BRICS RS-05-15 Biernacka & Danvy: A Concrete Framework for Environment Machines

BRICS

Basic Research in Computer Science

A Concrete Framework for Environment Machines

Małgorzata Biernacka
Olivier Danvy

BRICS Report Series **RS-05-15**

ISSN 0909-0878 **May 2005**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/05/15/`

A concrete framework for environment machines

Małgorzata Biernacka and Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

May 13, 2005

Abstract

We materialize the common belief that calculi with explicit substitutions provide an intermediate step between an abstract specification of substitution in the λ-calculus and its concrete implementations. To this end, we go back to Curien's original calculus of closures (an early calculus with explicit substitutions), we extend it minimally so that it can also express one-step reduction strategies, and we methodically derive a series of environment machines from the specification of two one-step reduction strategies for the λ-calculus: normal order and applicative order. The derivation extends Danvy and Nielsen's refocusing-based construction of abstract machines with two new steps: one for coalescing two successive transitions into one, and the other for unfolding a closure into a term and an environment in the resulting abstract machine. The resulting environment machines include both the idealized and the original versions of Krivine's machine, Felleisen et al.'s CEK machine, and Leroy's Zinc abstract machine.

*Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: {`mbiernac,danvy`}`@brics.dk`

Contents

1 Introduction

Krivine's machine and Felleisen et al.'s CEK machine are probably the simplest examples of abstract machines implementing an evaluation function of the λ-calculus [15,22, 27]. Like many other abstract machines for languages with binding constructs, they are environment-based, i.e., roughly, one component of each machine configuration stores terms that are substituted for free variables during the process of evaluation. The transitions of each machine provide a precise way of handling substitution. This precision contrasts with the traditional presentations of the λ-calculus [9, page 9] [7, page 51] where the β-rule is expressed using a meta-level notion of substitution:

$$(\lambda x.t_0)\, t_1 \rightarrow t_0\{t_1/x\}$$

On the right-hand side of this rule, all the free occurrences of x in t_0 are simultaneously replaced by t_1 (which may require auxiliary α-conversions to avoid variable capture). Most implementations, however, do not implement the β-rule using actual substitutions. Instead, they keep an explicit representation of what should have been substituted and leave the term untouched. This environment technique is due to Hasenjaeger in logic [38, § 54] and to Landin in computer science [29]. In logic, it makes it possible to reason by structural induction over λ-terms (since they do not change across β-reduction), and in computer science, it makes it possible to compile λ-terms (since they do not change at run time).

To bridge the two worlds of actual substitutions and explicit representations of what should have been substituted, various calculi of 'explicit substitutions' have been proposed [1,2,12,13,26,31,36,37]. The idea behind explicit substitutions is to incorporate the notion of substitution into the syntax of the language and then specify suitable rewrite rules that realize it.

In these calculi, the syntax is extended with a 'closure' (the word is due to Landin [29]), i.e., a term together with its environment—hereby referred to as 'substitution' to follow tradition [1, 12]. Moreover, variables are often represented with de Bruijn indices [19] (i.e., lexical offsets in compiler parlance [34]) rather than explicit names; this way, substitutions can be conveniently regarded as lists and the position of a closure in such a list indicates for which variable this closure is to be substituted.

Thus, in a calculus of explicit substitutions, β-reduction is specified using one rule for extending the substitution with a closure to be substituted, such as

$$((\lambda t)[s])\; c \rightarrow t[c \cdot s],$$

where c is prepended to the list s, and another rule that replaces a variable with the corresponding closure taken from the substitution, such as

$$i[s] \rightarrow c,$$

where c is the ith element of the list s.

Calculi of explicit substitutions come in two flavors: weak calculi, typically used to express weak normalization (evaluation); and strong calculi, that are expressive enough to allow strong normalization. The greater power of strong calculi comes from a richer set of syntactic constructs forming substitutions (for instance, substitutions can be composed, and indices can be lifted).

This work: We present a completely systematic way of deriving an environment machine from a specification of one-step reduction strategy in a weak calculus of closures, by employing Danvy and Nielsen's refocusing technique [18] followed by the fusion of two steps into one and an unfolding of the data type of closures.

We first consider Curien's original calculus of closures $\lambda\rho$ [12]. Curien argues that his calculus mediates between the standard λ-calculus and its implementations via abstract machines. He illustrates his argument by constructing Krivine's machine from a multi-step normal-order reduction strategy.

We observe, however, that one-step reductions cannot be expressed in $\lambda\rho$ and therefore we propose a minimal extension to make it capable of expressing such computations (we dub it the $\lambda\widehat{\rho}$-calculus). We then present a detailed derivation of the usual idealized version of Krivine's machine [10, 12, 24] from the specification of the normal-order one-step strategy in $\lambda\widehat{\rho}$, and we outline the derivation of its applicative-order analog, the CEK machine [22]. We also outline the derivation of the original version of Krivine's machine [28] and of its applicative-order analog, which we identify as Leroy's Zinc abstract machine [30].

Prerequisites and notation: We assume a basic familiarity with the λ-calculus, explicit substitutions, and abstract machines [12]. We also follow standard usage and overload the word "closure" (as in: a term together with a substitution, a reflexive and transitive closure, a compatible closure, and also a closed term) and the word "step" (as in: a derivation step, a decomposition step, and one-step reduction).

Overview: We first present a minimal extension to Curien's original calculus of closures that is expressive enough to account for one-step reduction (Section 2). We then methodically derive environment machines from a series of reduction semantics à la Felleisen (Sections 3 to 6) before drawing a bigger picture (Section 7).

2 One-step reduction in a calculus of closures

In this section we first briefly review Curien's original calculus of closures $\lambda\rho$ [12], and then present an extension of $\lambda\rho$ that facilitates the specification of one-step reduction strategies. We illustrate the power of the extended calculus with the standard definitions of normal-order and applicative-order strategies.

As a reference point, we first specify the one-step and multi-step reduction relations in the standard λ-calculus.

Reduction in the λ-calculus: Let us recall the formulation of the λ-calculus with de Bruijn indices. Terms are built according to the following grammar:

$$t ::= i \mid t\,t \mid \lambda t,$$

where i ranges over natural numbers (greater than 0).

In the λ-calculus, one-step reduction is defined as the compatible closure of the notion of reduction given by the β-rule [7, Section 3.1]:

$$(\beta) \qquad (\lambda t_0)\, t_1 \to t_0\{t_1/1\} \qquad\qquad (\mu) \quad \frac{t_1 \to t_1'}{t_0\, t_1 \to t_0\, t_1'}$$

$$(\nu) \qquad \frac{t_0 \to t_0'}{t_0\, t_1 \to t_0'\, t_1} \qquad\qquad (\xi) \quad \frac{t \to t'}{\lambda t \to \lambda t'}$$

where in (β), $t_0\{t_1/1\}$ is a meta-level substitution operation (with suitable reindexing of variables).

Weak (nondeterministic) subsystems are obtained by discarding the ξ-rule. The usual deterministic strategies are obtained as follows:

- The normal-order strategy is obtained by a further restriction, disallowing also the right-compatibility rule (μ). In effect, one obtains the following normal-order one-step reduction strategy for the λ-calculus, producing weak head normal forms:

$$
\begin{array}{ll}
(\beta) & (\lambda t_0)\, t_1 \to_{\mathrm{n}} t_0\{t_1/1\} \\[1ex]
(\nu) & \dfrac{t_0 \to_{\mathrm{n}} t_0'}{t_0\, t_1 \to_{\mathrm{n}} t_0'\, t_1}
\end{array}
$$

  Alternatively, the normal-order strategy can be expressed by the following rule:

$$
\frac{t_0 \to_{\mathrm{n}}^{*} \lambda t_0'}{t_0\, t_1 \to_{\mathrm{n}} t_0'\{t_1/1\}}
$$

  A rule of this form specifies a multi-step reduction strategy (witness the reflexive, transitive closure used in the premise).

- The applicative-order strategy is obtained by another restriction on the β-rule:

$$
(\beta_{\mathrm{v}}) \qquad (\lambda t_0)\, t_1 \to_{\mathrm{v}} t_0\{t_1/1\} \quad \text{if } t_1 \text{ is a value}
$$

  Values are variables (de Bruijn indices) and λ-abstractions.

  The following restriction on the right-compatibility rule (μ) makes the reduction strategy deterministically proceed from left to right:

$$
\begin{array}{lll}
(\nu) & \dfrac{t_0 \to_{\mathrm{v}} t_0'}{t_0\, t_1 \to_{\mathrm{v}} t_0'\, t_1} & \\[2ex]
(\mu) & \dfrac{t_1 \to_{\mathrm{v}} t_1'}{t_0\, t_1 \to_{\mathrm{v}} t_0\, t_1'} & \text{if } t_0 \text{ is a value}
\end{array}
$$

  The following restriction on the left-compatibility rule (ν) makes the reduction strategy deterministically proceed from right to left:

$$
\begin{array}{lll}
(\nu) & \dfrac{t_0 \to_{\mathrm{v}} t_0'}{t_0\, t_1 \to_{\mathrm{v}} t_0'\, t_1} & \text{if } t_1 \text{ is a value} \\[2ex]
(\mu) & \dfrac{t_1 \to_{\mathrm{v}} t_1'}{t_0\, t_1 \to_{\mathrm{v}} t_0\, t_1'} &
\end{array}
$$

2.1 Curien's calculus of closures

The language of $\lambda\rho$ [12] has three syntactic categories: terms, closures and substitutions:

$$
\begin{array}{ll}
(\mathsf{Term}) & t ::= i \mid t\, t \mid \lambda t \\
(\mathsf{Closure}) & c ::= t[s] \\
(\mathsf{Substitution}) & s ::= \bullet \mid c \cdot s
\end{array}
$$

Terms are defined as in the λ-calculus with de Bruijn indices. A $\lambda\rho$-closure is a pair consisting of a λ-term and a $\lambda\rho$-substitution, which itself is a finite list of $\lambda\rho$-closures to be substituted for free variables in the λ-term. We abbreviate $c_1 \cdot (c_2 \cdot (c_3 \cdot \ldots (c_m \cdot \bullet) \ldots))$ as $c_1 \cdots c_m$.

The weak reduction relation $\xrightarrow{\rho}$ is specified by the following rules:

$$\text{(Eval)} \quad \frac{t_0[s] \xrightarrow{\rho}{}^{*} (\lambda t'_0)[s']}{(t_0\, t_1)[s] \xrightarrow{\rho} t'_0[(t_1[s]) \cdot s']}$$

$$\text{(Var)} \quad i[c_1 \cdots c_m] \xrightarrow{\rho} c_i \ \text{ if } i \leq m$$

$$\text{(Sub)} \quad \frac{c_1 \xrightarrow{\rho}{}^{*} c'_1 \quad \ldots \quad c_m \xrightarrow{\rho}{}^{*} c'_m}{t[c_1 \cdots c_m] \xrightarrow{\rho} t[c'_1 \cdots c'_m]}$$

where $\xrightarrow{\rho}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{\rho}$. Reductions are performed on closures, and not on individual terms. Although minimalist (it is not possible to "push" a substitution inside a λ-abstraction), this calculus is powerful enough to compute weak head normal forms. The grammar of weak head normal forms is as follows:

$$t_{\mathrm{nf}} ::= (\lambda t)[s] \ \mid \ (\ldots((i[s])\, c_1) \ldots)\, c_m$$

where $i[s]$ is irreducible, which happens if and only if i is greater than the length of s. If we restrict ourselves to considering only closed terms, then the only weak head normal form is a closure whose term is an abstraction.

The rules of the calculus are nondeterministic and can be restricted to define a specific deterministic reduction strategy. For instance, the normal-order strategy (denoted $\xrightarrow{\rho}_{\mathrm{n}}$) is obtained by restricting the rules to (Eval) and (Var). This restriction specifies a multi-step reduction strategy because of the transitive closure used in the (Eval) rule.

2.2 A minimal extension to Curien's calculus of closures

The $\lambda\rho$-calculus is not expressive enough to specify one-step reduction because the specification of one-step reduction requires a way to "compose" intermediate results of computation—here, closures—to form a new closure that can be reduced further. In $\lambda\rho$, there is no such possibility. A simple solution is to extend the syntax of closures with a construct denoting closure composition. We denote it simply by juxtaposition:

(Term)	$t ::= i \mid t\, t \mid \lambda t$
(Closure)	$c ::= t[s] \mid c\, c$
(Substitution)	$s ::= \bullet \mid c \cdot s$

With the extended syntax we are now in position to define the one-step reduction relation as the compatible closure of the notion of (a closure-based variant of) β-reduction:

$$(\beta) \quad ((\lambda t)[s])\, c \xrightarrow{\widehat{\rho}} t[c \cdot s] \qquad \text{(Var)} \quad i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}} c_i \text{ if } i \leq m$$

$$(\nu) \quad \frac{c_0 \xrightarrow{\widehat{\rho}} c'_0}{c_0\, c_1 \xrightarrow{\widehat{\rho}} c'_0\, c_1} \qquad \text{(App)} \quad (t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\, (t_1[s])$$

$$(\mu) \quad \frac{c_1 \xrightarrow{\widehat{\rho}} c'_1}{c_0\, c_1 \xrightarrow{\widehat{\rho}} c_0\, c'_1} \qquad \text{(Sub)} \quad \frac{c_i \xrightarrow{\widehat{\rho}} c'_i}{t[c_1 \cdots c_i \cdots c_m] \xrightarrow{\widehat{\rho}} t[c_1 \cdots c'_i \cdots c_m]} \ \text{for } i \leq m$$

The $\lambda\widehat{\rho}$-calculus is nondeterministic and confluent. The following proposition formalizes the simulation of $\lambda\rho$ reductions in $\lambda\widehat{\rho}$ and vice versa.

Proposition 1 (Simulation). *Let c_0 and c_1 be $\lambda\rho$-closures. Then the following properties hold:*

1. *If $c_0 \xrightarrow{\rho} c_1$, then $c_0 \xrightarrow{\widehat{\rho}}{}^{+} c_1$.*
2. *If $c_0 \xrightarrow{\widehat{\rho}}{}^{*} c_1$, then $c_0 \xrightarrow{\rho}{}^{*} c_1$.*

Proof. The proofs are done by induction; we show only the interesting cases.

Proof of *1*. By induction on the derivation of $c_0 \xrightarrow{\rho} c_1$, with a subinduction on the length of the reduction sequence in the following auxiliary property:

$$\text{If } c'_0 \xrightarrow{\rho}{}^{*} c'_1, \text{ then } c'_0 \xrightarrow{\widehat{\rho}}{}^{*} c'_1,$$

where $c'_0 \xrightarrow{\rho}{}^{*} c'_1$ is a subderivation of $c_0 \xrightarrow{\rho} c_1$.

Case $(t_0\, t_1)[s] \xrightarrow{\rho} t'_0[(t_1[s]) \cdot s']$. We then know that $t_0[s] \xrightarrow{\rho}{}^{n} (\lambda t'_0)[s']$ for some $n \geq 0$. By induction on n we prove that $t_0[s] \xrightarrow{\widehat{\rho}}{}^{*} (\lambda t'_0)[s']$:

Case $n = 0$. Trivial.

Case $n = k + 1$. Assume $t_0[s] \xrightarrow{\rho} c'_0 \xrightarrow{\rho}{}^{k} (\lambda t'_0)[s']$. Applying the outer induction hypothesis to $t_0[s] \xrightarrow{\rho} c'_0$, we obtain $t_0[s] \xrightarrow{\widehat{\rho}}{}^{+} c'_0$, and then applying the inner induction hypothesis to $c'_0 \xrightarrow{\rho}{}^{k} (\lambda t'_0)[s']$ yields the desired property.

Hence we obtain the following reduction sequence in $\lambda\widehat{\rho}$:

$$(t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\, (t_1[s]) \xrightarrow{\widehat{\rho}}{}^{*} ((\lambda t'_0)[s'])\, (t_1[s]) \xrightarrow{\widehat{\rho}} t'_0[(t_1[s]) \cdot s'].$$

Proof of *2*. By induction on the length of the reduction sequence $c_0 \xrightarrow{\widehat{\rho}}{}^{*} c_1$. For the inductive case, assume $c_0 \xrightarrow{\widehat{\rho}}{}^{n+1} c_1$, i.e., $c_0 \xrightarrow{\widehat{\rho}} c'_0 \xrightarrow{\widehat{\rho}}{}^{n} c_1$. We distinguish two cases:

- $c'_0 = t[s]$: In this case, the first reduction step in $\lambda\widehat{\rho}$ is either according to the rule (Var), or to the rule (Sub); thus it can be simulated directly by the corresponding rule in $\lambda\rho$.
- $c'_0 = c\, c$: The only applicable reduction is (App), hence $c_0 = (t_0\, t_1)[s]$ and $c'_0 = (t_0[s])\, (t_1[s])$. Analyzing the reduction rules, we observe that in the subsequent reduction sequence the rule (β) must be applied, since c_1 is again a $\lambda\rho$-closure. Without loss of generality we can assume that

$$c'_0 \xrightarrow{\widehat{\rho}}{}^{k_1} ((\lambda t'_0)[s'])\, (t_1[s]) \xrightarrow{\widehat{\rho}}{}^{k_2} ((\lambda t'_0)[s'])\, (t'_1[s'']) \xrightarrow{\widehat{\rho}} t'_0[(t'_1[s'']) \cdot s'] \xrightarrow{\widehat{\rho}}{}^{k_3} c_1$$

for $k_1 + k_2 + 1 + k_3 = n$. By induction hypothesis $t_0[s] \xrightarrow{\rho}{}^{*} (\lambda t'_0)[s']$ and $t_1[s] \xrightarrow{\rho}{}^{*} t'_1[s'']$, and hence $c'_0 \xrightarrow{\rho} t'_0[(t_1[s]) \cdot s'] \xrightarrow{\rho}{}^{*} t'_0[(t'_1[s'']) \cdot s']$. Finally, $t'_0[(t'_1[s'']) \cdot s'] \xrightarrow{\rho}{}^{*} c_1$ by induction hypothesis.

$\square$

2.3 Specification of the normal-order reduction strategy

The normal-order strategy is obtained by restricting $\lambda\widehat{\rho}$ to the following rules:

$$(\beta) \quad ((\lambda t)[s])\; c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} t[c \cdot s] \qquad\qquad (\mathrm{Var}) \quad i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_i \quad \text{if } i \leq m$$

$$(\nu) \quad \frac{c_0 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_0'}{c_0\; c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_0'\; c_1} \qquad\qquad (\mathrm{App}) \quad (t_0\; t_1)[s] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[s])\; (t_1[s])$$

We consider only closed terms, and hence we omit the side condition on i in Sections 3 and 5.1.

Let $\xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*}$ (the call-by-name evaluation relation) denote the reflexive, transitive closure of $\xrightarrow{\widehat{\rho}}_{\mathrm{n}}$. The grammar of values and substitutions for call-by-name evaluation reads as follows:

$$\begin{array}{ll} (\mathsf{Value}) & v ::= (\lambda t)[s] \\ (\mathsf{Substitution}) & s ::= \bullet \mid c \cdot s \end{array}$$

2.4 Specification of the applicative-order reduction strategy

Similarly, the applicative-order strategy is obtained by restricting $\lambda\widehat{\rho}$ to the following rules:

$$(\beta) \quad ((\lambda t)[s])\; c \xrightarrow{\widehat{\rho}}_{\mathrm{v}} t[c \cdot s] \;\; \text{if } c \text{ is a value} \qquad (\mathrm{Var}) \quad i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{v}} c_i \quad \text{if } i \leq m$$

$$(\nu) \quad \frac{c_0 \xrightarrow{\widehat{\rho}}_{\mathrm{v}} c_0'}{c_0\; c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{v}} c_0'\; c_1} \qquad\qquad (\mathrm{App}) \quad (t_0\; t_1)[s] \xrightarrow{\widehat{\rho}}_{\mathrm{v}} (t_0[s])\; (t_1[s])$$

$$(\mu) \quad \frac{c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{v}} c_1'}{c_0\; c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{v}} c_0\; c_1'} \quad \text{if } c_0 \text{ is a value}$$

We consider only closed terms, and hence we omit the side condition on i in Sections 4 and 5.2.

Let $\xrightarrow{\widehat{\rho}}_{\mathrm{v}}{}^{*}$ (the call-by-value evaluation relation) denote the reflexive, transitive closure of $\xrightarrow{\widehat{\rho}}_{\mathrm{v}}$. The grammar of values and substitutions for call-by-value evaluation reads as follows:

$$\begin{array}{ll} (\mathsf{Value}) & v ::= (\lambda t)[s] \\ (\mathsf{Substitution}) & s ::= \bullet \mid v \cdot s \end{array}$$

Under call by value, both sub-components of any composition $c_0\; c_1$ (i.e., both c_0 and c_1) are evaluated. We consider left-to-right evaluation (i.e., c_0 is evaluated, and then c_1) in Section 4 and right-to-left evaluation in Section 5.2.

3 From normal-order reduction to call-by-name environment machine

We present a detailed and systematic derivation of an abstract machine for call-by-name evaluation in the λ-calculus, starting from the specification of the normal-order reduction strategy in the $\lambda\widehat{\rho}$-calculus. We first follow the steps outlined by Danvy and Nielsen in their work on refocusing [18]:

Section 3.1: We specify the normal-order reduction strategy in the form of a reduction semantics, i.e., with a one-step reduction function specified as decomposing a non-value term into a reduction context and a redex, contracting this redex, and plugging the contractum into the context. As is traditional, we also specify evaluation as the transitive closure of one-step reduction.

Section 3.2: We replace the combination of plugging and decomposition by a refocus function that iteratively goes from redex site to redex site in the reduction sequence. The resulting 'refocused' evaluation function is the transitive closure of the refocus function and takes the form of a 'pre-abstract machine.'

Section 3.3: We merge the definitions of the transitive closure and the refocus function into a 'staged abstract machine' that implements the reduction rules and the compatibility rules of the $\lambda\widehat{\rho}$-calculus with two separate functions.

Section 3.4: We inline the function implementing the reduction rules. The result is an eval/apply abstract machine consisting of an 'eval' transition function dispatching on closures and an 'apply' transition function dispatching on contexts.

Section 3.5: We inline the apply transition function. The result is a push/enter abstract machine.

We then simplify and transform the push/enter abstract machine:

Section 3.6: Observing that in a reduction sequence, an (App) reduction step is always followed by a decomposition step, we shortcut these two steps into one decomposition step. This simplification enables the following step.

Section 3.7: We unfold the data type of closures, making the simplified machine operate over two components—a term and a substitution—instead of over one—a closure. The substitution component is the traditional environment of environment machines, and the resulting machine is an environment machine. This machine coincides with the usual idealized version of Krivine's machine [10, 12, 24]. (The original version of Krivine's machine [27, 28] is a bit more complicated, and we treat it in Section 5.)

In Section 3.8, we state the correctness of the idealized version of Krivine's machine with respect to evaluation in the $\lambda\widehat{\rho}$-calculus, and in Section 3.9 we get back to the λ-calculus.

3.1 A reduction semantics for normal order

A reduction semantics [20, 21] consists of the grammar of a source language, a syntactic notion of value, a collection of reduction rules, and a reduction strategy. This reduction strategy is embodied in a grammar of reduction contexts (terms with one hole), a plug function mapping a term and a context into a new term, and a strategy for decomposing a non-value term into a redex and its reduction context. When the redexes do not overlap, the reduction rules are implemented by a contraction function that maps a redex into the corresponding contractum. When the source language satisfies a unique-decomposition property with respect to the reduction strategy, decomposing a non-value term into a redex and its reduction context is implemented by a decomposition function.

A reduction semantics for normal-order reduction in the $\lambda\widehat{\rho}$-calculus can be obtained from the specification of normal-order reduction strategy in Section 2.3 as follows: the syntactic notion of value and the collection of reduction rules are specified directly, and the compatibility rule (ν) induces the grammar of reduction contexts:[1]

$$C ::= [\,] \mid \mathsf{ARG}(c,\ C)$$

The redexes do not overlap and the source language satisfies a unique-decomposition property. Therefore we can define the following three functions:

$$\begin{aligned} \mathsf{contract} &: \mathsf{Redex} \rightarrow \mathsf{Closure} \\ \mathsf{decompose} &: \mathsf{Closure} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\ \mathsf{plug} &: \mathsf{Closure} \times \mathsf{Context} \rightarrow \mathsf{Closure} \end{aligned}$$

Given these three functions, we can define the following one-step reduction function that tests whether a closure is a value or can be decomposed into a redex and a context, that contracts this redex, and that plugs the contractum in the context:

$$\begin{aligned} \mathsf{reduce} &: \mathsf{Closure} \rightarrow \mathsf{Closure} \\ \mathsf{reduce}\, c &= \mathit{case}\ \mathsf{decompose}\, c \\ &\quad\ \mathit{of}\ v \Rightarrow v \\ &\quad\ \mid (r,\ C) \Rightarrow \mathsf{plug}\,(c, C) \qquad \text{where } c = \mathsf{contract}\, r \end{aligned}$$

The following proposition is a consequence of the unique-decomposition property.

Proposition 2. *For any non-value closure c and for any closure c', $c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c' \Leftrightarrow$ $\mathsf{reduce}\, c = c'$.*

Finally, we can define evaluation as the transitive closure of one-step reduction:

$$\begin{aligned} \mathsf{iterate} &: \mathsf{Closure} \rightarrow \mathsf{Value} \\ \mathsf{iterate}\, c &= c & &\text{if } c \text{ is a value} \\ \mathsf{iterate}\, c &= \mathsf{iterate}\,(\mathsf{reduce}\, c) & &\text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathsf{evaluate} &: \mathsf{Term} \rightarrow \mathsf{Value} \\ \mathsf{evaluate}\, t &= \mathsf{iterate}\,(t[\bullet]) \end{aligned}$$

This evaluation function is partial because a reduction sequence might not terminate.

Proposition 3. *For any closed term t and any value v, $t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*}\, v \Leftrightarrow \mathsf{evaluate}\, t = v$.*

For simplicity, we inline the definition of reduce and we use decompose to test whether a value has been reached:

$$\begin{aligned} \mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \rightarrow \mathsf{Value} \\ \mathsf{iterate}\, v &= v \\ \mathsf{iterate}\,(r,\ C) &= \mathsf{iterate}\,(\mathsf{decompose}\,(\mathsf{plug}\,(c, C))) \qquad \text{where } c = \mathsf{contract}\, r \end{aligned}$$

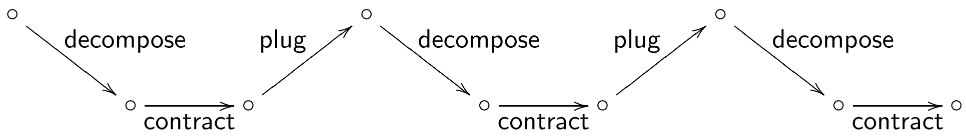
$$\begin{aligned} \mathsf{evaluate} &: \mathsf{Term} \rightarrow \mathsf{Value} \\ \mathsf{evaluate}\, t &= \mathsf{iterate}\,(\mathsf{decompose}\,(t[\bullet])) \end{aligned}$$

[1] In the standard inside-out notation, this grammar can also be stated as follows:

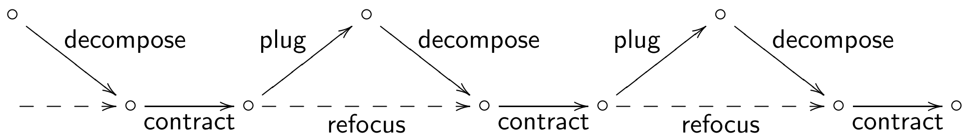
$$C ::= [\,] \mid C[[\,]c]$$

3.2 A pre-abstract machine

The reduction sequence implemented by evaluation can be depicted as follows:



In earlier work [18], Danvy and Nielsen stated formal conditions under which the composition of plug and $\mathsf{decompose}$ could be replaced by a more efficient function, $\mathsf{refocus}$, that would directly go from redex site to redex site in the reduction sequence, and they presented an algorithm for constructing such a $\mathsf{refocus}$ function:



The formal conditions are satisfied here: the unique-decomposition property holds and the grammar of reduction contexts is context free. The algorithm yields the following definition, which takes the form of two state-transition functions, i.e., of an abstract machine:

$$
\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}\ (i[s],\ C) &= (i[s],\ C) \\
\mathsf{refocus}\ ((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\ (C,\ (\lambda t)[s]) \\
\mathsf{refocus}\ ((t_0\ t_1)[s],\ C) &= ((t_0\ t_1)[s],\ C) \\
\mathsf{refocus}\ (c_0\ c_1,\ C) &= \mathsf{refocus}\ (c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus_{aux}}\ ([\,],\ v) &= v \\
\mathsf{refocus_{aux}}\ (\mathsf{ARG}(c,\ C),\ v) &= (v\ c,\ C)
\end{aligned}
$$

In this abstract machine, the configurations are pairs of a closure and a context; the final transitions are specified by $\mathsf{refocus_{aux}}$ and by the first and third clauses of $\mathsf{refocus}$; and the initial transition is specified by two clauses of the corresponding 'refocused' evaluation function, which reads as follows:

$$
\begin{aligned}
\mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \to \mathsf{Value} \\
\mathsf{iterate}\ v &= v \\
\mathsf{iterate}\ (r,\ C) &= \mathsf{iterate}\ (\mathsf{refocus}\ (c,\ C)) \quad \text{where } c = \mathsf{contract}\ r
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{evaluate} &: \mathsf{Term} \to \mathsf{Value} \\
\mathsf{evaluate}\ t &= \mathsf{iterate}\ (\mathsf{refocus}\ (t[\bullet],\ [\,]))
\end{aligned}
$$

(For the initial call to $\mathsf{iterate}$, we have exploited the double equality $\mathsf{decompose}\ (t[\bullet]) = \mathsf{decompose}\ (\mathsf{plug}\ (t[\bullet], [\,])) = \mathsf{refocus}\ (t[\bullet],\ [\,])$.)

Through the auxiliary function $\mathsf{iterate}$, this evaluation function computes the transitive closure of $\mathsf{refocus}$. Following Danvy and Nielsen, we refer to it as a 'pre-abstract machine.'

3.3 A staged abstract machine

To transform the pre-abstract machine into an abstract machine, we distribute the calls to $\mathsf{iterate}$ from the definitions of $\mathsf{evaluate}$ and of $\mathsf{iterate}$ to the definitions of $\mathsf{refocus}$ and $\mathsf{refocus_{aux}}$:

$$
\begin{aligned}
\mathsf{evaluate} &: \mathsf{Term} \rightarrow \mathsf{Value} \\
\mathsf{evaluate}\, t &= \mathsf{refocus}\,(t[\bullet],\ [\,])
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \rightarrow \mathsf{Value} \\
\mathsf{iterate}\, v &= v \\
\mathsf{iterate}\,(r,\ C) &= \mathsf{refocus}\,(c,\ C) \quad \text{where } c = \mathsf{contract}\, r
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}\,(i[s],\ C) &= \mathsf{iterate}\,(i[s],\ C) \\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s]) \\
\mathsf{refocus}\,((t_0\ t_1)[s],\ C) &= \mathsf{iterate}\,((t_0\ t_1)[s],\ C) \\
\mathsf{refocus}\,(c_0\ c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus_{aux}}\,([\,],\ v) &= \mathsf{iterate}\, v \\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ v) &= \mathsf{iterate}\,(v\ c,\ C)
\end{aligned}
$$

The resulting definitions of $\mathsf{evaluate}$, $\mathsf{iterate}$, $\mathsf{refocus}$, and $\mathsf{refocus_{aux}}$ are that of four mutually recursive transition functions that form an abstract machine. In this abstract machine, the configurations are pairs of a closure and a context, the initial transition is specified by $\mathsf{evaluate}$, and the final transition in the first clause of $\mathsf{iterate}$. The compatibility rules are implemented by $\mathsf{refocus}$ and $\mathsf{refocus_{aux}}$, and the reduction rules by the call to $\mathsf{contract}$ in the second clause of $\mathsf{iterate}$. We can make this last point even more manifest by inlining $\mathsf{contract}$ in the definition of $\mathsf{iterate}$:

$$
\begin{aligned}
\mathsf{iterate}\, v &= v \\
\mathsf{iterate}\,(i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\,(c_i,\ C) \\
\mathsf{iterate}\,((t_0\ t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\ (t_1[s]),\ C) \\
\mathsf{iterate}\,(((\lambda t)[s])\ c,\ C) &= \mathsf{refocus}\,(t[c \cdot s],\ C)
\end{aligned}
$$

By construction, the machine therefore separately implements the reduction rules and the compatibility rules; for this reason, we refer to it as a 'staged abstract machine.'

3.4 An eval/apply abstract machine

We now inline the calls to $\mathsf{iterate}$ in the staged abstract machine. The resulting machine reads as follows:

$$
\mathsf{evaluate}\, t = \mathsf{refocus}\,(t[\bullet],\ [\,])
$$

$$
\begin{aligned}
\mathsf{refocus}\,(i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\,(c_i,\ C) \\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s]) \\
\mathsf{refocus}\,((t_0\ t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\ (t_1[s]),\ C) \\
\mathsf{refocus}\,(c_0\ c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{refocus_{aux}}\,([\,],\ (\lambda t)[s]) &= (\lambda t)[s] \\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ (\lambda t)[s]) &= \mathsf{refocus}\,(t[c \cdot s],\ C)
\end{aligned}
$$

As already observed by Danvy and Nielsen in their work on refocusing, inlining $\mathsf{iterate}$ yields an eval/apply abstract machine [32]: $\mathsf{refocus}$ (the 'eval' transition function) dispatches on closures and $\mathsf{refocus_{aux}}$ (the 'apply' function) dispatches on contexts.

3.5 A push/enter abstract machine

We now inline the calls to $\mathsf{refocus_{aux}}$ in the eval/apply abstract machine. The resulting machine reads as follows:

$$\mathsf{evaluate}\, t = \mathsf{refocus}\, (t[\bullet],\ [\,])$$

$$\begin{aligned}
\mathsf{refocus}\, (i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\, (c_i,\ C)\\
\mathsf{refocus}\, ((\lambda t)[s],\ [\,]) &= (\lambda t)[s]\\
\mathsf{refocus}\, ((\lambda t)[s],\ \mathsf{ARG}(c,\ C)) &= \mathsf{refocus}\, (t[c \cdot s],\ C)\\
\mathsf{refocus}\, ((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\, ((t_0[s])\ (t_1[s]),\ C)\\
\mathsf{refocus}\, (c_0\, c_1,\ C) &= \mathsf{refocus}\, (c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

As already observed by Ager et al. [4], inlining the apply transition function in a call-by-name eval/apply abstract machine yields a push/enter machine [32].

3.6 An optimized push/enter machine

The abstract machine of Section 3.5 only produces a composition of closures through an (App) reduction step (second-to-last clause of $\mathsf{refocus}$). We observe that in a reduction sequence, an (App) reduction step is always followed by a decomposition step (last clause of $\mathsf{refocus}$). As an optimization, we shortcut these two consecutive steps into one decomposition step, replacing the last two clauses of $\mathsf{refocus}$ with the following one:

$$\mathsf{refocus}\, ((t_0\, t_1)[s],\ C) = \mathsf{refocus}\, (t_0[s],\ \mathsf{ARG}(t_1[s],\ C))$$

The optimized machine never produces any composition of closures and therefore works for the $\lambda\rho$-calculus as well as for the $\lambda\widehat{\rho}$-calculus with the grammar of closures restricted to that of $\lambda\rho$.

3.7 An environment machine

Finally, we unfold the data type of closures. If we read each syntactic category as a type, then the type of closures is recursive:

$$\mathsf{Closure} \stackrel{\text{def}}{=} \mu X.\mathsf{Term} \times \mathsf{List}(X)$$

and furthermore

$$\mathsf{Substitution} \stackrel{\text{def}}{=} \mathsf{List}(\mathsf{Closure}).$$

Hence one unfolding of the type $\mathsf{Closure}$ yields $\mathsf{Term} \times \mathsf{Substitution}$. Therefore, for any closure $t[s]$ of type $\mathsf{Closure}$, its unfolding gives a pair $(t,\ s)$ of type $\mathsf{Term} \times \mathsf{Substitution}$. We replace each closure in the definition of $\mathsf{evaluate}$ and $\mathsf{refocus}$, in Section 3.6, by its unfolding. Flattening $(\mathsf{Term} \times \mathsf{Substitution}) \times \mathsf{Context}$ into $\mathsf{Term} \times \mathsf{Substitution} \times \mathsf{Context}$ yields the following abstract machine:

$$
\begin{array}{rcl}
v & ::= & (\lambda t,\, s) \\
C & ::= & [\,] \;\mid\; \mathsf{ARG}((t,\, s),\, C)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{evaluate} & : & \mathsf{Term} \rightarrow \mathsf{Value} \\
\mathsf{evaluate}\, t & = & \mathsf{refocus}\,(t,\, \bullet,\, [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{refocus} & : & \mathsf{Term} \times \mathsf{Substitution} \times \mathsf{Context} \rightarrow \mathsf{Value} \\
\mathsf{refocus}\,(i,\, (t_1,\, s_1) \cdots (t_m,\, s_m),\, C) & = & \mathsf{refocus}\,(t_i,\, s_i,\, C) \\
\mathsf{refocus}\,(\lambda t,\, s,\, [\,]) & = & (\lambda t,\, s) \\
\mathsf{refocus}\,(\lambda t,\, s,\, \mathsf{ARG}((t',\, s'),\, C)) & = & \mathsf{refocus}\,(t,\, (t',\, s') \cdot s,\, C) \\
\mathsf{refocus}\,(t_0\, t_1,\, s,\, C) & = & \mathsf{refocus}\,(t_0,\, s,\, \mathsf{ARG}((t_1,\, s),\, C))
\end{array}
$$

We observe that this machine coincides with the usual idealized version of Krivine's machine [10, 12, 24], in which evaluation contexts are treated as last-in, first-out lists (i.e., stacks) of closures. In particular, the substitution component assumes the rôle of the environment.

3.8 Correctness

We state the correctness of the final result—the idealized version of Krivine's machine —with respect to evaluation in the $\lambda\widehat{\rho}$-calculus.

Theorem 1. *For any term t in $\lambda\widehat{\rho}$,*

$$t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*} (\lambda t')[s] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\, s).$$

Proof. The proof relies on the correctness of refocusing [18], and the (trivial) meaning preservation of each of the subsequent transformations. ☐

The theorem states that Krivine's machine is correct in the sense that it computes closed weak head normal forms, and it realizes the normal-order strategy in the $\lambda\widehat{\rho}$-calculus, which makes it a call-by-name machine [33]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-name evaluation in the $\lambda\widehat{\rho}$-calculus. Since the reductions of $\lambda\rho$ can be simulated in $\lambda\widehat{\rho}$ (see Proposition 1), as a byproduct we obtain the correctness of Krivine's machine also with respect to Curien's original calculus of closures:

Corollary 1. *For any term t in $\lambda\rho$,*

$$t[\bullet] \xrightarrow{\rho}_{\mathrm{n}}{}^{*} (\lambda t')[s] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\, s).$$

However, Krivine's machine is better known as an environment machine that computes weak head normal forms of λ-terms. We show the correspondence theorem next.

3.9 Correspondence with the λ-calculus

The call-by-name evaluation relation in the λ-calculus is defined as the reflexive, transitive closure of the normal-order one-step reduction strategy shown in Section 2.

In order to relate values in the λ-calculus with values in the language of closures, we define a function σ that forces all the delayed substitutions in a $\lambda\widehat{\rho}$-closure. The function takes a closure and a number k indicating the current depth of the processed term (with respect to the number of surrounding λ-abstractions), and returns a λ-term:

$$
\sigma(i[s], k) = \begin{cases} i & \text{if } i \leq k \\ \sigma(c_{i-k}, k) & \text{if } k < i \leq m + k \text{ and } s = c_1 \cdots c_m \\ i - m & \text{if } i > m + k \text{ and } s = c_1 \cdots c_m \end{cases}
$$

$$
\begin{aligned}
\sigma((t_0\, t_1)[s], k) &= (\sigma(t_0[s], k))\ (\sigma(t_1[s], k)) \\
\sigma((\lambda t)[s], k) &= \lambda(\sigma(t[s], k + 1)) \\
\sigma(c_0\, c_1, k) &= \sigma(c_0, k)\ \sigma(c_1, k)
\end{aligned}
$$

The function σ puts us in position to state the correspondence theorem.

Theorem 2 (Correspondence). *For any λ-term t, $t \rightarrow_{\mathrm{n}}^{*} \lambda t'$ if and only if*

$$
\mathsf{evaluate}\, t = (\lambda t'',\ s) \quad \textit{and} \quad \sigma((\lambda t'')[s], 0) = \lambda t'.
$$

Proof. The left-to-right implication relies on the following property, proved by structural induction on t:

$$
\text{If } t \rightarrow_{\mathrm{n}} t',\ \text{then } t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*}\, c \text{ in } \lambda\widehat{\rho}, \text{ and } \sigma(c, 0) = t'.
$$

In order to prove the converse implication, we observe that if $c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'$, then either $\sigma(c, 0) \rightarrow_{\mathrm{n}} \sigma(c', 0)$, if the smallest reduction step uses the rule (β), or $\sigma(c, 0) = \sigma(c', 0)$ otherwise. The proof is done by structural induction on c, using the fact that $\sigma(t[s], j + 1)\{\sigma(c, 0)/j + 1\} = \sigma(t[c \cdot s], j)$. □

Curien, Hardin and Lévy consider several weak calculi of explicit substitutions capable of simulating call-by-name evaluation [13]. They relate these calculi to the λ-calculus with de Bruijn indices in much the same way as we do above. In fact, our function σ performs exactly σ-normalization in their strong calculus $\lambda\sigma$ for the restricted grammar of closures and substitutions of the $\lambda\rho$-calculus, and the structure of the proof of Theorem 2 is similar to that of their Theorem 3.6 [13]. More recently, Wand has used a translation U from closure to λ-terms with names that is an analog of σ, and presented a similar simple proof of the correctness of Krivine's machine for the λ-calculus with names [40].

4 From applicative-order reduction to call-by-value environment machine

Starting with the applicative-order reduction strategy specified in Section 2.4, we follow the same procedure as in Section 3.

4.1 The reduction semantics for applicative order

We first specify a reduction semantics for applicative-order reduction. The grammar of the source language is specified in Section 2.2, the syntactic notion of value and the collection of reduction rules are specified in Section 2.4, and the compatibility rules (ν) and (μ) induce the following grammar of reduction contexts:[2]

$$
C ::= [\,]\ \mid\ \mathsf{ARG}(c,\ C)\ \mid\ \mathsf{FUN}(v,\ C)
$$

[2]In the standard inside-out notation, this grammar can also be stated as follows:

$$
C ::= [\,]\ \mid\ C[[\,]c]\ \mid\ C[v[\,]]
$$

The redexes do not overlap and the source language satisfies a unique-decomposition property with respect to the applicative-order reduction strategy. Therefore, as in Section 3.1 we can define a contraction function, a decomposition function, a plug function, a one-step reduction function, and an evaluation function.

4.2 From evaluation function to environment machine

We now take the same steps as in Section 3. The reduction semantics of Section 4.1 satisfies the refocusing conditions, and Danvy and Nielsen's algorithm yields the following definition:

$$\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}\,(i[s],\ C) &= (i[s],\ C) \\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s]) \\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= ((t_0\, t_1)[s],\ C) \\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus_{aux}}\,([\,],\ v) &= v \\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ v) &= \mathsf{refocus}\,(c,\ \mathsf{FUN}(v,\ C)) \\
\mathsf{refocus_{aux}}\,(\mathsf{FUN}(v',\ C),\ v) &= (v'\, v,\ C)
\end{aligned}$$

We successively transform the resulting pre-abstract machine into a staged abstract machine and an eval/apply abstract machine.

The eval/apply abstract machine reads as follows:

$$\mathsf{evaluate}\, t = \mathsf{refocus}\,(t[\bullet],\ [\,])$$

$$\begin{aligned}
\mathsf{refocus}\,(i[v_1 \cdots v_m],\ C) &= \mathsf{refocus_{aux}}\,(C,\ v_i) \\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s]) \\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\ (t_1[s]),\ C) \\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus_{aux}}\,([\,],\ v) &= v \\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ (\lambda t)[s]) &= \mathsf{refocus}\,(c,\ \mathsf{FUN}((\lambda t)[s],\ C)) \\
\mathsf{refocus_{aux}}\,(\mathsf{FUN}((\lambda t)[s],\ C),\ v) &= \mathsf{refocus}\,(t[v \cdot s],\ C)
\end{aligned}$$

As in Section 3.6, we observe that we can shortcut the third and the fourth clauses of refocus (they are the only producer and consumer of a composition of closures, respectively). We can then unfold the data type of closures, as in Section 3.7, and obtain an eval/apply environment machine. This environment machine coincides with Felleisen et al.'s CEK machine [21].

4.3 Correctness and correspondence with the λ-calculus

As in Section 3.8, we state the correctness of the eval/apply machine with respect to the $\lambda\widehat{\rho}$-calculus.

Theorem 3. *For any term t in $\lambda\widehat{\rho}$,*

$$t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{v}}{}^{*} (\lambda t')[s'] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\ s').$$

The theorem states that the eval/apply machine is correct in the sense that it computes closed weak head normal forms, and it realizes the applicative-order strategy in the $\lambda\widehat{\rho}$-calculus, which makes it a call-by-value machine [33]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-value evaluation in the $\lambda\widehat{\rho}$-calculus.

The CEK machine is an environment machine for call-by-value evaluation in the λ-calculus. The following theorem formalizes this correspondence, using the function σ defined in Section 3.9.

Theorem 4 (Correspondence). *For any λ-term t, $t \rightarrow_{\mathrm{v}} \lambda t'$ if and only if*

$$\mathsf{evaluate}\, t = (\lambda t'',\, s) \quad \textit{and} \quad \sigma((\lambda t'')[s], 0) = \lambda t'.$$

5 A notion of context-sensitive reduction

Krivine's machine, as usually presented in the literature [2,4,10–12,18,23–26,31,37,40], contracts one β-redex at a time. The original version [27, 28], however, grammatically distinguishes nested λ-abstractions and contracts nested β-redexes in one step. Similarly, Leroy's Zinc machine [30] optimizes curried applications.

Krivine's language of λ-terms reads as follows:

$$\text{(terms)} \qquad t ::= i \;\mid\; t\,t \;\mid\; \lambda^n t$$

for $n \geq 1$, and where a nested λ-abstraction $\lambda^n t$ corresponds to $\overbrace{\lambda\lambda\ldots\lambda}^{n} t$, i.e., to n nested λ-abstractions, where t is not a λ-abstraction.

In the original version of Krivine's machine, and using the same notation as in Section 3, (nested) β-reduction is implemented by the following transition:

$$\begin{aligned}&\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{ARG}((t_1,\, s_1),\, \ldots \mathsf{ARG}((t_n,\, s_n),\, C)\ldots))\\ &= \mathsf{refocus}\,(t,\, (t_n,\, s_n)\cdots(t_1,\, s_1)\cdot s,\, C)\end{aligned}$$

This transition implements a nested β-reduction not just for the pair of terms forming a β-redex, or even for a tuple of terms forming a nested β-redex, but *for a nested λ-abstraction and the context of its application.* The contraction function is therefore not solely defined over the redex to contract, but over a term and its context: it is context-sensitive.

In this section, we adjust the definition of a reduction semantics with a $\mathsf{contract}$ function that maps a redex and its context to a contractum and its context. Nothing else changes in the definition, and therefore the refocusing method still applies. We first consider normal order, and we show how the usual idealized version of Krivine's machine arises, how the original version of Krivine's machine also arises, and how one can derive a slightly more perspicuous version of the original version. We then consider applicative order, and we show how the Zinc machine arises.

5.1 Normal order: variants of Krivine's machine

Let us consider the language of the $\lambda\widehat{\rho}$-calculus based on Krivine's modified grammar of terms. Together with the language comes the following grammar of reduction contexts, which is induced by the compatibility rule (ν), just as in Section 3.1.

$$C ::= [\,] \;\mid\; \mathsf{ARG}(c,\, C)$$

Let us adapt the rules of Section 2.3 for context-sensitive reduction in $\lambda\widehat{\rho}$:

$$
\begin{array}{lrcl}
(\beta^+) & ((\lambda^n t)[s],\ \mathsf{ARG}(c_1,\ \ldots \mathsf{ARG}(c_n,\ C)\ldots)) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} & (t[c_n \cdots c_1 \cdot s],\ C) \\
(\mathrm{Var}) & (i[c_1 \cdots c_m],\ C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} & (c_i,\ C) \\
(\mathrm{App}) & ((t_0\ t_1)[s],\ C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} & ((t_0[s])\ (t_1[s]),\ C) \\
(\nu) & (c_0\ c_1,\ C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} & (c_0,\ \mathsf{ARG}(c_1,\ C))
\end{array}
$$

The new (β^+) rule is the only reduction rule that actually depends on the context (in the subsequent two rules the context remains unchanged). The left-compatibility rule (ν) now explicitly constructs the reduction context.

We notice that with the context-sensitive notion of reduction we are in position to express the one-step normal-order strategy already in the $\lambda\rho$-calculus, if we combine (App) and (ν) in one rule:

$$((t_0\ t_1)[s],\ C) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[s],\ \mathsf{ARG}(t_1[s],\ C)).$$

In the context-sensitive reduction semantics corresponding to this normal-order context-sensitive reduction strategy, $\mathsf{contract}$ has type $\mathsf{Redex} \times \mathsf{Context} \rightarrow \mathsf{Closure} \times \mathsf{Context}$ and the one-step reduction function (see Section 3.1) reads as follows:

$$
\begin{array}{l}
\mathsf{reduce}\, c = \mathit{case}\ \mathsf{decompose}\, c \\
\qquad\qquad\quad \mathit{of}\ v \Rightarrow v \\
\qquad\qquad\quad\ \mid (r,\ C) \Rightarrow \mathsf{plug}\,(c, C') \quad \text{where } (c,\ C') = \mathsf{contract}\,(r,\ C)
\end{array}
$$

We then take the same steps as in Section 3. We consider three variants, each of which depends on the specification of n in each instance of (β^+).

5.1.1 The idealized version of Krivine's machine

Here, we choose n to be 1. We then take the same steps as in Section 3, and obtain the same machine as in Section 3.7, i.e., the usual idealized version of Krivine's machine.

5.1.2 The original version of Krivine's machine

Here, we choose n to be the "arity" of each nested λ-abstraction, i.e., the number of nested λ's surrounding a term (the body of the innermost λ-abstraction) which is not a λ-abstraction.

We then take the same steps as in Section 3, and obtain the same machine as Krivine [27,28]. As pointed out by Wand [40], since the number of arguments is required to match the number of nested λ-abstractions, the machine becomes stuck if there are not enough arguments in the context. We handle this case by adapting the β^+-rule as described next.

5.1.3 An adjusted version of Krivine's machine

Here, we choose n to be the smallest number between the arity of each nested λ-abstraction and the number of nested applications. (So $n = 1$ for $(\lambda\lambda t,\ \mathsf{ARG}(c_1,\ [\,]))$, for example.)

We then take the same steps as in Section 3, and obtain a version of Krivine's machine that directly computes weak head normal forms.

5.2 Applicative order: variants of the Zinc machine

From Landin [29] to Leroy [30], implementors of call-by-value functional languages have looked fondly upon right-to-left evaluation (i.e., evaluating the actual parameter before the function part of an application) because of its fit with a stack implementation: when the function part of a (curried) application yields a functional value, its parameter(s) is (are) available on top of the stack, as in the call-by-name case. In this section, we consider right-to-left applicative order and call by value, which as in the normal order and call-by-name case, give rise to a push/enter abstract machine.

We first adapt the rules of Section 2.4 for context-sensitive reduction in $\lambda\widehat{\rho}$. First of all, the compatibility rules (ν) and (μ), for right-to-left applicative order, induce the following grammar of contexts:[3]

$$C ::= [\,] \mid \mathsf{ARG}(v,\ C) \mid \mathsf{FUN}(c,\ C)$$

This grammar differs from the one of Section 4.1 because it is for right-to-left instead of for left-to-right applicative order.

The rules therefore read as follows:

$$\begin{array}{lrl}
(\beta^+) & ((\lambda^n t)[s],\ \mathsf{ARG}(v_1,\ \ldots \mathsf{ARG}(v_n,\ C)\ldots)) & \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (t[v_n \cdots v_1 \cdot s],\ C) \\
(\mathrm{Var}) & (i[v_1 \cdots v_m],\ C) & \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (v_i,\ C) \\
(\mathrm{App}) & ((t_0\ t_1)[s],\ C) & \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} ((t_0[s])\ (t_1[s]),\ C) \\
(\nu') & (v,\ \mathsf{FUN}(c,\ C)) & \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (c,\ \mathsf{ARG}(v,\ C)) \\
(\mu') & (c_0\, c_1,\ C) & \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (c_1,\ \mathsf{FUN}(c_0,\ C))
\end{array}$$

Similarly to the call-by-name case, the only context-sensitive reduction rule is (β^+) (we specify n as in Section 5.1.3), and the two compatibility rules explicitly construct reduction contexts.

As in Section 5.1.1, we notice that with the context-sensitive notion of reduction we are in position to express the one-step applicative-order strategy in the $\lambda\rho$-calculus if we combine (App) and (μ') in one rule:

$$((t_0\ t_1)[s],\ C) \stackrel{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (t_1[s],\ \mathsf{FUN}(t_0[s],\ C)).$$

We now take the same steps as in Section 4. The reduction semantics of Section 4.1, adapted to the grammar of contexts and to the reduction rules above, satisfies the refocusing conditions, and Danvy and Nielsen's algorithm yields the following definition:

[3] In the standard inside-out notation, this grammar can also be stated as follows:

$$C ::= [\,] \mid C[[\,]v] \mid C[c[\,]]$$

$$
\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}\ (i[s],\ C) &= (i[s],\ C) \\
\mathsf{refocus}\ ((\lambda^n t)[s],\ C) &= \mathsf{refocus_{aux}}\ (C,\ (\lambda^n t)[s]) \\
\mathsf{refocus}\ ((t_0\ t_1)[s],\ C) &= \mathsf{refocus}\ (t_1[s],\ \mathsf{FUN}(t_0[s],\ C))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus_{aux}}\ ([\,],\ v) &= v \\
\mathsf{refocus_{aux}}\ (\mathsf{FUN}(c,\ C),\ v) &= \mathsf{refocus}\ (c,\ \mathsf{ARG}(v,\ C)) \\
\mathsf{refocus_{aux}}\ (\mathsf{ARG}(v',\ C),\ v) &= (v,\ \mathsf{ARG}(v',\ C))
\end{aligned}
$$

We then successively transform the resulting pre-abstract machine into a staged abstract machine, an eval/apply abstract machine, a push/enter abstract machine, and a push/enter abstract machine with unfolded closures.

The resulting environment machine reads as follows:

$$
\mathsf{evaluate}\ t = \mathsf{refocus}\ (t,\ \bullet,\ [\,])
$$

$$
\begin{aligned}
\mathsf{refocus}\ (i,\ (t_1,\ s_1)\cdots(t_m,\ s_m),\ C) &= \mathsf{refocus}\ (t_i,\ s_i,\ C) \\
\mathsf{refocus}\ (\lambda^n t,\ s,\ [\,]) &= (\lambda^n t,\ s) \\
\mathsf{refocus}\ (\lambda^n t,\ s,\ \mathsf{FUN}((t',\ s'),\ C)) &= \mathsf{refocus}\ (t',\ s',\ \mathsf{ARG}((\lambda^n t,\ s),\ C)) \\
\mathsf{refocus}\ (\lambda^n t,\ s,\ \mathsf{ARG}(c_1,\ \ldots \mathsf{ARG}(c_n,\ C)\ldots)) &= \mathsf{refocus}\ (t,\ c_n \cdot \ldots \cdot c_1 \cdot s,\ C) \\
\mathsf{refocus}\ (t_0\ t_1,\ s,\ C) &= \mathsf{refocus}\ (t_1,\ s,\ \mathsf{FUN}((t_0,\ s),\ C))
\end{aligned}
$$

This machine corresponds to an instance of Leroy's Zinc machine for the pure λ-calculus [30, Chapter 3], with the proviso that it operates directly on λ-terms instead of over an instruction set (which has been said to be the difference between an abstract machine and a virtual machine [3]). Moreover, in the Zinc machine the sequence of values on the stack (denoted here by $\mathsf{ARG}(c_1,\ \ldots \mathsf{ARG}(c_n,\ C)\ldots))$ is delimited by a stack mark that separates already evaluated terms from unevaluated ones. The Zinc machine was developed independently of Krivine's machine and to the best of our knowledge the reconstruction outlined here is new.

6 A space optimization for call-by-name evaluation

In the compilation model of ALGOL 60, which is a call-by-name programming language, identifiers occurring as actual parameters are compiled by (1) looking up their value in the current environment, and (2) passing this value to the callee [34, Section 2.5.4.10, pages 109-110]. The rationale is that under call by name, an identifier denotes a thunk, so there is no need to create another thunk for it. This compilation rule avoids a space leak at run time and it is commonly used in implementations of lazy functional programming languages.

To circumvent the space leak, one extends Krivine's machine with the following clause for the case where the actual parameter is a variable:

$$
\begin{aligned}
&\mathsf{refocus}\ (t_0\ i,\ (t_1,\ s_1)\cdots(t_m,\ s_m),\ C) \\
&= \mathsf{refocus}\ (t_0,\ (t_1,\ s_1)\cdots(t_m,\ s_m),\ \mathsf{ARG}((t_i,\ s_i),\ C))
\end{aligned}
$$

We observe that circumventing the space leak has an analogue in the normal-order reduction semantics: it corresponds to adding the following reduction rule to the $\lambda\widehat{\rho}$-calculus:

$$
(\mathrm{App}') \qquad (t_0\ i)[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[c_1 \cdots c_m])\ c_i
$$

This addition shortens the reduction sequence of a given λ-term towards its weak head normal form.

The context-insensitive reduction semantics: Taking the same steps as in Section 3 mechanically leads one to an idealized version of Krivine's machine with the space optimization. Crégut has considered this space-optimized version [10] and Friedman, Ghuloum, Siek, and Winebarger have measured the impact of this optimization for a lazy version of Krivine's machine [23].

The context-sensitive reduction semantics: Taking the same steps as in Section 3 mechanically leads one to an adjusted version of Krivine's machine with the space optimization.

7 Actual substitutions, explicit substitutions, and environments

The derivations in Sections 3, 4, 5, and 6 hint at a bigger picture that we draw in Figure 1 and describe in Section 7.1. We then address the reversibility of the derivation steps in Section 7.2.

7.1 A derivational taxonomy of abstract machines

Let us analyze Figure 1.

The left-most column concerns the reduction and evaluation of terms with actual substitutions (λ). The second column concerns the reduction and evaluation of terms with explicit substitutions ($\lambda\widehat{\rho}$). The third column concerns the evaluation of terms with explicit substitutions ($\lambda\rho$). The right-most column concerns the evaluation of terms using an environment.

Reading down each column follows Danvy and Nielsen's work on refocusing [18]. They show how to go from an evaluation function (obtained as the transitive closure of a one-step reduction function) to a pre-abstract machine, and then to a staged machine, an eval/apply machine, and, for call by name and right-to-left call by value, a push/enter machine.

The connections between the columns in the 4×4-matrix are new. Given a closure (i.e., a term and a substitution), carrying out the substitution in this term yields a new term. Through this operation (depicted with the short dotted arrow in the diagram), we can go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for λ-terms. To go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for $\lambda\rho$-closures, we shortcut the (App) reduction step with the following decomposition step. To go from each of the abstract machines for $\lambda\rho$-closures to the corresponding environment machine, we unfold the data type of closures into a pair of term and substitution (the unfolded substitution acts as the environment of the machine). Finally, given a term and an environment, carrying out the delayed substitutions represented by the environment in the term yields a new term. Through this operation (depicted with the long dotted arrow in the diagram), we can go from each of the environment machines to the corresponding abstract machine for λ-terms.

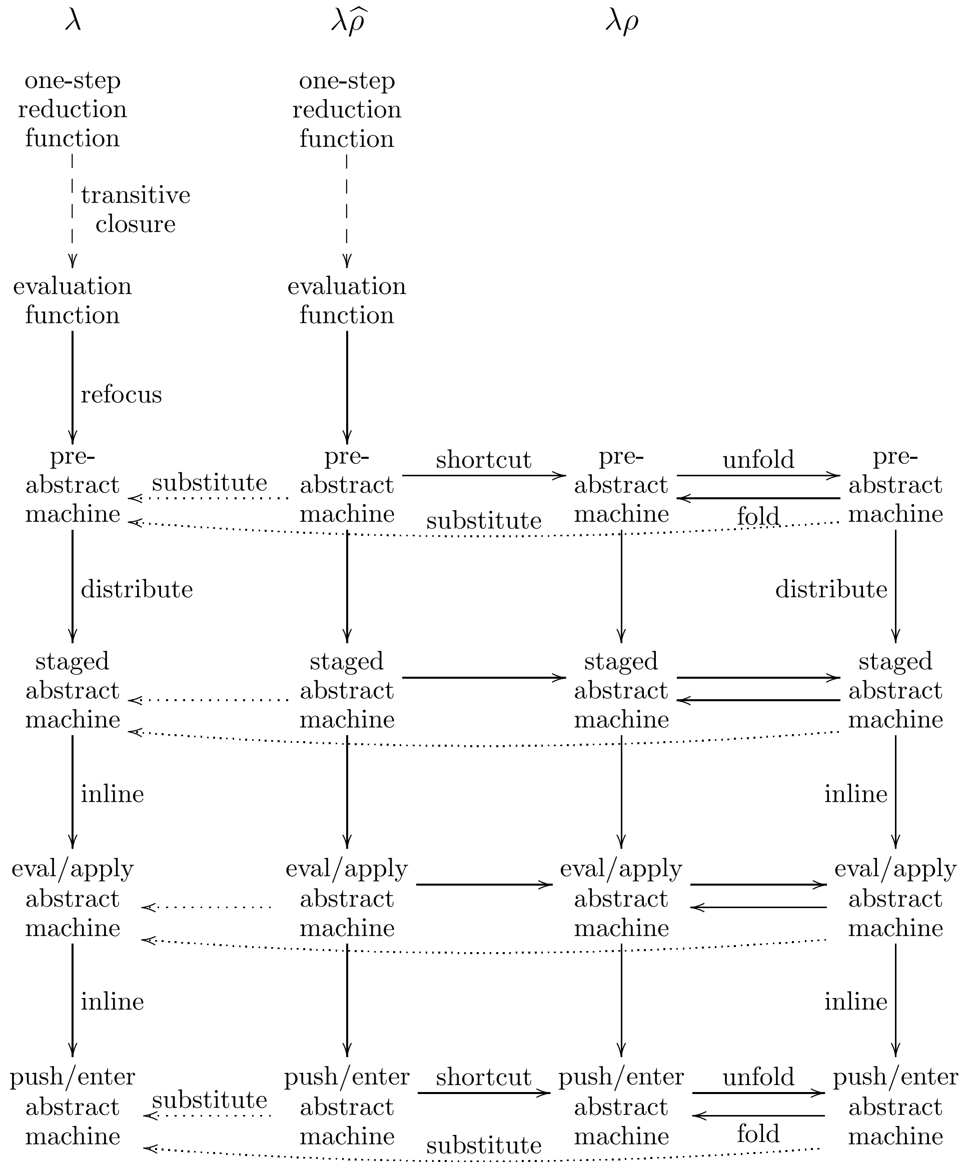


Figure 1: A derivational taxonomy of abstract machines

In Section 3, we observed that the push/enter environment machine corresponding to normal-order reduction coincides with the usual idealized version of Krivine's machine [10, 12, 24]. In Section 4, we observed that the eval/apply environment machine corresponding to applicative-order reduction coincides with Felleisen's et al.'s CEK machine [21]. In Section 5, we observed that a context-sensitive reduction semantics gives rise to the original version of Krivine's machine and to the Zinc machine. In Section 6, we observed that the space optimization of two families of call-by-name machines corresponds to two reduction semantics.

Earlier [18], Danvy and Nielsen observed that the eval/apply machine with actual substitution corresponding to applicative-order reduction coincides with Felleisen et al.'s CK machine [21]. Obtaining staged abstract machines was one of the goals of Hardin, Maranget, and Pagano's study of functional runtime systems using explicit substitutions [26]; these machines arise mechanically here. As investigated by Danvy and his students in their study of the functional correspondence between compositional evaluation functions and abstract machines [4–6, 16], the eval/apply machines are in defunctionalized form [17, 35] and they can be 'refunctionalized' into a continuation-passing evaluation function which itself can be written in direct style [14] and implements a big-step operational semantics. (Because of the closures, the result is again in defunctionalized form and can be refunctionalized into a compositional evaluation function implementing the valuation function of a denotational semantics.)

Each of the machines in Figure 1 is thus of independent value.

7.2 Reversibility of the derivation steps

Going from a push/enter machine to the corresponding eval/apply machine or from a push/enter machine or an eval/apply machine to a staged machine requires a degree of insight [26]. Going from a staged machine to a pre-abstract machine and from a pre-abstract machine to a reduction semantics is mechanical. We are however not aware of any systematic method to go from a given abstract machine to a reduction semantics [8, 22].

Going from an environment machine where the environment is treated as a list to a closure-based machine is done by folding the pair (term, environment) into a closure. The resulting machine mediates between an environment-based specification and an explicit-substitution-based specification. Obtaining an explicit-substitution-based machine from this intermediate machine, however, requires a major architectural overhaul.

8 Conclusion

Curien originally presented a simple calculus of closures, the $\lambda\rho$-calculus, as an abstract framework for environment machines [12]. This approach gave rise to a general study of explicit substitutions [1,2,13,26,31,36,37] where a number of abstract machines have been obtained through a combination of skill and ingenuity.

We have presented a concrete framework for environment machines where abstract machines are methodically derived from specifications of reduction strategies. The derivation is based on Danvy and Nielsen's refocusing method, which requires the one-step specification of a reduction strategy, i.e., a reduction semantics. For this reason, we needed to extend Curien's original $\lambda\rho$-calculus with closure composition, yielding the $\lambda\widehat{\rho}$-calculus.

We have illustrated the concrete framework by deriving several known environment machines—Krivine's abstract machine both in idealized form and in original form, Felleisen et al.'s CEK machine, and Leroy's Zinc machine—from the normal-order and the applicative-order reduction strategies expressed in the $\lambda\widehat{\rho}$-calculus, both in context-insensitive and in context-sensitive form. The last step of the derivation (closure unfolding) provides a precise characterization of what has now become folklore: that explicit substitutions mediate between actual substitutions and environments.

Acknowledgments: Thanks are due to Mads Sig Ager, Dariusz Biernacki, Mayer Goldberg, Julia Lawall, Jan Midtgaard, Kevin Millikin, and Kristian Støvring for comments, and to Ulrich Kohlenbach for providing us with what appears to be the original bibliographic reference of the logic counterpart of environments [38, § 54].

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California, January 1990. ACM Press.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[3] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Research Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[4] 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. ACM Press, August 2003.

[5] 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 technical report BRICS-RS-04-3.

[6] 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*, 2005. To appear. Extended version available as the technical report BRICS RS-04-28.

[7] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[8] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy (revised version). Research Report BRICS RS-05-11, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[9] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[10] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [39], pages 333–340.

[11] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 2006. To appear.

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

[13] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[14] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[15] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[16] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin price. Extended version available as the technical report BRICS-RS-03-33.

[17] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[18] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[19] Nicholas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[20] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[21] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[22] 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.

[23] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Lynn Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 2006. To appear.

[24] Chris Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[25] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In Wand [39], pages 323–332.

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

[27] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`, 1985.

[28] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 2006. To appear. Available online at `http://www.pps.jussieu.fr/~krivine/`.

[29] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[30] Xavier Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, February 1990.

[31] Pierre Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, January 1994. ACM Press.

[32] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

[33] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[34] Brian Randell and Lawford John Russell. *ALGOL 60 Implementation*. Academic Press, London and New York, 1964.

[35] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[36] Kristoffer H. Rose. Explicit substitution – tutorial & survey. BRICS Lecture Series LS-96-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1996.

[37] Kristoffer H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1996.

[38] Heinrich Scholz and Gisbert Hasenjaeger. *Grundzüge der Mathematischen Logik*. Springer-Verlag, 1961.

[39] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[40] Mitchell Wand. On the correctness of the Krivine machine. *Higher-Order and Symbolic Computation*, 2006. To appear.

Recent BRICS Report Series Publications

RS-05-15 **Małgorzata Biernacka and Olivier Danvy.** ***A Concrete Framework for Environment Machines*****. May 2005. ii+25 pp.**

RS-05-14 **Olivier Danvy and Henning Korsholm Rohde.** ***On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation*****. April 2005. ii+8 pp.**

RS-05-13 **Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan.** ***On the Dynamic Extent of Delimited Continuations*****. April 2005. ii+32 pp. Extended version of an article to appear in** ***Information Processing Letters*****. Subsumes BRICS RS-05-2.**

RS-05-12 **Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring.** ***Program Extraction from Proofs of Weak Head Normalization*****. April 2005. 19 pp. Extended version of an article to appear in the preliminary proceedings of MFPS XXI, Birmingham, UK, May 2005.**

RS-05-11 **Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy.** ***An Operational Foundation for Delimited Continuations in the CPS Hierarchy*****. March 2005. iii+42 pp. A preliminary version appeared in Thielecke, editor,** ***4th ACM SIGPLAN Workshop on Continuations*****, CW '04 Proceedings, Association for Computing Machinery (ACM) SIGPLAN Technical Reports CSR-04-1, 2004, pages 25–33. This version supersedes BRICS RS-04-29.**

RS-05-10 **Dariusz Biernacki and Olivier Danvy.** ***A Simple Proof of a Folklore Theorem about Delimited Control*****. March 2005. ii+11 pp.**

RS-05-9 **Gudmund Skovbjerg Frandsen and Peter Bro Miltersen.** ***Reviewing Bounds on the Circuit Size of the Hardest Functions*****. March 2005. 6 pp. To appear in** ***Information Processing Letters*****.**

RS-05-8 **Peter D. Mosses.** ***Exploiting Labels in Structural Operational Semantics*****. February 2005. 15 pp. Appears in** ***Fundamenta Informaticae*****, 60:17–31, 2004.**

RS-05-7 **Peter D. Mosses.** ***Modular Structural Operational Semantics*****. February 2005. 46 pp. Appears in** ***Journal of Logic and Algebraic Programming*****, 60–61:195–228, 2004.**

RS-05-6 **Karl Krukow and Andrew Twigg.** ***Distributed Approximation of Fixed-Points in Trust Structures*****. February 2005. 41 pp.**