Formalizing Constructions of Abstract Machines for Functional Languages in Coq

Małgorzata Biernacka[1]

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Parc Orsay Université, F-91893

Dariusz Biernacki[2]

INRIA Futurs, ProVal, Parc Orsay Université, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract

This note reports on preliminary work on formalizing and mechanizing derivations of abstract machines from reduction semantics of functional languages in Coq, based on the method of refocusing due to Danvy and Nielsen. The ultimate goal is to develop a general framework for representing programming languages and their various semantic specifications inside Coq, and to automatize derivations between these specifications while ensuring their correctness.
In this note, we describe two case studies of derivations of abstract machines from reduction semantics (i.e., machines realizing the reduction strategies given by the corresponding reduction semantics): we consider the standard call-by-value lambda calculus, for which the derived machine is the substitution-based CK machine of Felleisen and Friedman, and the call-by-name variant of the calculus of closures which leads to the environment-based Krivine machine. These correspondences have been discovered and reported in previous work by the first author and Danvy in the setting of functional programs and their transformations, whereas here we adopt a relational approach and we formalize the derivation steps and prove them correct within Coq's type theory. We also discuss some technical issues of our formalization and we discuss a possible construction of the general framework for refocusing in Coq.

Keywords: abstract machine, reduction semantics, refocusing, Coq

1 Introduction

Typically, a programming language is given several different semantic specifications, each designed for a particular purpose and each preferred by either the language designer, implementor or programmer. In order for all these different specifications to define the same behaviour, we need to provide proofs of their equivalence. With some practice, this task can be performed by hand and indeed, it usually is done – whenever the need arises, in an ad hoc manner.

[1] Email: mbiernac@lri.fr
[2] Email: dabi@lri.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

We argue that it need not be so, and in many cases we could benefit from considering systematic and to some point automatic derivation methods that would allow one to arrive from one semantic description taken as standard, to another one that is more convenient in a given application, while the correctness of the outcome is guaranteed by the derivation method. This work not only saves one the tedious task of proving similar kinds of theorems whenever one introduces even a mild variation into a language, but it also may help connect the same computational features as they occur in different contexts.

There has been some work aiming at connecting various forms of operational semantics of functional languages. In particular, Danvy et al. have discovered the functional correspondence between higher-order evaluators and abstract machines [1,2,3] when these semantic descriptions are implemented as ML-programs: one can then interderive them by means of standard program transformation techniques (CPS transformation, defunctionalization, closure conversion). In another line of work, connections between reduction semantics and abstract machines have been studied, and it has been discovered how the refocusing method provides a way to systematically derive the latter from the former [5,6,9]. This work studies the correspondence not only for the (standard) lambda calculus but also for a wide class of its extensions with a non-standard notion of reduction, thus accomodating such non-local computational tools and effect as, e.g., control operators, assignment, state, lazy evaluation, etc [6]. In particular, it has been applied to calculi of closures in order to materialize the slogan that calculi of explicit substitutions correspond to environment machines, and to context-sensitive reduction semantics that give rise to store-based machines. In addition, the systematic method shows how an abstract machine realizes a particular reduction strategy specified by the underlying reduction semantics. As with the functional correspondence, implementations of semantic descriptions in a functional programming language have been considered and the correctness of the derived machines relies on the correctness of standard program transformation techniques.

Since we argue for the generality and the uniformity of these correspondences, it is natural to try to automate them in an appropriate tool. In the present work, we report on an implementation of two case studies and we give some ideas on how a proof assistant can be used to aid in the tasks outlined above. Developing a general framework for expressing arbitrary languages and semantic descriptions is left as future work.

In this note, we look at the refocusing method from a different point of view – we recast the development in the language of Coq [7] and we then give formal proofs of correctness of all refocusing steps, applied to two prototypical applicative programming languages: in order to show the plain version of the method, we apply it to the standard call-by-value lambda calculus leading to the CK abstract machine [11], and to illustrate one of its extensions, the second case study focuses on the call-by-name version of the extended Curien's calculus of closures [8] as introduced in Biernacka and Danvy [5]. We suggest a logical characterization of what it means for an abstract machine to realize a particular reduction strategy, and we identify the general properties of each of the intermediate derivation steps and we hope to be able to use them in a general framework. In this work, we

consider only abstract machines, i.e., transition systems operating on the source code, as opposed to virtual machines, operating on compiled code. [3]

The previous articles on refocusing and the present one look at computation in the lambda calculus from the rewriting point of view. However, we must warn the reader that in this particular context, where the focus is on evaluation rather than general normalization, it is often convenient (and by now also standard) to use some term-rewriting concepts in a slightly different way, and to deviate from some traditional definitions. However, all the necessary definitions are introduced.
Prerequisites: We assume basic familiarity with the Coq proof assistant.

2 Refocusing in reduction semantics

The starting point of refocusing is a specification of a reduction semantics. Let us briefly recall its ingredients.

2.1 Reduction semantics

Reduction semantics is a form of small-step operational semantics based on term rewriting. In our approach, we assume it is given by the syntactic categories of terms, values (non-reducible terms taken as correct results of evaluation), redexes (or, more precisely: potential redexes possibly including "smallest" stuck terms), evaluation contexts and two functions: `contract` – performing a basic computation step (i.e., rewriting a redex) and `plug` – giving meaning to the evaluation contexts by assigning a term to each pair of a term and a context. (If we look at contexts as "terms with a hole", then the plug function simply fills the hole with a given term.) Evaluation contexts together with the `plug` function determine the reduction strategy: the next redex to reduce in a given term is found by decomposing the term into a redex and a context. The decomposition is either a pair of a value and the empty context (if the decomposed term is already a value) or a pair of a redex and a context such that plugging the redex in the context gives the original term. We furthermore assume that a value cannot be decomposed into a redex and an evaluation context.

The (naive) process of evaluation then consists in repeatedly decomposing a given term into a redex and an evaluation context (if it is not already a value), contracting the redex (if it is not a stuck redex) and plugging the contractum into the context. In general, evaluation may be nonterminating or stuck.

For the refocusing method to apply, we need to ensure the unique-decomposition property of the reduction semantics; in effect, we only allow sequential computation.

We say that a term t decomposes into another term t_0 and an evaluation context c, if $\texttt{plug}\,(c, t_0) = t$, where $=$ stands for syntactic equality. We use the notation using an explicit call to function `plug` rather than the usual notation $c[t_0]$ whenever we want to emphasize the fact that we mean the actual implementation of this operation by appropriately defined functions (here, in Coq). We further say that a decomposition (t, c) is trivial if c is the empty context or if t is a value. Hence, it is

[3] The distinction has been introduced by Ager et al. [1].

convenient to define potential redexes as non-value terms whose all decompositions are trivial.

The unique decomposition property can be formulated as follows:

Definition 2.1 We say that the reduction semantics has the unique-decomposition property if for each term t, t is either a value or there exists a unique redex r and a unique context c such that $\texttt{plug}\,(c, r) = t$ (i.e., for all redexes r, r' and evaluation contexts c, c', if $\texttt{plug}\,(c, r) = t$ and $\texttt{plug}\,(c', r') = t$, then $r = r'$ and $c = c'$.

Providing these ingredients is sufficient to precisely define a reduction semantics for a language, even though we leave the question of *decomposing* a term unspecified. Let us now consider this question in more detail. Clearly, when we want to implement the naive evaluation procedure according to a given reduction semantics, we need to write a decomposing function which, for a given non-value term, will return a pair of a redex and a context. Furthermore, we assume that for values, the decomposing function returns this value and the empty context and thus the function is total. We say that any function `decompose` from terms to decompositions is correct with respect to a given reduction semantics if and only if it is the right inverse of the `plug` function, i.e., if $\texttt{plug} \circ \texttt{decompose} = \texttt{id}$. Now, the consequence of the unique decomposition is the fact that any correct decomposing function will return the same decomposition for a given term. Hence the implementation of the evaluation process remains correct regardless of the particular implementation of the decomposing function, as long as this function is correct in the above sense.

In this work, however, we adopt a converse approach, i.e., rather than proving the unique decomposition directly (since this task can be quite involved [13]), we provide an equivalent condition using a decomposition function with an additional property that can be proved relatively easily if the decomposition function is defined recursively over terms and contexts.

Lemma 2.2 (Unique decomposition) *Assume we have total functions* `plug` *and* `decompose`. *If the decomposition function is correct with respect to the plug function, i.e., if* $\texttt{plug} \circ \texttt{decompose} = \texttt{id}$, *and moreover, if* $\texttt{decompose}(\texttt{plug}\,(\texttt{c}, \texttt{r})) = (\texttt{r}, \texttt{c})$ *for any redex* r *and any context* c, *then the reduction semantics has the unique decomposition property.*

The lemma states that it is enough to write a decomposition function and prove that it satisfies the two conditions in order to verify the unique decomposition property of the reduction semantics, and in fact in our implementation this approach turned out to be more convenient, since for applying refocusing we need to write a decomposition function anyway. (The correctness condition is needed to ensure existence of a decomposition, and the second condition is necessary for proving uniqueness.) There are several natural ways of implementing the decomposing function: either naively by recursive descent over terms, or derived from the plug function, roughly by inverting its defining clauses.

2.2 Refocusing

Refocusing is a derivation method used to obtain abstract machines from specifications of reduction semantics. By an abstract machine we mean a transition system

simulating evaluation of terms, with an explicit representation of control stacks, stores, etc., as found in real-life implementations, but still at a sufficiently abstract level to facilitate reasoning about execution without drowning in implementation details. The virtue of refocusing is to show how certain features of a higher-level specification of a language become concrete in the corresponding abstract machine, and in which sense an abstract machine realizes a particular evaluation strategy for a given language.

Originally, refocusing has been presented as an algorithm for constructing an efficient evaluation function, avoiding reconstructing intermediate terms in the decompose-contract-plug loop, as described in the previous section [10]. Only subsequently, it has been observed that, in effect, refocusing yields abstract machines. Moreover, it has been extended to account for calculi of explicit substitutions that yield environment-based machines, and to context-sensitive reduction semantics, leading to abstract machines with global store [5]. In the latter work, the method has been implemented, presented and studied in the setting of functional programming languages, and the correctness of the derivation steps have been established as a consequence of the correctness of the applied program transformations.

In this work, we shift to a more abstract view: we represent all the intermediate results (i.e., various, extensionally equivalent, functions computing the value of a term according to the same strategy, but becoming closer to a transition system) as relations (inductive predicates in Coq), and we provide uniform induction principles for proving correctness of the derivation steps that can be made into Coq tactics. Let us now discuss some implementation features.

2.3 The Coq proof assistant

Almost all the ingredients of reduction semantics as described in Section 2.1 can be represented in Coq in an intuitive way. The only problem arises when representing evaluation— it cannot be written as a function, since it is possibly nonterminating. Therefore, we choose to represent all the functions (except for `plug`) as relations, and in fact, this representation is quite natural and it gives rise to simple and uniform correctness proofs of intermediate transition systems: the Coq-generated induction principles are used to prove the simulations by induction on derivations (which corresponds to proofs by cases in the functional specification of these transition systems).

Below we present the main features of our implementation, common to both languages that we consider, and therefore suggesting a solution to be applied in a general framework. [4]

- We represent terms, values and redexes as different sets (datatypes) and we provide injection functions for values and redexes into terms, together with their injectivity proofs (the functions `value_to_term` and `redex_to_term` provide coercions from the set of values and from the set of redexes, respectively, to the set of terms).

[4] In the remainder of the article, we include fragments of Coq code, but at times we diverge from the Coq syntax for the sake of readability and accessibility by readers not familiar with Coq. The complete Coq development can be found at `http://www.lri.fr/~mbiernac/refocusing`.

Parameters `term value redex` : **Set**.
Parameter `value_to_term` : `value` → `term`.
Parameter `redex_to_term` : `redex` → `term`.
Axiom `value_to_term_injective` :
$\forall\, v, v'.\, \texttt{value\_to\_term}\ \texttt{v} = \texttt{value\_to\_term}\ \texttt{v}' \to \texttt{v} = \texttt{v}'$.
Axiom `redex_to_term_injective` :
$\forall\, r, r'.\, \texttt{redex\_to\_term}\ \texttt{r} = \texttt{redex\_to\_term}\ \texttt{r}' \to \texttt{r} = \texttt{r}'$.

- The reduction strategy is given by the grammar of reduction contexts and the plug function:

  Parameter `context` : **Set**.
  Parameter `plug` : `term` → `context` → `term`.

- The notion of a potential redex allows us to prove the totality of decomposition, i.e., for every non-value term there exists a decomposition of that term into a potential redex and a context:

  Axiom `decomposition` :
  $\forall\, t : \texttt{term}.\, (\exists\, v : \texttt{value}.\, t = v) \vee (\exists\, r : \texttt{redex}, c : \texttt{context}.\, t = \texttt{plug}\,(r,\, c))$.

- Having proven the decomposition theorem, we can write a total decomposition function. The particular definition, obtained by inverting the plugging function is naturally defined by two mutually recursive functions: one descending over the structure of the currently decomposed term, and an auxiliary function descending over the surrounding evaluation context when a value is encountered. This algorithm cannot be easily represented in Coq as a function, and so we choose to represent it as a relation. (It is possible to define this function in Coq, using the `Function` command with an appropriate measure function that takes into account both the term and the surrounding evaluation context. Such solution, however, does not seem to be worth the effort, since in effect it is defined using a well-founded relation that coincides with ours, and the computational overhead is significant.)

  Parameter `decomp` : **Set**.
  Parameter `decompose` : `term` → `context` → `decomp` → **Prop**.
  Parameter `decompose_ctx` : `context` → `value` → `decomp` → **Prop**.

- To complete the definition of reduction semantics, we need to ensure the unique decomposition property. As mentioned in Section 2.1, we choose to do it by proving the following condition on the function decompose, which implies the condition required in Lemma 2.2:

  Axiom `decompose_deterministic` : $\forall\, \mathbf{c}, \mathbf{c}'$: `context`, $\mathbf{t}$: `term`, $\mathbf{d}$: `decomp`.
  $\texttt{decompose}\,(\texttt{plug}\,(\texttt{t},\, \texttt{c}))\,\texttt{c}'\,\texttt{d} \leftrightarrow \texttt{decompose}\,\texttt{t}\,(\texttt{c} \circ \texttt{c}')\,\texttt{d}$.

  where $c \circ c'$ denotes the standard concatenation of contexts.

- All the intermediate functions are represented as relations and we prove that each step is correct in the sense that the evaluation relations defined on the basis of each of them are correct. Proofs of correctness are done by induction on derivations using the induction principles generated by the `Scheme` command (mutual induction in cases where there are mutually recursive relations).

3 From a reduction semantics to an abstract machine for the lambda calculus

In this section we detail the implementations of the derivation of abstract machines for two languages: the standard call-by-value λ-calculus, leading to the substitution-based CK machine, and the call-by-name extended calculus of closures, leading to the environment-based Krivine machine.

3.1 The call-by-value lambda-calculus

We consider the call-by-value λ-calculus with names drawn from a countable set `var_name`. [5]

$$\frac{x : \texttt{var\_name}}{\texttt{var}\,x : \texttt{term}} \qquad \frac{x : \texttt{var\_name} \quad t : \texttt{term}}{\texttt{lam}\,(x,\, t) : \texttt{term}} \qquad \frac{t_0 : \texttt{term} \quad t_1 : \texttt{term}}{\texttt{app}\,(t_0,\, t_1) : \texttt{term}}$$

3.1.1 A reduction semantics

A value is either a variable or a lambda abstraction, and a potential redex is an application of a value to a value (i.e., by the definition of Section 2.1, a non-value term with only trivial decompositions).

$$\frac{x : \texttt{var\_name}}{\texttt{v\_var}\,x : \texttt{value}} \qquad \frac{x : \texttt{var\_name} \quad t : \texttt{term}}{\texttt{v\_lam}\,(x,\, t) : \texttt{value}} \qquad \frac{v_0 : \texttt{value} \quad v_1 : \texttt{value}}{\texttt{beta}\,(v_0,\, v_1) : \texttt{redex}}$$

In the presence of ill-formed (stuck) terms, `contract` is a partial function, undefined for an application of a variable to a value.

$$\begin{aligned}\texttt{contract}\,(\texttt{beta}\,(v_0,\, v_1)) = {}& \textbf{match}\, v_0\, \textbf{with}\\ & |\,\texttt{var}\,x \Rightarrow \texttt{None}\\ & |\,\texttt{lam}\,(x,\, t) \Rightarrow \texttt{Some}\,(\texttt{subst}\,(t,\, x,\, v_1))\\ & \textbf{end}\end{aligned}$$

where $\texttt{subst} : \texttt{term} \to \texttt{var\_name} \to \texttt{term} \to \texttt{term}$ is a function performing the standard capture-avoiding substitution.

The following two definitions encode the call-by-value, left-to-right reduction strategy: first we give the structure of evaluation contexts, and below we give them a meaning by defining the `plug` function.

$$\frac{}{\texttt{mt} : \texttt{context}} \qquad \frac{v : \texttt{value} \quad c : \texttt{context}}{\texttt{ap\_l}\,(v,\, t) : \texttt{context}} \qquad \frac{t : \texttt{term} \quad c : \texttt{context}}{\texttt{ap\_r}\,(t,\, c) : \texttt{context}}$$

$$\begin{aligned}\texttt{plug}\,(t,\, c) = {}& \textbf{match}\, c\, \textbf{with}\\ & |\,\texttt{mt} \Rightarrow t\\ & |\,\texttt{ap\_l}\,(v,\, c') \Rightarrow \texttt{plug}\,(\texttt{app}\,(v,\, t),\, c')\\ & |\,\texttt{ap\_r}\,(t',\, c') \Rightarrow \texttt{plug}\,(\texttt{app}\,(t,\, t'),\, c')\\ & \textbf{end}\end{aligned}$$

Evaluation contexts c are represented "inside out" (this interpretation is defined by the function `plug`) which makes them suitable for performing evaluation by an abstract machine, where contexts become stacks: pushing a term on the stack

[5] In the sequel, we use inference rules to represent inductive datatypes and relations in Coq, keeping the original constructor names.

corresponds to the construction of the context, and popping a term from the stack corresponds to the destruction of the context.

A decomposition is either a pair of a redex and a context, or a value in the empty context, representing a single value:

$$\frac{v : \texttt{value}}{\texttt{d\_val}\, v : \texttt{decomposition}} \qquad \frac{r : \texttt{redex} \quad c : \texttt{context}}{\texttt{d\_red}\,(r,\, c) : \texttt{decomposition}}$$

Evaluation is then implemented as a relation representing the decompose-contract-plug loop, where the decomposing function is represented as a relation as shown below.

$$\frac{\texttt{decctx}\,(c,\, v,\, d)}{\texttt{dec}\,(v,\, c,\, d)}\ (\texttt{val\_ctx}) \qquad \frac{\texttt{dec}\,(t_0,\, \texttt{ap\_r}\,(t_1,\, c),\, d)}{\texttt{dec}\,(\texttt{app}\,(t_0,\, t_1),\, c,\, d)}\ (\texttt{app\_ctx})$$

$$\frac{}{\texttt{decctx}\,(\texttt{mt},\, v,\, \texttt{d\_val}\, v)}\ (\texttt{mt\_dec}) \qquad \frac{\texttt{dec}\,(t,\, \texttt{ap\_l}\,(v,\, c),\, d)}{\texttt{decctx}\,(\texttt{ap\_r}\,(t,\, c),\, v,\, d)}\ (\texttt{ap\_r\_dec})$$

$$\frac{}{\texttt{decctx}\,(\texttt{ap\_l}\,(v_0,\, c),\, v_1,\, \texttt{d\_red}\,(\texttt{beta}\,(v_0,\, v_1),\, c))}\ (\texttt{ap\_l\_dec})$$

$$\frac{\texttt{dec}\,(t,\, \texttt{mt},\, d)}{\texttt{decmt}\,(t,\, d)}\ (\texttt{d\_intro}) \qquad \frac{}{\texttt{iter}\,(\texttt{d\_val}\, v,\, v)}\ (\texttt{i\_val})$$

$$\frac{\texttt{contract}\,(r) = \texttt{Some}\, t \quad \texttt{decmt}\,(\texttt{plug}\,(t,\, c),\, d) \quad \texttt{iter}\,(d,\, v)}{\texttt{iter}\,(\texttt{d\_red}\,(r,\, c),\, v)}\ (\texttt{i\_red})$$

$$\frac{\texttt{decmt}\,(t,\, d) \quad \texttt{iter}\,(d,\, v)}{\texttt{eval}\,(t,\, v)}\ (\texttt{e\_intro})$$

It is straightforward to prove that `decmt`, `iter` and `eval` all define functions. `decmt` is a total function, returning a decomposition for all terms, but `iter` and `eval` are not total, since they are not defined for stuck terms. (If needed, they could be made total by introducing an additional constructor handling stuck terms in the relation `iter`.)

3.1.2 A pre-abstract machine

If the decomposing function satisfies the following property:

$$\forall\, t,\, c,\, d.\, \texttt{decmt}\,(\texttt{plug}\,(t,\, c),\, d) \leftrightarrow \texttt{dec}\,(t,\, c,\, d)$$

then in the relations of Section 3.1.1 we can replace the calls to `decmt` by calls to `dec`, and we obtain the following definitions of $\texttt{iter}_0$ and $\texttt{eval}_0$:

$$\frac{}{\texttt{iter}_0\,(\texttt{d\_val}\, v,\, v)}\ (\texttt{i\_val0}) \qquad \frac{\texttt{contract}\,(r) = \texttt{Some}\, t \quad \texttt{dec}\,(t,\, c,\, d) \quad \texttt{iter}_0\,(d,\, v)}{\texttt{iter}_0\,(\texttt{d\_red}\,(r,\, c),\, v)}\ (\texttt{i\_red0})$$

$$\frac{\texttt{dec}\,(t,\, \texttt{mt},\, d) \quad \texttt{iter}_0\,(d,\, v)}{\texttt{eval}_0\,(t,\, v)}\ (\texttt{e\_intro0})$$

The two relations `iter` and $\texttt{iter}_0$ are equivalent, and we prove it by induction on derivations.

The resulting definition is called a pre-abstract machine because it already optimizes the decompose-contract-plug loop by avoiding the reconstruction of intermediate terms, and directly proceeding to decomposing the contractum in the given evaluation context.

3.1.3 A staged abstract machine

Next, we observe that whenever we reach a decomposition, it will immediately be consumed by $\mathtt{iter}_0$, therefore we can distribute the calls to $\mathtt{iter}_0$ in the definition of `decctx`. This way, the new relations $\mathtt{dec}_1$ and $\mathtt{decctx}_1$ implement compatibility rules of the semantics (traversing terms and contexts in search of a redex) and $\mathtt{iter}_1$ implements contraction and immediately performs decomposition on the contractum.

$$\frac{\mathtt{decctx}_1\,(c,\,v,\,v')}{\mathtt{dec}_1\,(v,\,c,\,v')}\,(\mathtt{val\_ctx1}) \qquad \frac{\mathtt{dec}_1\,(t_0,\,\mathtt{ap\_r}\,(t_1,\,c),\,v)}{\mathtt{dec}_1\,(\mathtt{app}\,(t_0,\,t_1),\,c,\,v)}\,(\mathtt{app\_ctx1})$$

$$\frac{\mathtt{iter}_1\,(\mathtt{d\_val}\,v,\,v')}{\mathtt{decctx}_1\,(\mathtt{mt},\,v,\,v')}\,(\mathtt{mt\_dec1}) \qquad \frac{\mathtt{dec}_1\,(t,\,\mathtt{ap\_l}\,(v,\,c),\,v')}{\mathtt{decctx}_1\,(\mathtt{ap\_r}\,(t,\,c),\,v,\,v')}\,(\mathtt{ap\_r\_dec1})$$

$$\frac{\mathtt{iter}_1\,(\mathtt{d\_red}\,(\mathtt{beta}\,(v_0,\,v_1),\,c),\,v)}{\mathtt{decctx}_1\,(\mathtt{ap\_l}\,(v_0,\,c),\,v_1,\,v)}\,(\mathtt{ap\_l\_dec1})$$

$$\frac{}{\mathtt{iter}_1\,(\mathtt{d\_val}\,v,\,v)}\,(\mathtt{i\_val1}) \qquad \frac{\mathtt{contract}\,(r) = \mathtt{Some}\,t \quad \mathtt{dec}_1\,(t,\,c,\,v)}{\mathtt{iter}_1\,(\mathtt{d\_red}\,(r,\,c),\,v)}\,(\mathtt{i\_red1})$$

$$\frac{\mathtt{dec}_1\,(\mathtt{mt},\,t,\,v)}{\mathtt{eval}_1\,(t,\,v)}\,(\mathtt{e\_intro1})$$

Theorem 3.1 (`eval01`) $\forall\, t,\, v.\, \mathtt{eval}_0\,(t,\,v) \leftrightarrow \mathtt{eval}_1\,(t,\,v)$.

The proof of equivalence is done by induction on derivations and relies on the totality of `dec` and on the following auxiliary properties, capturing the nature of the transition from the pre-abstract machine to the staged abstract machine.

(i) $\forall\, t,\, c,\, d.\, \mathtt{dec}\,(t,\,c,\,d) \Rightarrow \forall\, v.\, \mathtt{iter}_1\,(d,\,v) \Rightarrow \mathtt{dec}_1\,(t,\,c,\,v)$

(ii) $\forall\, c,\, v,\, d.\, \mathtt{decctx}_0\,(c,\,v,\,d) \Rightarrow \forall\, v'.\, \mathtt{iter}_1\,(d,\,v') \Rightarrow \mathtt{decctx}_1\,(c,\,v,\,v')$

(iii) $\forall\, t,\, c,\, v.\, \mathtt{dec}_1\,(t,\,c,\,v) \Rightarrow \forall\, d.\, \mathtt{dec}_0\,(t,\,c,\,d) \Rightarrow \mathtt{iter}_0\,(d,\,v)$

(iv) $\forall\, c,\, v,\, v'.\, \mathtt{decctx}_1\,(c,\,v,\,v') \Rightarrow \forall\, d.\, \mathtt{decctx}_0\,(c,\,v,\,d) \Rightarrow \mathtt{iter}_0\,(d,\,v')$

3.1.4 An eval/apply abstract machine

The next step consists in inlining the iterating function and obtaining a proper abstract machine, i.e., a transition system without a trampoline function triggering the loop. This stage makes us "forget" the sites where we found redexes and performed reductions, and that is why it is not always trivial to unwind abstract machines of this form to read the underlying reduction semantics [4,11,12]. The eval/apply abstract machine consists of two kinds of transitions: one that analyzes terms and one that analyzes reduction contexts. [6]

[6] In the *eval/apply* model the function in a function call is evaluated and applied to its arguments.

$$\frac{\texttt{decctx}_2\,(c,\,v,\,v')}{\texttt{dec}_2\,(v,\,c,\,v')}\;(\texttt{val\_ctx2}) \qquad \frac{\texttt{dec}_2\,(t_0,\,\texttt{ap\_r}\,(t_1,\,c),\,v)}{\texttt{dec}_2\,(\texttt{app}\,(t_0,\,t_1),\,c,\,v)}\;(\texttt{app\_ctx2})$$

$$\frac{}{\texttt{decctx}_2\,(\texttt{mt},\,v,\,v)}\;(\texttt{mt\_dec2}) \qquad \frac{\texttt{dec}_2\,(t,\,\texttt{ap\_l}\,(v,\,c),\,v')}{\texttt{decctx}_2\,(\texttt{ap\_r}\,(t,\,c),\,v,\,v')}\;(\texttt{ap\_r\_dec2})$$

$$\frac{\texttt{contract}\,(\texttt{beta}\,(v_0,\,v_1)) = \texttt{Some}\,t \quad \texttt{dec}_2\,(t,\,c,\,v)}{\texttt{decctx}_2\,(\texttt{ap\_l}\,(v_0,\,c),\,v_1,\,v)}\;(\texttt{ap\_l\_dec2})$$

$$\frac{\texttt{dec}_2\,(t,\,\texttt{mt},\,v)}{\texttt{eval}_2\,(t,\,v)}\;(\texttt{e\_intro2})$$

Again, the equivalence of the two evaluation relations is proved in each direction by induction on derivations of the corresponding decomposing function. Moreover, for the left-to-right direction we need the auxiliary property:

$$\begin{aligned}\forall\, d,\, v.\, \texttt{iter}_1\,(d,\,v) \rightarrow\ & \textbf{match}\ d\ \textbf{with}\\ & \mid \texttt{d\_val}\,v' \Rightarrow \texttt{decctx}_2\,(\texttt{mt},\,v',\,v)\\ & \mid \texttt{d\_red}\,(r,\,c) \Rightarrow \texttt{dec}_2\,(r,\,c,\,v)\\ & \textbf{end}\end{aligned}$$

3.1.5 The CK abstract machine

The definition from Section 3.1.4 is a relational presentation of an abstract machine. Another, more traditional presentation of such a transition system is shown below: we arrive at it simply transposing each inference rule described in $\texttt{dec}_2$ and $\texttt{decctx}_2$ into a new relation between the premise and the conclusion of each rule, giving rise to two kinds of configurations corresponding to these relations. We also add the initial and final configurations (and their corresponding transitions), corresponding to loading the machine with a term to evaluate, and recovering the value, respectively. (These transitions are read off the $\texttt{eval}_2$ relation.)

$$\frac{t : \texttt{term}}{\texttt{c\_init}\,t : \texttt{configuration}} \qquad \frac{t : \texttt{term} \quad c : \texttt{context}}{\texttt{c\_eval}\,(t,\,c) : \texttt{configuration}}$$

$$\frac{c : \texttt{context} \quad v : \texttt{value}}{\texttt{c\_apply}\,(c,\,v) : \texttt{configuration}} \qquad \frac{v : \texttt{value}}{\texttt{c\_final}\,v : \texttt{configuration}}$$

$$\begin{aligned}
\texttt{c_init}\,t &\rhd \texttt{c_eval}\,(t,\,\texttt{mt})\\
\texttt{c_eval}\,(v,\,c) &\rhd \texttt{c_apply}\,(c,\,v)\\
\texttt{c_eval}\,(\texttt{app}\,(t_0,\,t_1),\,c) &\rhd \texttt{c_eval}\,(t_0,\,\texttt{ap_r}\,(t_1,\,c))\\
\texttt{c_apply}\,(\texttt{mt},\,v) &\rhd \texttt{c_final}\,v\\
\texttt{c_apply}\,(\texttt{ap_l}\,(v_0,\,c),\,v_1) &\rhd \texttt{c_eval}\,(t,\,c) \ \text{ if } \texttt{contract}\,(\texttt{beta}\,(v_0,\,v_1)) = \texttt{Some}\,t\\
\texttt{c_apply}\,(\texttt{ap_r}\,(t,\,c),\,v) &\rhd \texttt{c_eval}\,(t,\,\texttt{ap_l}\,(v,\,c))
\end{aligned}$$

$$\frac{\texttt{c\_init}\,t \rhd^* \texttt{c\_final}\,v}{\texttt{eval}_3\,(t,\,v)}\;\texttt{e\_intro}_3$$

The transitive closure of transitions is defined in a standard way and is represented by an inductive predicate `transitive_closure : configuration → configuration → Prop`.

The equivalences are proved as before with an auxiliary property:

$$\begin{array}{l}\forall\, C_0, C_1.\, C_0 \rhd^* C_1 \to \textbf{match}\, C_0, C_1\, \textbf{with}\\ \quad \mid \mathtt{c\_eval}\,(t,\, c), \mathtt{c\_final}\, v \;\Rightarrow\; \mathtt{dec_2}\,(t,\, c,\, v)\\ \quad \mid \mathtt{c\_apply}\,(c,\, v'), \mathtt{c\_final}\, v \;\Rightarrow\; \mathtt{decctx_2}\,(c,\, v',\, v)\\ \quad \mid \_, \_ \;\Rightarrow\; \textbf{True}\\ \quad \textbf{end}\end{array}$$

The resulting abstract machine coincides with Felleisen and Friedman's cannonical substitution-based abstract machine for evaluating λ-terms under call-by-value [11].

3.2 The call-by-name lambda calculus with explicit substitutions

In this section we discuss the call-by-name λ-calculus with de Bruijn indices and with closures (a weak version of explicit substitutions). Due to lack of space, we only outline the most important differences with respect to the previous section.

3.2.1 A reduction semantics

Let us start with the specification of the language and its reduction semantics. The terms are defined similarly as for the previous case (see p. 7), but with de Bruijn indices to represent variables, and we introduce a new syntactic category of closures, representing terms with delayed substitutions. From now on, all reduction takes place at the level of closures, and not at the level of terms.

$$\frac{t : \mathtt{term} \quad s : \mathtt{list\ closure}}{\mathtt{pair}\,(t,\, s) : \mathtt{closure}} \qquad \frac{cl_0, cl_1 : \mathtt{closure}}{\mathtt{comp}\,(cl_0,\, cl_1) : \mathtt{closure}}$$

A value is a closure representing a lambda abstraction paired with an arbitrary delayed substitution (i.e., with a list of closures).

$$\frac{t : \mathtt{term} \quad s : \mathtt{list\ closure}}{\mathtt{v\_val}\,(t,\, s) : \mathtt{value}} \qquad \frac{v : \mathtt{value} \quad cl : \mathtt{closure}}{\mathtt{r\_beta}\,(v,\, s) : \mathtt{redex}}$$

$$\frac{n : \mathtt{nat} \quad s : \mathtt{list\ closure}}{\mathtt{r\_get}\,(n,\, s) : \mathtt{redex}} \qquad \frac{t_0, t_1 : \mathtt{term} \quad s : \mathtt{list\ closure}}{\mathtt{r\_app}\,(t_0,\, t_1,\, s) : \mathtt{redex}}$$

$$\begin{array}{l}\mathtt{contract}\, r = \textbf{match}\, r\, \textbf{with}\\ \quad \mid \mathtt{r\_beta}\,(\mathtt{v\_val}\,(t,\, s),\, cl) \;\Rightarrow\; \mathtt{Some}\,(\mathtt{pair}\,(t,\, cl :: s))\\ \quad \mid \mathtt{r\_get}\,(n,\, s) \;\Rightarrow\; \mathtt{nth\_option}\,(s,\, n)\\ \quad \mid \mathtt{r\_app}\,(t,\, s,\, cl) \;\Rightarrow\; \mathtt{Some}\,(\mathtt{comp}\,(\mathtt{pair}\,(t,\, cl),\, \mathtt{pair}\,(s,\, cl)))\\ \quad \textbf{end}\end{array}.$$

We define the left-to-right call-by-name reduction strategy as follows:

$$\frac{}{\mathtt{mt} : \mathtt{context}} \qquad \frac{cl : \mathtt{closure} \quad c : \mathtt{context}}{\mathtt{ap\_r}\,(cl,\, c) : \mathtt{context}}$$

$$\begin{array}{l}\mathtt{plug}\,(cl,\, c) = \textbf{match}\, c\, \textbf{with}\\ \quad \mid \mathtt{mt} \;\Rightarrow\; cl\\ \quad \mid \mathtt{ap\_r}\,(cl',\, c') \;\Rightarrow\; \mathtt{plug}\,(\mathtt{comp}\,(cl,\, cl'),\, c')\\ \quad \textbf{end}\end{array}$$

The decomposition relation is performed as follows:

$$\frac{\mathtt{decctx}\,(c,\,v,\,d)}{\mathtt{dec}\,(v,\,c,\,d)}\;(\mathtt{val\_ctx}) \qquad \frac{\mathtt{dec}\,(cl_0,\,\mathtt{ap\_r}\,(cl_1,\,c),\,d)}{\mathtt{dec}\,(\mathtt{comp}\,(cl_0,\,cl_1),\,c,\,d)}\;(\mathtt{comp\_ctx})$$

$$\frac{}{\mathtt{dec}\,(\mathtt{pair}\,(n,\,s),\,c,\,\mathtt{d\_red}\,(\mathtt{r\_get}\,(n,\,s),\,c))}\;(\mathtt{var\_ctx})$$

$$\frac{}{\mathtt{dec}\,(\mathtt{pair}\,(\mathtt{app}\,(t_0,\,t_1),\,s),\,c,\,\mathtt{d\_red}\,(\mathtt{r\_app}\,(t_0,\,t_1,\,s),\,c))}\;(\mathtt{app\_ctx})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{mt},\,v,\,\mathtt{d\_val}\,v)}\;(\mathtt{mt\_dec})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{ap\_r}\,(cl,\,c),\,v_1,\,\mathtt{d\_red}\,(\mathtt{r\_beta}\,(v,\,cl),\,c))}\;(\mathtt{ap\_r\_dec})$$

$$\frac{\mathtt{dec}\,((t,\,\mathtt{mt}),\,\mathtt{mt},\,d)}{\mathtt{decmt}\,(t,\,d)}\;(\mathtt{d\_intro})$$

Next, we define the decompose-contract-plug loop just as in the case of the call-by-value λ-calculus described in Section 3.1 except that the role of terms is now played by closures.

All the derivation steps up to the eval/apply machine are performed analogously to the development in Section 3.1.

3.2.2 A push/enter abstract machine

However, in the case of the calculus of closures our goal is to show how we can arrive at the Krivine machine, the canonical abstract machine for the call-by-name evaluation in the λ-calculus, using environments. First of all, the Krivine machine is a push/enter machine which means that it has only one kind of configurations.[7] We obtain a push/enter machine from an eval/apply machine by inlining the definition of $\mathtt{decctx}_2$ in the definition of $\mathtt{dec}_2$, and thus obtaining the following specification:

$$\frac{}{\mathtt{dec}_3\,(v,\,\mathtt{mt},\,\mathtt{d\_val}\,v)}\;(\mathtt{val\_ctx\_mt3}) \qquad \frac{\mathtt{dec}_3\,(cl_0,\,\mathtt{ap\_r}\,(cl_1,\,c),\,d)}{\mathtt{dec}_3\,(\mathtt{comp}\,(cl_0,\,cl_1),\,c,\,d)}\;(\mathtt{comp\_ctx3})$$

$$\frac{\mathtt{dec}_3\,(\mathtt{pair}\,(t,\,cl :: s),\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(\mathtt{lam}\,t,\,s),\,\mathtt{ap\_r}\,(cl,\,c),\,v)}\;(\mathtt{val\_ctx\_ap3})$$

$$\frac{\mathtt{contract}\,((\mathtt{r\_get}\,(n,\,s))) = \mathtt{Some}\,cl \quad \mathtt{dec}_3\,(cl,\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(n,\,s),\,c,\,v)}\;(\mathtt{var\_ctx3})$$

$$\frac{\mathtt{dec}_3\,(\mathtt{comp}\,(\mathtt{pair}\,(t_0,\,s),\,\mathtt{pair}\,(t_1,\,s)),\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(\mathtt{app}\,(t_0,\,t_1),\,s),\,c,\,v)}\;(\mathtt{app\_ctx3})$$

$$\frac{\mathtt{dec}_3\,(\mathtt{pair}\,(t,\,nil),\,\mathtt{mt},\,v)}{\mathtt{eval}_3\,(t,\,v)}\;(\mathtt{e\_intro3})$$

The correctness of this step is stated in the expected way and proved by induction on the derivation decompose2 with the auxiliary property:

[7] In the *push/enter* model the function in a function call is entered and it finds its arguments pushed on the stack.

$$\forall\, c, v, v'.\, \mathtt{decctx_2}\,(c,\, v,\, v') \to \textbf{match}\, c\, \textbf{with} \\ \mid \mathtt{mt} \Rightarrow \mathtt{dec_3}\,(v,\, \mathtt{mt},\, v') \\ \mid \mathtt{ap\_r}\,(cl,\, c) \Rightarrow \mathtt{dec_3}\,(\mathtt{r\_beta}\,(v,\, s),\, c,\, v') \\ \textbf{end}$$

3.2.3 An environment abstract machine

The final two steps are performed to get from a machine operating on closures to a machine operating on terms and substitutions which become environments. To this end, we first observe that if we start evaluating a closure which is a pair of a term and a substitution, then we can bypass the step where the comp constructor is used, i.e., whenever `app_ctx3` is used in the derivation, then the next step will necessary be the application of `comp_ctx3`. Therefore, we can merge the two steps together, and in this way we obtain a relation that only operates on closures formed with the pair constructor. Moreover, all substitutions and contexts occurring in this new relation also contain only the restricted form of closures. This sublanguage coincides with Curien's calculus of closures [8]. We represent it using the datatypes `closureC`, `substitutionC` and `contextC`, together with their injection functions into the unrestricted language. The final relation, obtained by splitting components of the restricted closure is the following:

$$\frac{}{\mathtt{dec_5}\,(\mathtt{lam}\, t,\, s,\, \mathtt{mtC},\, \mathtt{v\_val}\,(t,\, s))}\ (\mathtt{val\_ctx\_mt5})$$

$$\frac{\mathtt{dec_5}\,(t,\, cl :: s,\, c,\, v)}{\mathtt{dec_5}\,(\mathtt{lam}\, t,\, s,\, \mathtt{apC\_r}\,(cl,\, c),\, v)}\ (\mathtt{val\_ctx\_ap5})$$

$$\frac{\mathtt{contract}\,((\mathtt{r\_get}\,(n,\, s))) = \mathtt{Some}\,(\mathtt{pairC}\,(cl,\, s'))\quad \mathtt{dec_5}\,(cl,\, s',\, c,\, v)}{\mathtt{dec_5}\,(n,\, s,\, c,\, v)}\ (\mathtt{var\_ctx5})$$

$$\frac{\mathtt{dec_5}\,(t_0,\, s,\, \mathtt{apC\_r}\,(\mathtt{pairC}\,(t_1,\, s),\, c),\, v)}{\mathtt{dec_5}\,(\mathtt{app}\,(t_0,\, t_1),\, s,\, c,\, v)}\ (\mathtt{app\_ctx5})$$

$$\frac{\mathtt{dec_5}\,(t,\, nil,\, \mathtt{mtC},\, v)}{\mathtt{eval_5}\,(t,\, v)}\ (\mathtt{e\_intro5})$$

The equivalence between the two evaluation relations holds for restricted closures only.

3.2.4 The Krivine abstract machine

Transforming the relation from Section 3.2.3 into the usual transition systems yields the following result, the canonical Krivine machine:

$$\frac{t : \mathtt{term}}{\mathtt{c\_init}\, t : \mathtt{configuration}} \qquad \frac{t : \mathtt{term}\quad s : \mathtt{substitutionC}\quad c : \mathtt{contextC}}{\mathtt{c\_eval}\,(t,\, s,\, c) : \mathtt{configuration}}$$

$$\frac{v : \mathtt{value}}{\mathtt{c\_final}\, v : \mathtt{configuration}}$$

$$\begin{aligned}
\texttt{c_init}\,t &\rhd \texttt{c_eval}\,(t,\,\texttt{nil},\,\texttt{mtC})\\
\texttt{c_eval}\,(\texttt{lam}\,t,\,s,\,\texttt{mtC}) &\rhd \texttt{c_final}\,(\texttt{v_val}\,(t,\,s))\\
\texttt{c_eval}\,(\texttt{lam}\,t,\,s,\,\texttt{apC_r}\,(cl,\,c)) &\rhd \texttt{c_eval}\,(t,\,cl :: s,\,c)\\
\texttt{c_eval}\,(\texttt{var}\,i,\,s,\,c) &\rhd \texttt{c_eval}\,(t,\,s',\,c)\ \text{ if } \texttt{nth}\,s\,i = \texttt{Some}\,(\texttt{pairC}\,(t,\,s'))\\
\texttt{c_eval}\,(\texttt{app}\,(t_0,\,t_1),\,s,\,c) &\rhd \texttt{c_eval}\,(t_0,\,s,\,\texttt{apC_r}\,(\texttt{pairC}\,(t_1,\,s),\,c))
\end{aligned}$$

$$\frac{\texttt{c\_init}\,t \rhd^* \texttt{c\_final}\,v}{\texttt{eval}_6\,(t,\,v)}\ \texttt{e\_intro}_6$$

4 Towards a generic framework

The refocusing method allows one to split the process of writing an abstract machine into smaller, simpler stages and intuitive proofs of their equivalence, as outlined in previous sections. These intermediate proofs can be automated: we can write Coq tactics for proving the equivalence of the pre-abstract machine with the staged abstract machine, etc., as long as these machines are specified in the way described in Section 3.1. Furthermore, if the user can provide a decomposition function together with the proof of the Property 2.1 for this function, we can automate the proof of unique decomposition for the specified reduction semantics. It seems plausible to make the development modular, with the use of Coq module types for describing signatures of reduction semantics and the intermediate machines.

In particular, we can imagine the following signature of a reduction semantics:

Parameters `term value redex context` : **Set**.
Parameter `plug` : `context` → `term` → `term`.

Then we can define an eval/apply abstract machine realizing the reduction strategy specified by this semantics as a transition system, i.e., the transitive closure of a transition function:

Parameter `configuration` : **Set**.
Parameter `transition` : `configuration` → `configuration` → **Prop**.
Parameter `transitive_closure` : `configuration` → `configuration` → **Prop**.

and such that the following properties hold:

Axiom `search_redex` :
$\forall\, t, c, c', r.\, \texttt{plug}\,(t,\,c) = \texttt{plug}\,(r,\,c') \rightarrow \texttt{c\_eval}\,(t,\,c) \rhd^* \texttt{c\_eval}\,(r,\,c')$.
Axiom `search_value` :
$\forall\, t, c, v.\, \texttt{plug}\,(t,\,c) = \texttt{plug}\,(v,\,\texttt{mt}) \rightarrow \texttt{c\_eval}\,(t,\,c) \rhd^* \texttt{c\_final}\,v$.
Axiom `decompose_redex` :
$\forall\, r, t, c.\, \texttt{contract}\,(r) = \texttt{Some}\,t \rightarrow \texttt{c\_eval}\,(r,\,c) \rhd^* \texttt{c\_eval}\,(t,\,c)$.

These properties characterize the correspondence between a reduction semantics and an abstract machine extensionally. If we can prove that these properties hold (regardless of the particular implementation of the decomposing function, then we ensure that the abstract machine–represented by transitive_closure–realizes the reduction strategy of the given reduction semantics. This issue has been studied also by Hardin et al. [12] who considered the converse direction and identified the reduction strategies "wired" in several virtual machines for the λ-calculus, by establishing

a bisimulation between the machines and the underlying reduction semantics (of λ-calculi with explicit substitutions), and thus justifying each machine transition with respect to the underlying reduction semantics. It would also be interesting to connect our extensional approach to reduction strategies with the intentional, transition-system-based approach.

Looking at other examples of refocusing, we are able to identify tactics also for closure unfolding (leading to environment machines) and store unfolding (leading to store machines), and to extend the method to work for context-sensitive reduction semantics where contraction rules take into account not only the redex but also its surrounding context (as found, e.g., in languages with control operators).

Taking this idea further, we would like to be able to automatically generate the abstract machine corresponding to a given reduction semantics, together with the proof of its correctness. To this end, we need to develop a meta-language for representing such abstract machines and manipulate them as Coq objects.

5 Conclusion

We have described a formalization in Coq of the refocusing method applied to two prototypical programming languages: the call-by-value λ-calculus, and the call-by-name calculus of closures. These case studies are to serve as basis for a general framework for representing programming languages and their operational semantics in Coq, and to automate their interderivations, while ensuring their correctness.

A generalized refocusing method has been shown to apply to languages with a non-standard (context-sensitive) notion of reduction, e.g., languages with control operators, state, lazy evaluation, etc., as studied by Biernacka and Danvy [6]. Such languages are in common practical use, therefore our intended framework seems of significant practical importance.

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] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[4] 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. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[5] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the research report BRICS RS-06-3.

[6] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375:76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[7] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*, 2006. http://coq.inria.fr.

[8] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[9] Olivier Danvy. From reduction-based to reduction-free normalization. Research Report BRICS RS-04-30, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2004. Invited talk at the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004), Aachen, Germany, June 2, 2004. To appear in ENTCS.

[10] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[11] 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.

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

[13] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.