plugging a term into a context frame and has to be provided by the user. We denote with $E[t]$ the term obtained by plugging a term t into a context E. Our approach enforces that the composition of contexts and *plug* satisfy the property

$$\text{(plug-compose)} \quad (E_1 \circ E_2)[t] = E_2[E_1[t]],$$

which otherwise would have to be proved. In the example language λ_v, the syntactic categories can be defined with the following grammars, where x ranges over the (unspecified) set of variables:

$$
\begin{aligned}
t &::= x \mid \lambda x.t \mid t\,t && \text{(terms)}\\
v &::= x \mid \lambda x.t && \text{(values)}\\
r &::= v\,v && \text{(potential redexes)}\\
f &::= [\,]\,t \mid v\,[\,] && \text{(context frames)}
\end{aligned}
$$

The grammar of potential redexes includes both standard beta redexes ("actual redexes") and stuck terms, e.g., $x\,v$.

2.2 Decompositions and contraction

The notion of decomposition is defined as in Danvy and Nielsen [12]: any pair (E, t) is a decomposition of the term $E[t]$. A decomposition of the form (E, v) is said to be *trivial*, while the decomposition $([\,], t)$ is called *empty*. Our axiomatization requires that for both values and potential redexes every nonempty

[3] In some recent articles [11,13,14] this function is called *recompose*.

decomposition is trivial and that every term that has only trivial or empty decompositions is either a value or a potential redex.[4] We also define a partial function *contract* that takes a potential redex as argument and returns the term resulting from reducing it. For stuck terms, the function *contract* is undefined. All the definitions and properties introduced so far are specified in the module type `RED_LANG`.

Let us now look at the λ_v-calculus. It is easy to see that a value can never be decomposed into a term and a nonempty context. It follows that potential redexes can only be decomposed trivially or into the empty context and the redex itself: any redex $r = v\,v'$ can be decomposed either as $([\,], r)$, $([\,]\,v', v)$, or as $(v\,[\,], v')$, so the obligations on decompositions of values and redexes are fulfilled. The requirement that a term with only trivial or empty decompositions is either a value or a redex is also easy to prove by case analysis on the term.

Contraction in our example is defined using the standard capture-avoiding substitution: $(\lambda x.t)\,v$ reduces to $t[x/v]$. Both the semantics before and after the refocusing transformation are defined using contraction as a black box, and for the basic transformation we do not require any specific properties of this function.

2.3 Reduction semantics

For a language satisfying the conditions stated above we can now specify a reduction semantics. First, we notice that a decomposition of any term t can lead to one of three possibilities:

1. t is a redex r that cannot be further decomposed
2. t is a value v that cannot be decomposed (e.g., a lambda form or a lazy constructor)
3. t can be decomposed into a term t' and a context frame f

The first and the third case are straightforward—either we have found a decomposition by locating the next potential redex, or we need to further decompose t' in the context extended with f. In the second case, we have to look at the current context: if it is empty, we have finished decomposing and reached a value. Otherwise, we have to examine the innermost context frame f in the surrounding context, and the value v. Together they can either form a potential redex or a value (in the case when all the decompositions of $f[v]$ are trivial, e.g., when $f[v]$ is a pair constructed of two values), or they can be decomposed into a new term t' and a new context frame f'. We require the user to provide two functions that capture this insight: $\texttt{dec}_\texttt{t}$, that for a given term describes how it can be decomposed in one step, and an analogous function $\texttt{dec}_\texttt{f}$, that does the same but for a pair of a context frame and a value. These "atomic" decomposition functions let us build a generic decomposition predicate. We require the

[4] It is tempting to use a stronger requirement for values: a value can only have the empty decomposition. However, such a condition would preclude, i.e., values of the form $\texttt{S}\,v$ (representing natural numbers), where v is a value.

user to provide not only the definitions of these two functions $\mathtt{dec_t}$ and $\mathtt{dec_f}$, but also a proof of their correctness with respect to the (atomic) plug function, i.e., that these functions are inverses of the atomic plug function. The decomposition relation can now be defined by iterating these user-defined functions until a decomposition is found. Formally, it is defined as an indexed family of inductive predicates in Coq, and its transcribed definition is presented in the top part of Figure 1.

In the case of the λ_v-calculus, we can define the atomic decomposition functions as follows:

$$\mathtt{dec_t}\, v = v \qquad \mathtt{dec_f}\, ([\,]\, t)\, v = (t, v\, [\,])$$
$$\mathtt{dec_t}\, (t_1\, t_2) = (t_1, [\,]\, t_2) \qquad \mathtt{dec_f}\, (v\, [\,])\, v' = v\, v'$$

Next, we introduce two strict, well-founded orderings: a subterm order $\prec_{\mathtt{t}}$ that characterizes the relation of being a subterm of another term (and is defined using the atomic plug function), and an order on context frames $\prec_{\mathtt{f}}$ that describes the order of evaluation of subterms (intuitively, $f \prec_{\mathtt{f}} f'$ means that f' has more subterms left to be visited than f). The latter order is left to be provided by the user, together with a proof that this order is compatible with what the atomic decomposition functions describe. The relation might be definable in a similar manner to $\prec_{\mathtt{t}}$, however, it seems that the required proofs are easier when the relation is defined by the user. Specifically, we require the following properties to hold:

- if $\mathtt{dec_t}\, t = (t', f)$, then f is maximal with respect to $\prec_{\mathtt{f}}$
- if $\mathtt{dec_f}\, f\, v = (t, f')$, then $f' \prec_{\mathtt{f}} f$ and $\forall f''. f'' \prec_{\mathtt{f}} f \implies f'' \preceq_{\mathtt{f}} f'$
- if $\mathtt{dec_f}\, f\, v$ returns a redex or a value, then f is minimal with respect to $\prec_{\mathtt{f}}$
- if $\mathtt{dec_t}\, t \neq (t', f)$ for all t', f, then t has only the empty decomposition

Additionally, we require that all the elementary decompositions of a given term are comparable, i.e., if $f[t] = f'[t']$, then $f \prec_{\mathtt{f}} f' \vee f' \prec_{\mathtt{f}} f \vee (f = f' \wedge t = t')$, and that if $f[t] = f'[t'] \wedge f \prec_{\mathtt{f}} f'$, then t' is a value, which effectively fixes the order of evaluation. Of course, both the structure of terms and the order of evaluation exist without specifying these orders: their explicit definition, however, allows us to conduct inductive reasoning without knowing the precise structure of terms. We need this kind of reasoning to prove the necessary properties of the semantics, such as the unique-decomposition lemma.

In our example of the λ_v-calculus, the order on context frames can be defined as the smallest strict order with the property $v\, [\,] \prec_{\mathtt{f}} [\,]\, t$ for any term t and value v. This relation should hold for any t and v, because the term $(v\, t)$ can be decomposed into both contexts. It is also easy to see that the orders satisfy all the required conditions.

The properties of orders stated above are similar to those specified in Danvy and Nielsen [12], but they are in general more lax for the purpose of a formalization in a proof assistant. For example, unlike Danvy and Nielsen, we do not impose a fixed evaluation order but we leave it to the user to specify it.

$$\mathtt{dec}\, t\, E\, d \iff \begin{cases} d = (r, E) & \text{if } \mathtt{dec_t}\, t = r \\ \mathtt{dec_{ctx}}\, E\, v\, d & \text{if } \mathtt{dec_t}\, t = v \\ \mathtt{dec}\, t'\, (f :: E)\, d & \text{if } \mathtt{dec_t}\, t = (t', f) \end{cases}$$

$$\mathtt{dec_{ctx}}\, [\,]\, v\, d \iff d = v$$

$$\mathtt{dec_{ctx}}\, (f :: E)\, v\, d \iff \begin{cases} d = (r, E) & \text{if } \mathtt{dec_f}\, f\, v = r \\ \mathtt{dec_{ctx}}\, E\, v'\, d & \text{if } \mathtt{dec_f}\, f\, v = v' \\ \mathtt{dec}\, t\, (f' :: E)\, d & \text{if } \mathtt{dec_f}\, f\, v = (t, f') \end{cases}$$

$$\mathtt{iter}\, v\, v' \iff v = v'$$

$$\mathtt{iter}\, (r, E)\, v \iff r \mapsto t \wedge \boxed{\mathtt{dec}\, (E[t])\, [\,]\, d} \wedge \mathtt{iter}\, d\, v \quad \text{for some } d$$

$$\mathtt{eval}\, t\, v \iff \mathtt{dec}\, t\, [\,]\, d \wedge \mathtt{iter}\, d\, v \quad \text{for some } d$$

Fig. 1. A generic evaluator for reduction semantics

The module type `RED_REF_LANG` contains all the requirements on the language stated above in the form of parameter declarations to be provided by the implementation, definitions (including inductive definitions), and axioms expressing required properties to be proved in the implementation. The definition of the reduction semantics is then given by the module types `RED_SEM` and `RED_REF_SEM` parameterized by `RED_LANG`.

3 From reduction semantics to abstract machine by refocusing

In this section, we present the formalization of the refocusing transformation for a language conforming to the axiomatization of reduction semantics specified in Section 2.3. The transformation follows the steps of the original refocusing method as presented by Danvy and Nielsen [12] and it consists of a series of semantics, each obtained in a systematic way from the preceding one and provably equivalent to it. Each of the semantics is given by an inductively defined relation that in the Coq formalization can be obtained by instantiating a functor with the module implementing the language under consideration.

3.1 An evaluator

The starting point of the refocusing transformation is the evaluation function obtained by naively iterating the reduce-plug-decompose procedure that uses the components introduced in the previous section. This semantics is shown in Figure 1, where successful contraction of a potential redex is denoted with $\mapsto$, and the $\mathtt{dec_t}$ and $\mathtt{dec_f}$ functions are the elementary decomposition functions from Section 2.3. Both the iterating function `iter` and the evaluation function `eval` are represented as inductively defined relations (see module type `RED_SEM`).

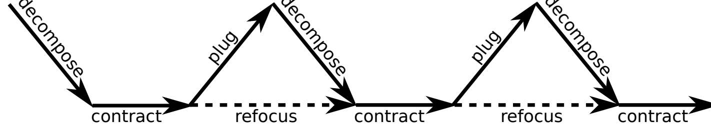

Fig. 2. A schematic view of the refocusing transformation

$$\begin{aligned}
\texttt{iter}_{\texttt{ssam}}\, v\, v' &\iff v = v' \\
\texttt{iter}_{\texttt{ssam}}\, (r, E)\, v &\iff r \mapsto t \wedge \boxed{\texttt{dec}\, t\, E\, d} \wedge \texttt{iter}_{\texttt{ssam}}\, d\, v \quad \text{for some } d \\
\texttt{eval}_{\texttt{ssam}}\, t\, v &\iff \texttt{dec}\, t\, [\,]\, d \wedge \texttt{iter}_{\texttt{ssam}}\, d\, v \quad \text{for some } d
\end{aligned}$$

Fig. 3. A generic small-step abstract machine for reduction semantics

3.2 A small-step abstract machine

The first step of the transformation, where the actual *refocusing* happens, builds on the observation that in a deterministic reduction semantics the following property should hold

$$\texttt{dec}\, (E[t])\, [\,]\, d \iff \texttt{dec}\, t\, E\, d.$$

Indeed, this property is a special case of (plug-compose), which we have asserted in Section 2. Thus, we can substitute the right-hand side of this equivalence for the left-hand side in the definition of the `iter` function in the naive evaluator (the place where this change takes place is indicated with a gray box in Figure 1). The resulting definition is more efficient in that it avoids reconstructing the entire term after a contraction and then decomposing it again. The situation is shown in Figure 2, where "refocus" denotes the more efficient procedure that bypasses the successive *plug* and *decompose* steps, and instead it continues to decompose with the term obtained after contraction in the current context. Hence we arrive at a more efficient evaluator, a small-step abstract machine (a.k.a a pre-abstract machine) [3,12]. The definition of this machine is presented in Figure 3, where the only change compared to Figure 1 is in the contents of the gray box (the `dec` relation is defined as before and omitted).

The correctness of this step of transformation is captured by the following proposition.

Proposition 1. *For any term t and any value v of a language satisfying the axioms of Section 2, the equivalence* $\texttt{eval}\, t\, v \iff \texttt{eval}_{\texttt{ssam}}\, t\, v$ *holds.*

The proof follows immediately from a similar equivalence defined for `iter` and $\texttt{iter}_{\texttt{ssam}}$, which in turn is done by induction on the derivation and uses the (plug-compose) property.

$$
\begin{aligned}
\texttt{dec}_{\texttt{sam}}\, t\, E\, v &\iff \begin{cases} \texttt{iter}_{\texttt{sam}}\,(r, E)\, v & \text{if } \texttt{dec}_{\texttt{t}}\, t = r \\ \texttt{dec}_{\texttt{ctx-sam}}\, E\, v'\, v & \text{if } \texttt{dec}_{\texttt{t}}\, t = v' \\ \texttt{dec}_{\texttt{sam}}\, t'\,(f :: E)\, v & \text{if } \texttt{dec}_{\texttt{t}}\, t = (t', f) \end{cases} \\
\texttt{dec}_{\texttt{ctx-sam}}\, [\,]\, v\, v' &\iff \texttt{iter}_{\texttt{sam}}\, v\, v' \\
\texttt{dec}_{\texttt{ctx-sam}}\, (f :: E)\, v\, v' &\iff \begin{cases} \texttt{iter}_{\texttt{sam}}\,(r, E)\, v' & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = r \\ \texttt{dec}_{\texttt{ctx-sam}}\, E\, v''\, v' & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = v'' \\ \texttt{dec}_{\texttt{sam}}\, t\,(f' :: E)\, v' & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = (t, f') \end{cases} \\
\texttt{iter}_{\texttt{sam}}\, v\, v' &\iff v = v' \\
\texttt{iter}_{\texttt{sam}}\,(r, E)\, v &\iff r \mapsto t \wedge \texttt{dec}_{\texttt{sam}}\, t\, E\, v \\
\texttt{eval}_{\texttt{sam}}\, t\, v &\iff \texttt{dec}_{\texttt{sam}}\, t\, [\,]\, v
\end{aligned}
$$

Fig. 4. A generic staged abstract machine for reduction semantics

The specification of a small-step abstract machine is captured in the module type `SS_ABSTRACT_MACHINE` in the Coq development, and it is constructed by the functor `SSAbstractMachine` given a module implementing the reduction semantics.

3.3 A staged abstract machine

The next step of the transformation consists in fusing computations, using lightweight fusion [10], so that the definitions of the relations `dec` and $\texttt{dec}_{\texttt{ctx}}$ are now made recursively dependent on the `iter` relation. The definition of the generic staged abstract machine is shown in Figure 4.

The correctness of this step of derivation is captured by the following proposition:

Proposition 2. *For any term t and any value v of a language satisfying the axioms of Section 3, the equivalence* $\texttt{eval}_{\texttt{ssam}}\, t\, v \iff \texttt{eval}_{\texttt{sam}}\, t\, v$ *holds.*

The proof is a little more complicated than in the case of the small-step abstract machine. To prove the "only if" case we need the following lemma:

Lemma 1. *For any term t, context E, decomposition d, and value v of a language satisfying the axioms of Section 2, the following implications hold:*

$$
\begin{aligned}
\texttt{dec}\, t\, E\, d \wedge \texttt{iter}_{\texttt{sam}}\, d\, v &\implies \texttt{dec}_{\texttt{sam}}\, t\, E\, v, \\
\texttt{iter}_{\texttt{ssam}}\, d\, v &\implies \texttt{iter}_{\texttt{sam}}\, d\, v.
\end{aligned}
$$

The lemma is proved by induction on the derivations of $\texttt{dec}\, t\, E\, d$ and $\texttt{iter}_{\texttt{ssam}}\, d\, v$, respectively. For the "if" case, we need the property stating that `dec` is a total function, which follows from the axiomatization, and a similar lemma:

$$
\begin{array}{rcl}
\langle t \rangle_{\mathtt{init}} & \rhd & \langle t, [\,] \rangle_{\mathtt{e}} \\
\langle t, E \rangle_{\mathtt{e}} & \rhd & \begin{cases} \langle t', E \rangle_{\mathtt{e}} & \text{if } \mathtt{dec}_{\mathtt{t}}\, t = r \text{ and } r \mapsto t' \\ \langle E, v \rangle_{\mathtt{c}} & \text{if } \mathtt{dec}_{\mathtt{t}}\, t = v \\ \langle t', f :: E \rangle_{\mathtt{e}} & \text{if } \mathtt{dec}_{\mathtt{t}}\, t = (t', f) \end{cases} \\
\langle [\,], v \rangle_{\mathtt{c}} & \rhd & \langle v \rangle_{\mathtt{fin}} \\
\langle f :: E, v \rangle_{\mathtt{c}} & \rhd & \begin{cases} \langle t, E \rangle_{\mathtt{e}} & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = r \text{ and } r \mapsto t' \\ \langle E, v' \rangle_{\mathtt{c}} & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = v' \\ \langle t, f' :: E \rangle_{\mathtt{e}} & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = (t, f') \end{cases}
\end{array}
$$

$$
\mathtt{eval}_{\mathtt{ecam}}\, t\, v \iff \langle t \rangle_{\mathtt{init}} \rhd^{+} \langle v \rangle_{\mathtt{fin}}
$$

Fig. 5. A generic eval/continue abstract machine semantics derived from reduction semantics

Lemma 2. *For any term t, context E, decomposition d and value v of a language satisfying the axioms of Section 2, the following implication holds:*

$$
\mathtt{dec}_{\mathtt{sam}}\, t\, E\, v \wedge \mathtt{dec}\, t\, E\, d \implies \mathtt{iter}_{\mathtt{ssam}}\, d\, v.
$$

The lemma is proved by induction on the derivation of $\mathtt{dec}_{\mathtt{sam}}$.

The specification of a staged abstract machine is captured in the module type `STAGED_ABSTRACT_MACHINE` in the Coq development, and it is constructed by the functor `StagedAbstractMachine` given a module implementing the reduction semantics.

3.4 The result: an eval/continue abstract machine

The final step of the transformation yields an eval/continue abstract machine by inlining the definition of $\mathtt{iter}_{\mathtt{sam}}$ in $\mathtt{dec}_{\mathtt{sam}}$ and $\mathtt{dec}_{\mathtt{ctx\text{-}sam}}$ and by introducing the relation $\rhd$ (and its transitive closure) between configurations defined by the latter two. The grammar of configurations of the machine reads as follows:

$$
c ::= \langle t \rangle_{\mathtt{init}} \mid \langle t, E \rangle_{\mathtt{e}} \mid \langle E, v \rangle_{\mathtt{c}} \mid \langle v \rangle_{\mathtt{fin}}
$$

Apart from the initial ($\langle t \rangle_{\mathtt{init}}$) and the final ($\langle v \rangle_{\mathtt{fin}}$) configurations corresponding to the "loading" and the "unloading" of the machine, there are two other kinds of configurations: an *eval*-configuration of the form $\langle t, E \rangle_{\mathtt{e}}$ and an *continue*-configuration of the form $\langle E, v \rangle_{\mathtt{c}}$. The *eval*-configurations arise from decompositions of terms in the reduction semantics, and the *continue*-configurations arise from analyzing values in context. The transitions of the machine and the induced evaluation function are presented in Figure 5.

The correctness of the overall transformation can then be stated with the following theorem, which follows from the correctness of individual transformation steps.

Theorem 1. *For any term t and any value v of a language satisfying the axioms of Section 2, the equivalence* $\mathtt{eval}\, t\, v \iff \mathtt{eval_{ecam}}\, t\, v$ *holds.*

The specification of an eval/continue machine is captured in the module type `EVAL_CONTINUE_MACHINE` in the Coq development, and it is constructed by the functor `EvalContinueMachine` given a module implementing the reduction semantics.

Redundancies in the generic abstract machine. Due to the transformation working in a very general setting, the resulting abstract machine may contain transitions that are not actually possible, e.g., in the case of the λ_v-calculus, the transition from $\langle f :: E, v\rangle_{\mathtt{c}}$ to $\langle E, v'\rangle_{\mathtt{c}}$ is present in the derived machine, but it is never made because the side condition can never arise. It is however possible to simplify the machine by replacing the $\mathtt{dec_t}$ and $\mathtt{dec_f}$ functions with their definitions, compressing corridor transitions, and then removing unreachable transitions. In the case of the λ_v-calculus the obtained abstract machine coincides with Felleisen's CK machine [15].

3.5 An eval abstract machine

In some cases we can obtain an eval abstract machine from an eval/continue abstract machine. It is possible when the reduction semantics satisfies an extra property that amounts to the condition that values have only empty decompositions. When this condition is fulfilled, it is possible to eliminate the *continue*-configurations since then the machine never makes a single transition from one *continue*-configuration to another *continue*-configuration. This step of the transformation has also been shown in Danvy and Nielsen [12], but the authors do not specify conditions on the reduction semantics under which it can be performed.

In the case of an eval abstract machine there are no *continue*-configurations in the machine:

$$c ::= \langle t\rangle_{\mathtt{init}} \mid \langle t, E\rangle_{\mathtt{e}} \mid \langle v\rangle_{\mathtt{fin}}$$

The transitions of the machine and the induced evaluation function are presented in Figure 6.

The correctness of the overall transformation to an eval machine follows from the correctness of each of its steps, and is summarized by the following theorem:

Theorem 2. *Let L be a language satisfying the axioms of Section 2 and such that for any frame f and any value v,* $\mathtt{dec_f}\, f\, v$ *is not a value. Then for any term t and any value v of L, the equivalence* $\mathtt{eval}\, t\, v \iff \mathtt{eval_{eam}}\, t\, v$ *holds.*

The specification of an eval abstract machine is captured in the module type `EVAL_MACHINE` in the Coq development, and it is constructed by the functor `EvalMachine` given a module implementing the reduction semantics.

$$
\begin{aligned}
\langle t\rangle_{\mathtt{init}} \quad &\rhd \quad \langle t, [\,]\rangle_{\mathtt{e}} \\
\langle t, E\rangle_{\mathtt{e}} \quad &\rhd \quad \begin{cases}
\langle v\rangle_{\mathtt{fin}} & \text{if } \mathtt{dec_t}\, t = v \text{ and } E = [\,] \\
\langle t', E\rangle_{\mathtt{e}} & \text{if } \mathtt{dec_t}\, t = r \text{ and } r \mapsto t' \\
\langle t', E'\rangle_{\mathtt{e}} & \text{if } \mathtt{dec_t}\, t = v, E = f :: E', \mathtt{dec_f}\, f\, v = r, \text{ and } r \mapsto t \\
\langle t', f' :: E'\rangle_{\mathtt{e}} & \text{if } \mathtt{dec_t}\, t = v, E = f :: E', \text{ and } \mathtt{dec_f}\, f\, v = (t', f') \\
\langle t', f :: E\rangle_{\mathtt{e}} & \text{if } \mathtt{dec_t}\, t = (t', f)
\end{cases}
\end{aligned}
$$

$$
\mathtt{eval_{eam}}\, t\, v \iff \langle t\rangle_{\mathtt{init}} \rhd^{+} \langle v\rangle_{\mathtt{fin}}
$$

Fig. 6. A generic eval abstract machine semantics derived from reduction semantics

4 Refocusing in reduction semantics with explicit substitutions

In this section we sketch the formalization of a derivation method that produces environment-based abstract machines for languages with closures. A closure is a purely syntactic entity that consists of a term together with an explicit substitution. The idea that a language with closures corresponds more faithfully to abstract machines using environments than a language with substitution as a meta-level operation originates in Curien's work: he introduced the calculus of closures $\lambda\rho$ as the simplest calculus of closures accounting for environment machines for the λ-calculus [6].

The extension of refocusing that operates on languages with closures is due to Biernacka and Danvy [2]. The method uses an intermediate calculus (the $\lambda\widehat{\rho}$-calculus) that minimally extends Curien's $\lambda\rho$-calculus in order to accommodate all the necessary refocusing steps, but the final result it produces, i.e., an environment-based abstract machine, operates on terms of the smaller $\lambda\rho$-calculus. In the formalization we also use two calculi: one denoted by C (the calculus corresponding to Curien's $\lambda\rho$-calculus) and the other denoted by $\widehat{C}$, which is an extended version of C amenable to refocusing (the calculus corresponding to Biernacka and Danvy's $\lambda\hat{\rho}$-calculus). It might seem that given C one should be able to compute $\widehat{C}$, however, this is a task that requires some insight, and so we formalize both connected calculi.

4.1 Axiomatization of closure calculi

For both the calculi C and $\widehat{C}$ we need to specify the syntactic categories of closures, values and context frames, denoted c, v, and f in the C-calculus, and $\hat{c}$, $\hat{v}$, and $\hat{f}$ in the $\widehat{C}$-calculus, respectively. Since $\widehat{C}$ is an extension of C, we require that $c \subseteq \hat{c}$, $v \subseteq \hat{v}$, and $f \subseteq \hat{f}$ hold. We will apply the refocusing transformation to the $\widehat{C}$-calculus, hence we require that it fulfills all the obligations of Section 2.3 with closures taking on the role of terms.

$$
\mathtt{dec}_{\mathtt{ecamc}}\, c\, E\, v \iff \begin{cases} \mathtt{dec}_{\mathtt{ecamc}}\, c'\, (E' \cdot E)\, v & \text{if } \mathtt{dec}_{\mathtt{t}}\, c = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[c'] = \hat{c}' \\ \mathtt{dec}_{\mathtt{ctx\text{-}ecamc}}\, E\, v'\, v & \text{if } \mathtt{dec}_{\mathtt{t}}\, c = \hat{v}' \text{ and } v' = \hat{v}' \end{cases}
$$

$$
\mathtt{dec}_{\mathtt{ctx\text{-}ecamc}}\, [\,]\, v\, v' \iff v = v'
$$

$$
\mathtt{dec}_{\mathtt{ctx\text{-}ecamc}}\, (f :: E)\, v\, v' \iff \begin{cases} \mathtt{dec}_{\mathtt{ecamc}}\, c'\, (E' \cdot E)\, v' & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[c'] = \hat{c}' \\ \mathtt{dec}_{\mathtt{ctx\text{-}ecamc}}\, E\, v''\, v' & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = v'' \\ \mathtt{dec}_{\mathtt{ecamc}}\, c\, (f' :: E)\, v' & \text{if } \mathtt{dec}_{\mathtt{f}}\, f\, v = (\hat{c}, \hat{f}'), \\ & c = \hat{c} \text{ and } f' = \hat{f}' \end{cases}
$$

$$
\mathtt{eval}_{\mathtt{ecamc}}\, t\, v \iff \mathtt{dec}_{\mathtt{ecamc}}\, (t[\bullet])\, [\,]\, v
$$

Fig. 7. A generic eval/continue like semantics utilizing the C calculus

Furthermore, we require that in the C-calculus each closure is either a term with an explicit substitution (i.e., with a list of closures) or a value, and that a closure has only the empty decomposition.

Finally, we also need the following compatibility properties expressing the fact that the syntactic extension in the $\widehat{C}$-calculus is inessential with respect to the C-calculus:

$$
\begin{aligned}
\mathtt{dec}_{\mathtt{t}}\, c = \hat{r} \wedge \hat{r} \mapsto \hat{c}' &\implies \hat{c}' \text{ is a } C\text{-closure} \vee \exists c', f. f[c'] = \hat{c}' \\
\mathtt{dec}_{\mathtt{t}}\, c = \hat{v} &\implies \hat{v} \text{ is a } C\text{-value} \\
\mathtt{dec}_{\mathtt{f}}\, f\, v = \hat{r} \wedge \hat{r} \mapsto \hat{c} &\implies \hat{c} \text{ is a } C\text{-closure} \vee \exists c, f'. f'[c] = \hat{c} \\
\mathtt{dec}_{\mathtt{f}}\, f\, v = \hat{v}' &\implies \hat{v}' \text{ is a } C\text{-value} \\
\mathtt{dec}_{\mathtt{f}}\, f\, v = (\hat{c}, \hat{f}) &\implies \exists c', f'. f' = \hat{f} \wedge c' = \hat{c}
\end{aligned}
$$

4.2 Towards an efficient eval/continue machine

Most of the derivation is adapted directly from Section 3 with the transformation working over the closures of the $\widehat{C}$-calculus. However, the properties stated in Section 4.1 allow us to make two additional steps in the derivation just before changing the format of the semantics to the abstract machine. The purpose of these two steps is to expose the environment in the resulting abstract machine.

Transition compression—back to the C-calculus. After performing the refocusing steps as in the standard version, we obtain a semantics in the form of an eval/continue machine for the $\widehat{C}$-calculus. We can now exploit the connection between the two calculi C and $\widehat{C}$ to arrive at a semantics defined only on C-closures. We do this by compressing transitions that first introduce, and then immediately consume the extra syntactic constructs of the $\widehat{C}$-calculus that are

$$
\begin{array}{rcl}
\langle t \rangle_{\texttt{init}} & \rhd & \langle t, \bullet, [\,] \rangle_{\texttt{e}} \\[1ex]
\langle t, s, E \rangle_{\texttt{e}} & \rhd & \begin{cases} \langle t', s', E' \cdot E \rangle_{\texttt{e}} & \text{if } \texttt{dec}_{\texttt{t}}\, t[s] = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[t'[s']] = \hat{c} \\ \langle E' \cdot E, v \rangle_{\texttt{c}} & \text{if } \texttt{dec}_{\texttt{t}}\, t[s] = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[v] = \hat{c} \\ \langle E, v \rangle_{\texttt{c}} & \text{if } \texttt{dec}_{\texttt{t}}\, t[s] = v \end{cases} \\[1ex]
\langle [\,], v \rangle_{\texttt{c}} & \rhd & \langle v \rangle_{\texttt{fin}} \\
\langle f :: E, v \rangle_{\texttt{c}} & \rhd & \begin{cases} \langle t, s, E' \cdot E \rangle_{\texttt{e}} & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[t[s]] = \hat{c} \\ \langle E' \cdot E, v' \rangle_{\texttt{c}} & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[v'] = \hat{c} \\ \langle E, v' \rangle_{\texttt{c}} & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = v' \\ \langle t, s, f' :: E \rangle_{\texttt{e}} & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = (t[s], f') \\ \langle f' :: E, v' \rangle_{\texttt{c}} & \text{if } \texttt{dec}_{\texttt{f}}\, f\, v = (v', f') \end{cases} \\[1ex]
\texttt{eval}_{\texttt{ecam-env}}\, t\, v & \iff & \langle t \rangle_{\texttt{init}} \rhd^{+} \langle v \rangle_{\texttt{fin}}
\end{array}
$$

Fig. 8. A generic eval/continue environment-based abstract machine semantics derived from reduction semantics

not present in the C-calculus. This step results in the semantics presented in Figure 7 and it relies on the compatibility properties stated in the previous subsection. For example, we observe that the existence of the context E' and the closure c' that appear in the first clause of the definition is ensured by the first of the compatibility properties.

The correctness of this step of the derivation is summarized by the following proposition:

Proposition 3. *For any term t and any value v of a language satisfying the axioms of Section 4.2, the equivalence* $\texttt{eval}_{\texttt{ecam}}\, t\, v \iff \texttt{eval}_{\texttt{ecamc}}\, t\, v$ *holds.*

Unfolding the closures. The final step of the extended transformation consists in "unfolding" the closures into their components (i.e., terms and substitutions) and it yields an eval/continue environment-based machine. As before, we introduce the transition relation $\rhd$ together with its transitive closure. The grammar of configurations of the machine reads as follows:

$$c ::= \langle t \rangle_{\texttt{init}} \mid \langle t, s, E \rangle_{\texttt{e}} \mid \langle E, v \rangle_{\texttt{c}} \mid \langle v \rangle_{\texttt{fin}}$$

Note the change in the *eval* configuration, which now operates separately on terms and on substitutions that have become environments. The transitions of the machine and the induced evaluation function are presented in Figure 8.

The correctness of the transformation can then be stated with the following theorem, which follows from the correctness of individual transformation steps.

Theorem 3. *For any term t and any value v of a language satisfying the axioms of Section 4.2, the equivalence* $\texttt{eval}\, t\, v \iff \texttt{eval}_{\texttt{ecam-env}}\, t\, v$ *holds.*

Under conditions similar to those for the substitution-based eval/continue abstract machine of Section 3.5, one can transform the environment-based eval/continue abstract machine into an environment-based eval abstract machine.

5 Refocusing in context-sensitive reduction semantics

In this section we sketch the (minor) changes needed to adapt the formalization from Sections 2, 3 and 4 to languages with context-sensitive reduction. The refocusing method has been formalized for both substitution-based and closure-based source languages.

The notion of *context-sensitive reduction semantics* was first introduced by Biernacka and Danvy in order to deal with languages with multiple binders in the refocusing framework [2]. They also used it to account for languages with control effects such as the control operators `call/cc` or `shift` and `reset` [3].

In a standard reduction semantics the contracting function has type `redex` $\to$ `term`. This, however, can be insufficient for languages that contain more sophisticated constructs, where contraction depends not only on the redex, but also on the shape of the entire context surrounding that redex. For example, the control operator `call/cc` can be seen as a binding construct that captures the context which can then be applied to a value inside its body. Now, when we plug a term built with `call/cc` in a context, we trigger a contraction, where the structure of the contracted term depends on the structure of the context.

This more general notion of reduction requires only a small adaptation in the reduction semantics: we need to change the type of the contracting function into `redex` $\times$ `context` $\to$ `term` $\times$ `context`. Such a formulation of reduction semantics admits refocusing, as no part of the transformation depends on any specific properties of contraction.

The changes in the formalization needed to account for context-sensitive reduction are minor. In the axiomatization, they consist only in changing the type of contraction and propagating this change in the `dec`, `iter`, and `eval` relations. This change is then propagated through definitions of the refocusing steps and proofs without any impact on the structure or difficulty of proofs. As mentioned above, besides introducing context-sensitive reductions in the standard transformation, a combination with the environment-based extension is provided, as this is the setting in which context-sensitive reduction semantics have appeared and is a source of many interesting case studies.

6 Conclusion

We have formalized and proved correct the refocusing derivation method in the Coq proof assistant. The formalization is done as a generic framework and can be used to derive abstract machines for any language satisfying the requirements described in Section 2 (and in Section 4 for closure calculi). These (standard) requirements have to be packaged in a Coq module as specified in the signature. The output is an abstract machine extensionally equivalent to the initial

semantics and is obtained by applying a sequence of functors to the module implementing the reduction semantics. The correctness of the final machine is a consequence of the correctness of each step of the transformation which in turn is ensured by each functor. The framework is quite general: it allows one to express languages with meta-level substitutions or with explicit substitutions, as well as languages with context-sensitive reduction. It is also possible to express the transition function of the final abstract machine as a Coq function and to employ the code extraction mechanism of Coq to generate a certified executable implementation of the machine.

The axiomatization of reduction semantics that we present in this article seems usable in practice, but it would be interesting to see if there are other, simpler axiomatizations, especially for languages with closures. Also, further automatization of some of the tasks that are now delegated to the user—including providing atomic decomposition functions—should be investigated.

Whereas the current article shows the subsequent semantics in the derivation chain are extensionally equivalent, there is an ongoing work on characterizing such equivalence in terms of execution traces keeping track of the reduction sequence. This approach can further lead, with the help of coinductive reasoning of Coq, to refocusing and its correctness proof for potentially infinite computations.

Acknowledgements We would like to thank Olivier Danvy and the anonymous reviewers of IFL'10 for numerous useful comments on the presentation of this work. This work has been supported by the MNiSW grant number N N206 357436, 2009-2011.

References

1. Małgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. In Jürgen Giesl, editor, *Preliminary proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, Paris, France, June 2007.
2. Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007.
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. Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, number 5700 in Lecture Notes in Computer Science, pages 186–206. Springer, 2009.
5. Jacek Chrząszcz. Implementing modules in the Coq system. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.
6. Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

7. 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 Press. Invited talk.
8. Olivier Danvy. From reduction-based to reduction-free normalization. In *Advanced Functional Programming, Sixth International School*, number 5832 in Lecture Notes in Computer Science, pages 64–164, Nijmegen, The Netherlands, May 2008. Springer-Verlag.
9. Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76:302–323, 2010.
10. Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.
11. Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer.
12. 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.
13. Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. In María Alpuente, editor, *20th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2010*, number 3-30 in RISC-Linz Report Series, Castle of Hagenberg, Austria, July 2010. Research Institute for Symbolic Computation (RISC), Johannes Kepler University in Linz. Invited talk.
14. Olivier Danvy, Ian Zerny, and Jacob Johannsen. A walk in the semantic park. In *Proceedings of the 2011 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011)*, Austin, USA, January 2011. ACM Press. Invited talk.
15. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.
16. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. *Logical Methods in Computer Science*, 6(3:1):1–39, July 2010.
17. Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.