BRICS RS-06-3 Biernacka & Danvy: A Concrete Framework for Environment Machines

Basic Research in Computer Science

# A Concrete Framework for Environment Machines

**Małgorzata Biernacka**
**Olivier Danvy**

**BRICS Report Series** **RS-06-3**

**ISSN 0909-0878** **February 2006**

**Copyright © 2006, Małgorzata Biernacka & Olivier Danvy.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**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
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
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/06/3/`

# A Concrete Framework for Environment Machines*

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

February 3, 2006

**Abstract**

We materialize the common understanding that calculi with explicit substitutions provide an intermediate step between an abstract specification of substitution in the lambda-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 lambda-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 Krivine machine and the original version of Krivine's machine, Felleisen et al.'s CEK machine, and Leroy's Zinc abstract machine.

*To appear in the ACM Transactions on Computational Logic.
†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** **1**

**2 One-step reduction in a calculus of closures** **3**
2.1 Curien's calculus of closures . . . . . 3
2.2 A minimal extension to Curien's calculus of closures . . . . . 4
2.3 Specification of the normal-order reduction strategy . . . . . 7
2.4 Specification of the applicative-order reduction strategy . . . . . 7
2.5 Correspondence with the $\lambda$-calculus . . . . . 8

**3 From normal-order reduction to call-by-name environment machine** **8**
3.1 A reduction semantics for normal order . . . . . 9
3.1.1 Reduction semantics . . . . . 9
3.1.2 A normal-order reduction semantics for the $\lambda\widehat{\rho}$-calculus . . . . 10
3.1.3 Decomposition . . . . . 10
3.1.4 Contraction . . . . . 11
3.1.5 Plugging . . . . . 11
3.1.6 One-step reduction . . . . . 11
3.1.7 Reduction-based evaluation . . . . . 11
3.2 A pre-abstract machine . . . . . 12
3.3 A staged abstract machine . . . . . 13
3.4 An eval/apply abstract machine . . . . . 14
3.5 A push/enter abstract machine . . . . . 14
3.6 A push/enter abstract machine for the $\lambda\rho$-calculus . . . . . 14
3.7 An environment machine for the $\lambda$-calculus . . . . . 15
3.8 Correctness . . . . . 15
3.9 Correspondence with the $\lambda$-calculus . . . . . 16

**4 From applicative-order reduction to call-by-value environment machine** **16**
4.1 The reduction semantics for left-to-right applicative order . . . . . 16
4.2 From evaluation function to environment machine . . . . . 17
4.3 Correctness and correspondence with the $\lambda$-calculus . . . . . 18

**5 A notion of context-sensitive reduction** **18**
5.1 Normal order: variants of Krivine's machine . . . . . 19
5.1.1 The Krivine machine . . . . . 19
5.1.2 The original version of Krivine's machine . . . . . 20
5.1.3 An adjusted version of Krivine's machine . . . . . 20
5.2 Right-to-left applicative order: variants of the Zinc machine . . . . . 20

**6 A space optimization for call-by-name evaluation** **22**

**7 Actual substitutions, explicit substitutions, and environments** **22**
7.1 A derivational taxonomy of abstract machines . . . . . 22
7.2 Reversibility of the derivation steps . . . . . 25

**8 Conclusion** **25**

# 1 Introduction

The Krivine machine and the CEK machine are probably the simplest examples of abstract machines—i.e., first-order state-transition functions [37]—implementing an evaluation function of the $\lambda$-calculus [16, 24, 30]. 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 an explicit way of handling substitution. In contrast, the traditional presentations of the $\lambda$-calculus [9, page 9] [6, page 51] implicitly specify the $\beta$-rule using a meta-level notion of substitution:

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

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 $\alpha$-conversions to avoid variable capture). Most implementations, however, do not realize the $\beta$-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 [43, § 54] and to Landin in computer science [32]. In logic, it makes it possible to reason by structural induction over $\lambda$-terms (since they do not change across $\beta$-reduction), and in computer science, it makes it possible to compile $\lambda$-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, 12, 13, 29, 34, 41, 42]. 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 [32]), i.e., a term together with its lexical environment. Such an environment is hereby referred to as 'substitution' to follow tradition [1, 12]. Moreover, variables are often represented with de Bruijn indices [21] (i.e., lexical offsets in compiler parlance [38]) 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, $\beta$-reduction is specified using one rule for extending the substitution with a closure to be substituted, such as

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

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

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

where $c$ is the $i$th 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). In this article, we consider weak calculi.

**This work:** We present a simple method for deriving an environment machine from the specification of a one-step reduction strategy in a weak calculus of closures. The method consists in 'refocusing' the evaluation function [20], coalescing two steps into one, and unfolding 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 $\lambda$-calculus and its implementations via abstract machines. He illustrates his argument by constructing the Krivine machine from a multi-step normal-order reduction strategy.

We observe, however, that one-step reductions cannot be expressed in $\lambda\rho$ and therefore in Section 2, we propose a minimal extension—the $\lambda\widehat{\rho}$-calculus—to make it capable of expressing such computations. In Section 3, we present a detailed derivation of the Krivine machine [10, 12, 27] from the specification of the normal-order one-step strategy in $\lambda\widehat{\rho}$. In Section 4, we outline the derivation of its applicative-order analog, the CEK machine [24]. In Section 5, we outline the derivation of the original version of Krivine's machine [31] and of its applicative-order analog, which we identify as Leroy's Zinc abstract machine [33]. In Section 6, we consider a space optimization for call by name that originates in Algol 60, and we characterize it as the machine counterpart of an extra contraction rule in the $\lambda\widehat{\rho}$-calculus. In Section 7, we draw a bigger picture before concluding in Section 8.

**Prerequisites and notation:** We assume a basic familiarity with the $\lambda$-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).

**Reduction in the $\lambda$-calculus:** As a reference point, let us specify the one-step and multi-step reduction relations in the standard $\lambda$-calculus.

We first recall the formulation of the $\lambda$-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 $\lambda$-calculus, one-step reduction is defined as the compatible closure of the notion of reduction given by the $\beta$-rule [6, Section 3.1]:

$$(\beta) \qquad (\lambda t_0)\,t_1 \to t_0\{t_1/1\}$$

where $t_0\{t_1/1\}$ is a meta-level substitution operation (with suitable reindexing of variables). As for the compatible closure, it is built according to the following compatibility rules:

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

Weak (nondeterministic) subsystems are obtained by discarding the $\xi$-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 ($\mu$). In effect, one obtains the following normal-order one-step reduction strategy for the $\lambda$-calculus, producing weak head normal forms:

$$(\beta) \qquad (\lambda t_0)\, t_1 \to_{\mathrm{n}} t_0\{t_1/1\}$$

$$(\nu) \qquad \frac{t_0 \to_{\mathrm{n}} t_0'}{t_0\, t_1 \to_{\mathrm{n}} t_0'\, t_1}$$

  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 imposes a restriction on the $\beta$-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 $\lambda$-abstractions.

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

$$(\nu) \qquad \frac{t_0 \to_{\mathrm{v}} t_0'}{t_0\, t_1 \to_{\mathrm{v}} t_0'\, t_1}$$

$$(\mu) \qquad \frac{t_1 \to_{\mathrm{v}} t_1'}{t_0\, t_1 \to_{\mathrm{v}} t_0\, t_1'} \quad \text{if } t_0 \text{ is a value}$$

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

$$(\nu) \qquad \frac{t_0 \to_{\mathrm{v}} t_0'}{t_0\, t_1 \to_{\mathrm{v}} t_0'\, t_1} \quad \text{if } t_1 \text{ is a value}$$

$$(\mu) \qquad \frac{t_1 \to_{\mathrm{v}} t_1'}{t_0\, t_1 \to_{\mathrm{v}} t_0\, t_1'}$$

# 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.

## 2.1 Curien's calculus of closures

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

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

Terms are defined as in the $\lambda$-calculus with de Bruijn indices. A $\lambda\rho$-closure is a pair consisting of a $\lambda$-term and a $\lambda\rho$-substitution, which itself is a finite list of $\lambda\rho$-closures to be substituted for free variables in the $\lambda$-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:

$$\text{(Eval)} \qquad \frac{t_0[s] \xrightarrow{\rho}{}^* (\lambda t_0')[s']}{(t_0\ t_1)[s] \xrightarrow{\rho} t_0'[(t_1[s]) \cdot s']}$$

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

$$\text{(Env)} \qquad \frac{c_1 \xrightarrow{\rho}{}^* c_1' \quad \ldots \quad c_m \xrightarrow{\rho}{}^* c_m'}{t[c_1 \cdots c_m] \xrightarrow{\rho} t[c_1' \cdots c_m']}$$

where $\xrightarrow{\rho}{}^*$ is the reflexive, transitive closure of $\xrightarrow{\rho}$. Reductions are performed on closures, and not on individual terms. The grammar of weak head normal forms is as follows:

$$c_{\text{nf}} ::= (\lambda t)[s] \mid (i\ t_1 \ldots t_m)[s],$$

where $i$ is greater than the length of $s$. If we restrict ourselves to considering only closures with no free variables (i.e., closures $t[s]$ whose term $t$ is closed by the substitution $s$), then weak head normal forms are closures 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 $\xrightarrow{\rho}_{\text{n}}$) 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 "combine" 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 application. We denote it simply by juxtaposition:

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

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) $\beta$-reduction:

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

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

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

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:*

*a. If $c_0 \xrightarrow{\rho} c_1$, then $c_0 \xrightarrow{\widehat{\rho}}{}^* c_1$.*

*b. If $c_0 \xrightarrow{\widehat{\rho}}{}^* c_1$, then $c_0 \xrightarrow{\rho}{}^* c_1$.*

*Proof.* The proofs are done by induction:

*Property a*: We define the complexity of derivations $c_0 \xrightarrow{\rho}{}^* c_1$ in $\lambda\rho$ as follows: the complexity of a multiple-step derivation is the sum of the complexities of all its steps. (In particular, 0-step derivations have complexity 0.) For one-step derivations, the complexity of a derivation starting with (Eval) or (Env) is defined as the maximum of the complexities of its immediate subderivations augmented by 1, and the complexity of an instance of (Var) is 1.

By induction on the complexity of derivations, we prove that if $c_0 \xrightarrow{\rho}{}^* c_1$, then $c_0 \xrightarrow{\widehat{\rho}}{}^* c_1$. From this, Property *a.* follows.

For $n = 0$, it is trivial. For the inductive step $n+1$, we note that the derivation $c_0 \xrightarrow{\rho}{}^* c_1$ has at least one step: $c_0 \xrightarrow{\rho} c_0' \xrightarrow{\rho}{}^* c_1$, and the complexity of $c_0 \xrightarrow{\rho} c_0'$ is at most $n+1$. We prove that $c_0 \xrightarrow{\widehat{\rho}}{}^* c_0'$ by distinguishing three cases, depending on the rule applied in the first step:

**Case** (Eval)**.** Then $c_0$ is $(t_0\, t_1)[s]$ and $c_0'$ is $t_0'[(t_1[s]) \cdot s']$, with $t_0[s] \xrightarrow{\rho}{}^* (\lambda t_0')[s']$. We then know that the derivation $t_0[s] \xrightarrow{\rho}{}^* (\lambda t_0')[s']$ has complexity $\leq n$. Then by induction hypothesis, $t_0[s] \xrightarrow{\widehat{\rho}}{}^* (\lambda t_0')[s']$. Hence we obtain the following reduction sequence in $\lambda\widehat{\rho}$, where the first step is an application of (App), and the last one is an application of ($\beta$):

$$(t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\ (t_1[s]) \xrightarrow{\widehat{\rho}}{}^* ((\lambda t_0')[s'])\ (t_1[s]) \xrightarrow{\widehat{\rho}} t_0'[(t_1[s]) \cdot s'].$$

**Case** (Var)**.** This step is directly simulated by (Var) in $\lambda\widehat{\rho}$.

**Case** (Env)**.** Then $c_0$ is $t[c_1 \cdots c_m]$ and $c_0'$ is $t[c_1' \cdots c_m']$, with $c_i \xrightarrow{\rho}{}^* c_i'$ for all $1 \leq i \leq m$. By the definition of complexity, each of the subderivations $c_i \xrightarrow{\rho}{}^* c_i'$ is of complexity not greater than $n$. Hence, by the induction hypothesis, $c_i \xrightarrow{\widehat{\rho}}{}^* c_i'$ for all $i$, and successively applying (Sub) in $\lambda\widehat{\rho}$ yields

$$t[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}{}^* t[c_1' \cdots c_m] \xrightarrow{\widehat{\rho}}{}^* \cdots \xrightarrow{\widehat{\rho}}{}^* t[c_1' \cdots c_m'].$$

Now we can apply the induction hypothesis to the remaining reduction sequence $c_0' \xrightarrow{\rho}{}^* c_1$, whose complexity is at most $n$, and we obtain a complete simulation of $c_0 \xrightarrow{\rho} c_0' \xrightarrow{\rho}{}^* c_1$ in $\lambda\widehat{\rho}$.

*Property b*: For the proof of this property we need the following three lemmas.

**Lemma 1.** *If $c_0$ is a $\lambda\rho$-closure and $c_0 \xrightarrow{\widehat{\rho}}{}^* t[s]$, then there exists a $\lambda\rho$-closure $t[s_0]$ such that $c_0 \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} t[s_0]$ and $t[s_0] \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{Sub}} t[s]$, where $\xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}}$ denotes the normal order reduction strategy in $\lambda\widehat{\rho}$ obtained by dropping the reduction rules* (Sub) *and* ($\mu$)*, and $\xrightarrow{\widehat{\rho}}{}^*_{\mathrm{Sub}}$ denotes a reduction sequence consisting only of* (Sub) *steps.*

*Proof.* The proof proceeds by induction on the complexity of the derivation $c_0 \xrightarrow{\widehat{\rho}}{}^* t[s]$ along the lines of an analogous property for Curien's $\lambda\rho$-calculus [12, Lemma 2.2]. To this end, we use an intermediate calculus $\lambda\tilde{\rho}$ obtained from $\lambda\widehat{\rho}$ by replacing the (Sub) rule by an analog of the (Env) rule of the $\lambda\rho$-calculus (i.e., we can collect in one step a number of (Sub)-reductions). We immediately see that any reduction sequence in $\lambda\widehat{\rho}$ can be simulated in $\lambda\tilde{\rho}$ and vice versa. We then prove the above lemma for $\lambda\tilde{\rho}$, using the measure of complexity of (Env) derivations as in the $\lambda\rho$-calculus.

In the proofs of Lemmas 1 and 2 we make use of the following observation: in a reduction sequence of the form

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

the rule ($\beta$) must be applied at some point, because ($\beta$) is the only rule that transforms a composition of closures into a single closure. Hence, within such a derivation, $t_0[s]$ must reduce to a value, and there must be a ($\beta$) reduction applied to $t_1[s]$ (possibly also reduced). □

**Lemma 2.** *If $c_0$ and $c_1$ are $\lambda\rho$-closures, and $c_0 \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} c_1$, then $c_0 \xrightarrow{\rho}{}^* c_1$.*

*Proof.* By induction on the length of the derivation $c_0 \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} c_1$. For the inductive case, assume $c_0 \xrightarrow{\widehat{\rho}}_{\mathrm{no}} c_0' \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} c_1$ of length $n+1$. In the first step, two rules can be applied: (Var) and (App). The former case follows immediately from the induction hypothesis. In the latter case: according to (App), $c_0 = (t_0 t_1)[s]$ and $c_0' = (t_0[s])(t_1[s])$. Analyzing the reduction rules, we observe that in the subsequent reduction sequence the rule ($\beta$) must be applied. Hence, we have $t_0[s] \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} (\lambda t_0')[s_0]$ of length $k_1$, and

$$c_0' \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} ((\lambda t_0')[s_0])\,(t_1[s]) \xrightarrow{\widehat{\rho}} t_0'[(t_1[s]) \cdot s_0] \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} c_1,$$

where the length of $t_0'[(t_1[s]) \cdot s_0] \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{no}} c_1$ is $k_2$ and $k_1 + 1 + k_2 = n$. It is easy to see that if we start with a $\lambda\rho$-closure, then using the normal order reduction strategy, $(\lambda t_0')[s_0]$ must also be a $\lambda\rho$-closure. By induction hypothesis then $t_0[s] \xrightarrow{\rho}{}^* (\lambda t_0')[s_0]$, hence $c_0 = (t_0\ t_1)[s] \xrightarrow{\rho} t_0'[(t_1[s]) \cdot s_0]$ by (Eval), and $t_0'[(t_1[s]) \cdot s_0] \xrightarrow{\rho}{}^* c_1$ again by induction hypothesis. □

Using these two lemmas, we can show that the reduction sequence in $\lambda\widehat{\rho}$ using only (Sub) steps can be simulated in $\lambda\rho$ as well.

**Lemma 3.** *If $c$ and $c'$ are $\lambda\rho$-closures, and $c \xrightarrow{\widehat{\rho}}{}^*_{\mathrm{Sub}} c'$, then $c \xrightarrow{\rho}{}^* c'$.*

*Proof.* By induction on the structure of $c'$, using Lemmas 1 and 2. □

Putting the three lemmas together, we can show that for any $\lambda\rho$-closures $c_0$ and $c_1$ such that $c_0 \stackrel{\widehat{\rho}}{\rightarrow}{}^* c_1$, the reduction sequence can be reordered so that we first follow the normal order strategy and then apply the necessary (Sub)-reductions (by Lemma 1). By Lemmas 2 and 3, each of the two reduction sequences can then be simulated in the $\lambda\rho$-calculus. □

## 2.3 Specification of the normal-order reduction strategy

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

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

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

We consider only closed terms, and hence the side condition on $i$ can be omitted. (We omit it in Sections 3 and 5.1.)

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

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

## 2.4 Specification of the applicative-order reduction strategy

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

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

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

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

We consider only closed terms, and hence the side condition on $i$ can be omitted. (We omit it in Sections 4 and 5.2.)

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

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

Under call by value, both sub-components of any application $c_0\ c_1$ (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.

### 2.5 Correspondence with the $\lambda$-calculus

In order to relate values in the $\lambda$-calculus with values in the language of closures, we define a function $\sigma$ 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 $\lambda$-abstractions), and returns a $\lambda$-term:

$$\sigma(i[s],k) = \begin{cases} i & \text{if } i \leq k \\ \sigma(c_{i-k},k) & \text{if } k < i \leq m+k \text{ and } s = c_1 \cdots c_m \\ i-m & \text{if } i > m+k \text{ and } s = c_1 \cdots c_m \end{cases}$$
$$\sigma((t_0\ t_1)[s],k) = (\sigma(t_0[s],k))\ (\sigma(t_1[s],k))$$
$$\sigma((\lambda t)[s],k) = \lambda(\sigma(t[s],k+1))$$
$$\sigma(c_0\ c_1,k) = \sigma(c_0,k)\ \sigma(c_1,k)$$

# 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 $\lambda$-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 [20]:

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 transition functions.

Section 3.4: We inline the transition 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 coalesce these two steps into one.

This shortcut makes the machine operate in the $\lambda\rho$-calculus instead of the $\lambda\widehat{\rho}$-calculus.

Section 3.7: We unfold the data type of closures, making the $\lambda\rho$ 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 operating in the $\lambda$-calculus. This machine coincides with the Krivine machine [10, 12, 27]. (The original version of Krivine's machine [30, 31] is a bit more complicated, and we treat it in Section 5.)

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

## 3.1 A reduction semantics for normal order

### 3.1.1 Reduction semantics

A reduction semantics [22,23] consists of a grammar of terms from a source language, syntactic notions of value and redex, a collection of contraction rules, and a reduction strategy. This reduction strategy is embodied in a grammar of reduction contexts (terms with a hole as induced by the compatibility rules) and a plug function mapping a term and a context into a new term. One-step reduction of a non-value term consists in

1. decomposing the term into a redex and a reduction context,
2. contracting the redex, and
3. plugging the contractum in the reduction context.

In some reduction semantics, non-value terms are uniquely decomposed into a redex and a context. Decomposition can then be implemented as a function mapping a non-value term to a redex and a reduction context. Danvy and Nielsen have shown that together with the unique decomposition property, the following property of a reduction semantics is sufficient to define a decomposition function by induction on terms and reduction contexts [20, Figure 2, page 8]:

**Property 1.** *For each syntactic construct building a term out of $n$ subterms, there is a number $0 \leq i \leq n$ and a fixed traversal order of subterms encoded in the grammar of the reduction contexts for this construct such that the holes of these contexts are precisely the positions of the $i$ subterms. Furthermore, a term with all the chosen $i$ subterms reduced to values is either a value or a redex,*[1] *but not both.*

Furthermore, if the redexes do not overlap, then the contraction rules can be implemented as a function. This contraction function maps a redex into the corresponding contractum.

[1] More precisely, such a term can be a *potential redex*, i.e., a proper redex or a "stuck term." For simplicity, we omit this issue here, since the reduction strategies we consider do not contain stuck terms.

### 3.1.2 A normal-order reduction semantics for the $\lambda\widehat{\rho}$-calculus

A normal-order reduction semantics for the $\lambda\widehat{\rho}$-calculus can be obtained from the specification of Section 2.3 as follows: the syntactic notion of value and the collection of reduction rules are already specified in Section 2.3, the grammar of redexes reads

$$\text{(Redex)} \qquad r ::= v\ c \mid i[s] \mid (t_0\ t_1)[s],$$

and the compatibility rule ($\nu$) induces the following grammar of reduction contexts (written inside out):

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

For clarity of presentation, in the rest of this article we use the following abstract-syntax notation, where a context $C[[\,]\ c]$ is represented by a tagged pair $\mathsf{ARG}(c,\ C)$:

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

In the present case, all the conditions mentioned in Section 3.1.1 are satisfied; in particular, the traversal order of subterms is leftmost innermost. We can therefore define the following three functions:

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

### 3.1.3 Decomposition

We define $\mathsf{decompose}$ as a transition function mapping a non-value closure to a redex and a context. For simplicity, we generalize this function to arbitrary closures by making it map value closures to themselves:

$$\begin{aligned} \mathsf{decompose} &: \mathsf{Closure} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\ \mathsf{decompose}\ c &= \mathsf{decompose}'\ (c,\ [\,]) \\ \\ \mathsf{decompose}' &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\ \mathsf{decompose}'\ (i[s],\ C) &= (i[s],\ C) \\ \mathsf{decompose}'\ ((\lambda t)[s],\ C) &= \mathsf{decompose}'_{\mathsf{aux}}\ (C,\ (\lambda t)[s]) \\ \mathsf{decompose}'\ ((t_0\ t_1)[s],\ C) &= ((t_0\ t_1)[s],\ C) \\ \mathsf{decompose}'\ (c_0\ c_1,\ C) &= \mathsf{decompose}'\ (c_0,\ \mathsf{ARG}(c_1,\ C)) \\ \\ \mathsf{decompose}'_{\mathsf{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\ \mathsf{decompose}'_{\mathsf{aux}}\ ([\,],\ v) &= v \\ \mathsf{decompose}'_{\mathsf{aux}}\ (\mathsf{ARG}(c,\ C),\ v) &= (v\ c,\ C) \end{aligned}$$

The main decomposition function, $\mathsf{decompose}$, uses two auxiliary transition functions that work according to Property 1:

- $\mathsf{decompose}'$ is passed a closure and a reduction context. It dispatches on the closure and iteratively builds the reduction context:
  - if the current closure is a value, then $\mathsf{decompose}'_{\mathsf{aux}}$ is called to inspect the context;
  - if the current closure is a redex, then a decomposition is found; and

– otherwise, a subclosure of the current closure is chosen to be visited in a new context.

- $\mathsf{decompose'}_{\mathsf{aux}}$ is passed a reduction context and a value. It dispatches on the reduction context: if the current context is empty, then the value is the result of the function; otherwise, the top constructor of the context is analyzed. In the present case, there is only one such constructor and no more subclosures need to be visited since a redex has been found.

### 3.1.4 Contraction

We define contract by cases, as a straightforward implementation of the contraction rules:

$$\begin{aligned}
\mathsf{contract} &: \mathsf{Redex} \to \mathsf{Closure} \\
\mathsf{contract}\,(((\lambda t)[s])\;c) &= t[c \cdot s] \\
\mathsf{contract}\,(i[c_1 \cdots c_m]) &= c_i \\
\mathsf{contract}\,((t_0\;t_1)[s]) &= (t_0[s])\;(t_1[s])
\end{aligned}$$

### 3.1.5 Plugging

We define plug by structural induction over the reduction context. It iteratively peels off the context and thus also takes the form of a transition function:

$$\begin{aligned}
\mathsf{plug} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Closure} \\
\mathsf{plug}\,(c, [\,]) &= c \\
\mathsf{plug}\,(c_0, \mathsf{ARG}(c_1,\; C)) &= \mathsf{plug}\,(c_0\;c_1, C)
\end{aligned}$$

### 3.1.6 One-step reduction

Given these three functions, we can define a one-step reduction function that decomposes a non-value closure into a redex and a context, contracts this redex, and plugs the contractum in the context. For simplicity, we generalize this function to arbitrary closures by making it map value closures to themselves:

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

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

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

### 3.1.7 Reduction-based evaluation

Finally, we can define evaluation using the reflexive, transitive closure of one-step reduction. For simplicity, we use decompose to test whether a value has been reached:

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

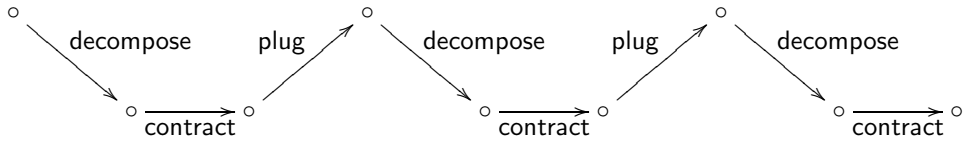
$$\begin{aligned}&\mathsf{evaluate} \;:\; \mathsf{Term} \to \mathsf{Value}\\ &\mathsf{evaluate}\, t = \mathsf{iterate}\,(\mathsf{decompose}\,(t[\bullet]))\end{aligned}$$

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

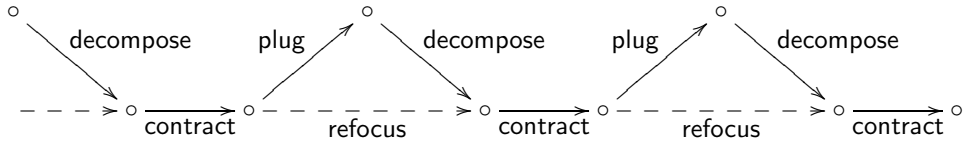
**Proposition 3.** *For any closed term $t$ and any value $v$, $t[\bullet] \xrightarrow{\widehat{\rho}}{}_{\mathrm{n}}{}^{*}\, v \Leftrightarrow \mathsf{evaluate}\, t = v$.*

## 3.2 A pre-abstract machine

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




At each step, an intermediate term is constructed by the function plug; it is then immediately decomposed by the subsequent call to decompose. In earlier work [20], Danvy and Nielsen pointed out that the composition of plug and decompose could be replaced by a more efficient function, refocus, that would directly go from redex site to redex site in the reduction sequence:




The essence of refocusing for a reduction semantics satisfying the unique decomposition property is captured in the following proposition:

**Proposition 4 (Danvy and Nielsen [15, 20]).** *For any closure $c$ and reduction context $C$,*

$$\mathsf{decompose}\,(\mathsf{plug}\,(c,\, C)) = \mathsf{decompose}'\,(c,\, C)$$

The definition of the refocus function is therefore a clone of that of decompose′. In particular, it involves an auxiliary function $\mathsf{refocus}_{\mathsf{aux}}$ and takes the form of two state-transition functions, i.e., of an abstract machine:

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

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

In this abstract machine, the configurations are pairs of a closure and a context; the final transitions are specified by $\mathsf{refocus}_{\mathsf{aux}}$ and by the first and third clauses of 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 iterate, we have exploited the double equality $\mathsf{decompose}(t[\bullet]) = \mathsf{decompose}\, (\mathsf{plug}\, (t[\bullet], [\,])) = \mathsf{refocus}\, (t[\bullet],\ [\,])$.)

This evaluation function computes the transitive closure of refocus using the auxiliary function iterate as a trampoline [26]. Due to the non-tail call to refocus in iterate, we refer to this evaluation function as a 'pre-abstract machine.'

## 3.3 A staged abstract machine

To transform the pre-abstract machine (i.e., the transitive closure of a state-transition function) into an abstract machine (i.e., a state-transition function), we distribute the calls to iterate from the definitions of evaluate and of iterate to the definitions of refocus and $\mathsf{refocus}_{\mathsf{aux}}$:

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

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

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

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

The resulting definitions of evaluate, iterate, refocus, and $\mathsf{refocus}_{\mathsf{aux}}$ 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 evaluate, and the final transition in the first clause of iterate. The compatibility rules are implemented by refocus and $\mathsf{refocus}_{\mathsf{aux}}$, and the reduction rules by the call to contract in the second clause of iterate. We can make this last point even more manifest by inlining contract in the definition of 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 (with iterate) and the compatibility rules (with refocus and $\mathsf{refocus}_{\mathsf{aux}}$); for this reason, we refer to it as a 'staged abstract machine.'

## 3.4 An eval/apply abstract machine

As already observed by Danvy and Nielsen in their work on refocusing, inlining iterate yields an eval/apply abstract machine [35]. Inlining the calls to iterate in the staged abstract machine yields the following eval/apply machine, where refocus (the 'eval' transition function) dispatches on closures and $\mathsf{refocus}_{\mathsf{aux}}$ (the 'apply' function) dispatches on contexts:

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

## 3.5 A push/enter abstract machine

As already observed by Ager et al. [3], inlining the apply transition function in a call-by-name eval/apply abstract machine yields a push/enter machine: when a function is entered, its arguments are immediately available on the stack [35]. Inlining the calls to $\mathsf{refocus}_{\mathsf{aux}}$ in the eval/apply abstract machine yields the following machine:

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

## 3.6 A push/enter abstract machine for the $\lambda\rho$-calculus

The abstract machine of Section 3.5 only produces an application of closures through an (App) reduction step (second-to-last clause of refocus). We observe that in a reduction sequence, an (App) reduction step is always followed by a decomposition step (last clause of refocus). As a shortcut, we coalesce the two consecutive transitions into one, replacing the last two clauses of 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 resulting machine never produces any application 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 for the $\lambda$-calculus

Finally, we unfold the data type of closures. If we read each syntactic category in $\lambda\rho$ 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 Closure yields $\mathsf{Term} \times \mathsf{Substitution}$. Therefore, for any closure $t[s]$ of type Closure, its unfolding gives a pair $(t,\ s)$ of type $\mathsf{Term} \times \mathsf{Substitution}$. We replace each closure in the definition of evaluate and 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:

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

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

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

We observe that this machine coincides with the Krivine machine [10,12,27], in which evaluation contexts are treated as last-in, first-out lists (i.e., stacks) of closures. In particular, the substitution component assumes the role of the environment.

Therefore, unfolding the data type of closures crystallizes the connection between explicit substitutions in calculi and environments in abstract machines.

## 3.8 Correctness

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

**Theorem 1.** *For any closed 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 [20], and the (trivial) meaning preservation of each of the subsequent transformations. □

The theorem states that the Krivine machine is correct in the sense that it computes closed weak head normal forms and that it realizes the normal-order strategy in the $\lambda\widehat{\rho}$-calculus, which makes it a call-by-name machine [36]. 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 according to the normal-order strategy in $\lambda\rho$ can be simulated in $\lambda\widehat{\rho}$ (see Proposition 1), as a byproduct we obtain the correctness of the Krivine machine also with respect to Curien's original calculus of closures:

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

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

### 3.9 Correspondence with the $\lambda$-calculus

The Krivine machine is generally presented as an environment machine for call-by-name evaluation in the $\lambda$-calculus. The following theorem formalizes this correspondence using the substitution function $\sigma$ defined in Section 2.5.

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

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

*Proof.* Both implications rely on Corollary 1. The left-to-right implication relies on the following property, proved by structural induction on $t$:

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

In order to prove the converse implication, we observe that if $c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'$, then either $\sigma(c, 0) \to_{\mathrm{n}} \sigma(c', 0)$, if the $(\beta)$ rule is applied 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 $\lambda$-calculus with de Bruijn indices in much the same way as we do above. In fact, our substitution function $\sigma$ performs exactly $\sigma$-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 closures to $\lambda$-terms with names that is an analog of $\sigma$, and presented a similar simple proof of the correctness of the Krivine machine for the $\lambda$-calculus with names [44].

## 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 left-to-right 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 $(\nu)$ and $(\mu)$ induce the following grammar of reduction contexts:

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

Again, for clarity we use an abstract-syntax notation for reduction contexts, extending the one of Section 3.1 with $\mathsf{FUN}(v,\ C)$ for $C[v\ [\,]]$:

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

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 Property 1 and the corresponding refocus function is therefore defined as follows:

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

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

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:

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

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

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

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 an application 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 [23].

Unlike in Section 3.5, inlining $\mathsf{refocus}_{\mathsf{aux}}$ does not yield a push/enter architecture because when a function is entered, its arguments are not immediately available on the stack but still have to be evaluated. In addition, the resulting transition function dispatches on all its arguments, like the SECD machine [17, 32].

### 4.3 Correctness and correspondence with the $\lambda$-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 [36]. 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 generally presented as an environment machine for call-by-value evaluation in the $\lambda$-calculus. The following theorem formalizes this correspondence, using the substitution function $\sigma$ defined in Section 2.5.

**Theorem 4 (Correspondence).** *For any $\lambda$-term $t$, $t \to_{\mathrm{v}}^{*} \lambda t'$ 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

The Krivine machine, as usually presented in the literature [1, 3, 10–12, 20, 25, 27–29, 34, 42, 44], contracts one $\beta$-redex at a time. The original version [30, 31], however, grammatically distinguishes nested $\lambda$-abstractions and contracts nested $\beta$-redexes in one step. Similarly, Leroy's Zinc machine [33] optimizes curried applications.

Krivine's language of $\lambda$-terms reads as follows:

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

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

In Krivine's machine, and using the same notation as in Section 3, (nested) $\beta$-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 $\beta$-reduction not just for the pair of terms forming a $\beta$-redex, or even for a tuple of terms forming a nested $\beta$-redex, but *for a nested $\lambda$-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 Krivine machine arises, how the original version of Krivine's machine also arises, and how one can derive a slightly more perspicuous version of this original version, based on an observation due to Wand. 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 ($\nu$), just as in Section 3.1:

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

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

$$(\beta^+) \qquad ((\lambda^n t)[s],\ \mathsf{ARG}(c_1,\ \ldots \mathsf{ARG}(c_n,\ C)\ldots)) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t[c_n \cdots c_1 \cdot s],\ C)$$

$$\text{(Var)} \qquad (i[c_1 \cdots c_m],\ C) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (c_i,\ C)$$

$$\text{(App)} \qquad ((t_0\ t_1)[s],\ C) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} ((t_0[s])\ (t_1[s]),\ C)$$

The new ($\beta^+$) rule is the only reduction rule that actually depends on the context (i.e., the context remains unchanged in the other two rules).

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 bypass the construction of closure application in (App) and directly construct the reduction context obtained in Section 3.6:

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

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

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

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 ($\beta^+$).

### 5.1.1 The Krivine machine

Here, for each application 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 Krivine machine.

Therefore, the Krivine machine can be obtained in three ways:

1. using a context-insensitive reduction semantics in $\lambda\widehat{\rho}$ (as in Section 3),
2. using a context-sensitive reduction semantics in $\lambda\widehat{\rho}$, as in the present section, and
3. using a context-sensitive reduction semantics in $\lambda\rho$, as in the present section.

Similarly, there are three ways of obtaining an abstract machine for right-to-left applicative order.

### 5.1.2 The original version of Krivine's machine

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

We then take the same steps as in Section 3, and obtain the same machine as Krivine [30,31]. As pointed out by Wand [44], however, since the number of arguments is required to match the number of nested $\lambda$-abstractions, the machine becomes stuck if there are not enough arguments in the context, even though a weak head normal form exists. We handle this case by adapting the $\beta^+$-rule as described next.

### 5.1.3 An adjusted version of Krivine's machine

Here, for each application we choose $n$ to be the smallest number between the arity of each nested $\lambda$-abstraction and the number of nested applications.[2] (So, for example, $n = 1$ for $(\lambda\lambda t,\, \mathsf{ARG}(c_1,\, [\,]))$.)

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 Right-to-left applicative order: variants of the Zinc machine

From Landin [32] to Leroy [33], 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 ($\nu$) and ($\mu$), for right-to-left applicative order, induce the following grammar of contexts:

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

As in Sections 3.1 and 4.1, we introduce a more convenient abstract-syntax notation for reduction contexts:

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

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; $C[[\,]v]$ is written $\mathsf{ARG}(v,\, C)$ and $C[c[\,]]$ is written $\mathsf{FUN}(c,\, C)$.

[2] Alternatively, we can handle this case by stating the $\beta^+$-rule as follows:

$$(\beta^+)\quad ((\lambda^n t)[s],\, \mathsf{ARG}(c_1,\, \ldots \mathsf{ARG}(c_{n'},\, C))) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} ((\lambda^{n''} t)[c_{n''} \cdots c_1 \cdot s],\, \mathsf{ARG}(c_{n''+1},\, \ldots \mathsf{ARG}(c_{n'},\, C)))$$

where $n'' = min(n, n')$.

The notion of reduction is defined through the following rules:

$$(\beta^+) \qquad ((\lambda^n t)[s],\, \mathsf{ARG}(v_1,\, \ldots \mathsf{ARG}(v_n,\, C)\ldots)) \xrightarrow{\widehat{\rho}}_{\mathrm{v}} (t[v_n \cdots v_1 \cdot s],\, C)$$

$$(\text{Var}) \qquad (i[v_1 \cdots v_m],\, C) \xrightarrow{\widehat{\rho}}_{\mathrm{v}} (v_i,\, C)$$

$$(\text{App}) \qquad ((t_0\, t_1)[s],\, C) \xrightarrow{\widehat{\rho}}_{\mathrm{v}} ((t_0[s])\,(t_1[s]),\, C)$$

Similarly to the call-by-name case, the only context-sensitive reduction rule is $(\beta^+)$ (we specify $n$ as in Section 5.1.3).

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 replace (App) with a context-sensitive rule as follows:

$$(\text{App}') \qquad ((t_0\, t_1)[s],\, C) \xrightarrow{\widehat{\rho}}_{\mathrm{v}} (t_1[s],\, \mathsf{FUN}(t_0[s],\, C)).$$

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 Property 1 and the corresponding refocus function is defined as follows:

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

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

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:

$$\begin{aligned}
\mathsf{evaluate}\, t &= \mathsf{refocus}\,(t,\, \bullet,\, [\,]) \\
\mathsf{refocus}\,(i,\, (t_1,\, s_1) \cdots (t_m,\, s_m),\, C) &= \mathsf{refocus}\,(t_i,\, s_i,\, C) \\
\mathsf{refocus}\,(\lambda^n t,\, s,\, [\,]) &= (\lambda^n t,\, s) \\
\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{FUN}((t',\, s'),\, C)) &= \mathsf{refocus}\,(t',\, s',\, \mathsf{ARG}((\lambda^n t,\, s),\, C)) \\
\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{ARG}(v_1,\, \ldots \mathsf{ARG}(v_n,\, C)\ldots)) &= \mathsf{refocus}\,(t,\, v_n \cdot \ldots \cdot v_1 \cdot s,\, C) \\
\mathsf{refocus}\,(t_0\, t_1,\, s,\, C) &= \mathsf{refocus}\,(t_1,\, s,\, \mathsf{FUN}((t_0,\, s),\, C))
\end{aligned}$$

This machine corresponds to an instance of Leroy's Zinc machine for the pure $\lambda$-calculus [33, Chapter 3], with the proviso that it operates directly on $\lambda$-terms instead of over an instruction set (which has been said to be the difference between an abstract machine and a virtual machine [2]). Moreover, in the Zinc machine the sequence of values on the stack (denoted here by $\mathsf{ARG}(v_1,\, \ldots \mathsf{ARG}(v_n,\, C)\ldots)$) 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 [38, 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 the Krivine machine with the following clause for the case where the actual parameter is a variable:

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

We observe that circumventing the space leak has an analogue in the normal-order reduction semantics: it corresponds to adding the following contraction 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 $\lambda$-term towards its weak head normal form.

**The context-insensitive reduction semantics:** Taking the same steps as in Section 3 mechanically leads one to the Krivine 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 the Krivine machine [25].

**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 ($\lambda$). 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.

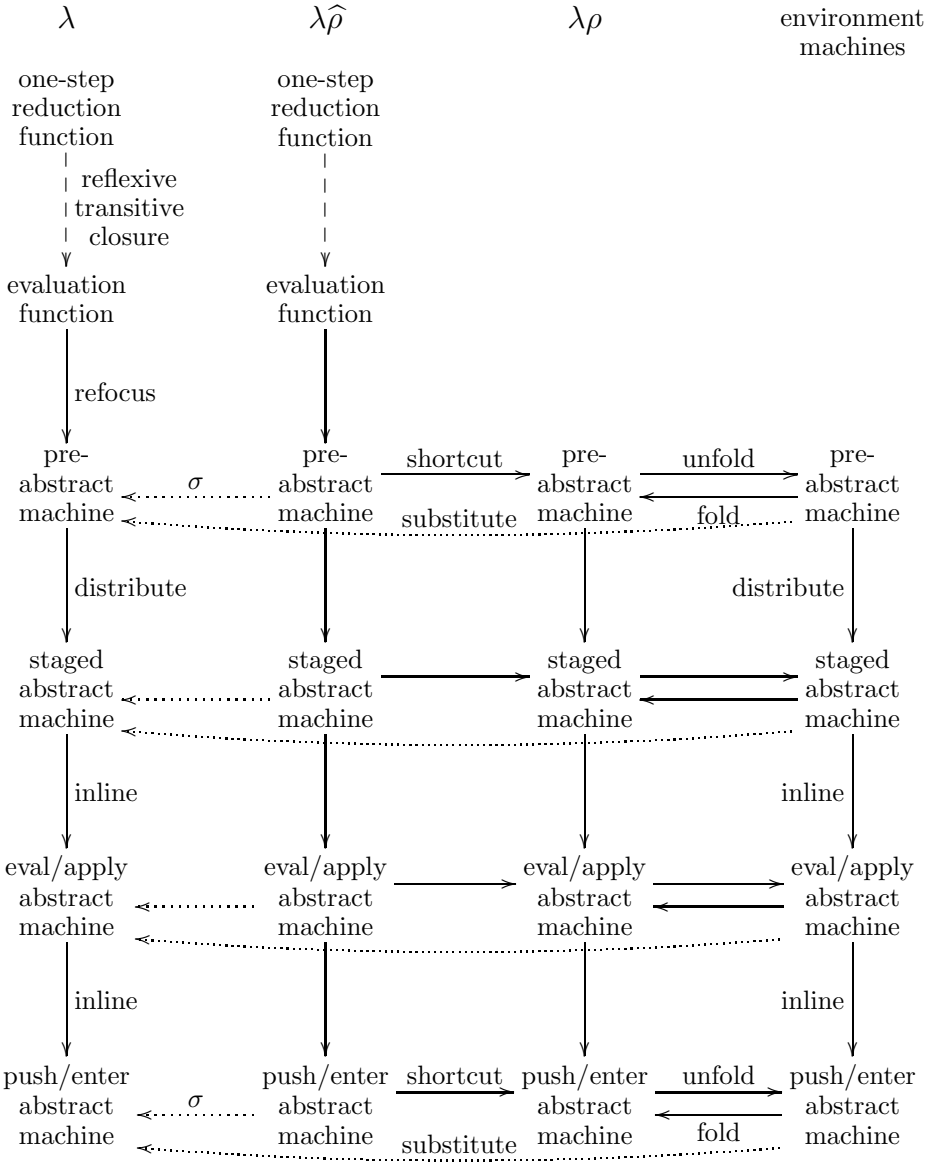



Figure 1: A derivational taxonomy of abstract machines

Reading down each column follows Danvy and Nielsen's work on refocusing [20]. They show how to go from a reduction-based evaluation function (obtained as the reflexive-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 $4 \times 4$-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 and written $\sigma$ in Section 2.5), we can go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for $\lambda$-terms. To go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for $\lambda\rho$-closures, we coalesce the (App) reduction step with the subsequent 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 (using $\sigma$ again) 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 $\lambda$-terms.

In their original presentation of refocusing [20], Danvy and Nielsen considered substitution-based machines and followed the first column in the diagram. In Sections 3 and 5, the derivations follow the second column all the way down to a push/enter machine, and then across the columns to the right, to the corresponding push/enter environment machine. In Section 4, we go down the second column to an eval/apply abstract machine, and then across the columns to the right, to the corresponding eval/apply environment machine.

In their original presentation of refocusing [20], 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 [23]. In Section 3, we observed that the push/enter environment machine corresponding to normal-order reduction coincides with the Krivine machine [10, 12, 27]. 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 [23]. In Section 5, we observed that a context-sensitive reduction semantics gives rise to the original version of Krivine's machine [31] and to the Zinc machine [33]. In Section 6, we observed that the space optimization of two families of call-by-name machines corresponds to two reduction semantics. Obtaining staged abstract machines was one of the goals of Hardin, Maranget, and Pagano's study of functional runtime systems using explicit substitutions [29]; these machines arise mechanically here.

Each of the machines in Figure 1 is thus of independent value. Furthermore, and as investigated by Danvy and his students in their study of the functional correspondence between compositional evaluation functions and abstract machines [3–5,17], the eval/apply machines are in defunctionalized form [19, 39] and they can be 'refunctionalized' into a continuation-passing evaluation function which itself can be written in direct style [14]. This direct-style evaluation function implements a big-step operational semantics. Because of the closures, the result is again in defunctionalized form and can be refunctionalized into an evaluation function. It turns out that the resulting evaluation functions are compositional and therefore each of them implements the valuation function of a denotational semantics, where environments are also used. Put together, the syntactic correspondence between calculi and abstract machines presented here and the functional correspondence between intensional abstract machines and extensional evaluation functions therefore pave the way from reduction-based to reduction-free evaluation [15].

### 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 [29]. 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 an arbitrary abstract machine to a reduction semantics [7, 24].

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 [1,13,29,34,41,42] 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 correctness of the resulting machines is ensured by the correctness of the derivation method. The derivation is based on Danvy and Nielsen's refocusing technique, 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 application, which results in the $\lambda\widehat{\rho}$-calculus.

We have illustrated the concrete framework by uniformly deriving several independently known environment machines—the Krivine machine, the original version of Krivine's machine, 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) crystallizes the connection between calculi of explicit substitutions and environment machines.

In a further work [8], we have used the concrete framework to study context-sensitive calculi of explicit substitutions for a number of impure cases: first-class continuations, delimited continuations, i/o, stack inspection, proper tail-recursion, lazy evaluation, and the $\lambda\mu$-calculus. We have found that the concrete framework scales up seamlessly and provides a syntactic correspondence between context-sensitive calculi of explicit substitutions and environment machines accounting for computational effects.

**Acknowledgments:** Thanks are due to Mads Sig Ager, Dariusz Biernacki, Pierre-Louis Curien, Mayer Goldberg, Julia Lawall, Jan Midtgaard, Kevin Millikin, Kristian Støvring, David Van Horn, and the anonymous reviewers for comments; to Xavier Leroy for a sanity check; and to Ulrich Kohlenbach for providing us with what appears to be the original bibliographic reference of the logic counterpart of environments [43, § 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. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] 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.

[3] 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.

[4] 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.

[5] 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 technical report BRICS RS-04-28.

[6] 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.

[7] 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).

[8] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Research Report BRICS RS-05-38, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2005.

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

[10] Pierre Crégut. An abstract machine for lambda-terms normalization. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[11] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. In Danvy [18]. To appear. Journal version of [10].

[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. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, number 124 in Electronic Notes in Theoretical Computer Science, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[16] 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 (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[17] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, 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 prize. Extended version available as the technical report BRICS RS-03-33.

[18] Olivier Danvy, editor. *Special Issue on the Krivine Abstract Machine*, Higher-Order and Symbolic Computation. Springer, 2006. In preparation.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Lynn Winebarger. Improving the lazy Krivine machine. In Danvy [18]. In preparation.

[26] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 18–27, Paris, France, September 1999. ACM Press.

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

[28] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

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

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

[31] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In Danvy [18]. To appear. Available online at `http://www.pps.jussieu.fr/~krivine/`.

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

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

[34] Pierre Lescanne. From $\lambda\sigma$ to $\lambda v$ 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.

[35] 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*, SIGPLAN Notices, Vol. 39, No. 9, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

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

[37] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

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

[39] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [40].

[40] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[41] 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.

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

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

[44] Mitchell Wand. On the correctness of the Krivine machine. In Danvy [18]. In preparation.

# Recent BRICS Report Series Publications

**RS-06-3** **Małgorzata Biernacka and Olivier Danvy.** ***A Concrete Framework for Environment Machines*****. February 2006. ii+29 pp. To appear in the** ***ACM Transactions on Computational Logic*****. Supersedes BRICS RS-05-15.**

**RS-06-2** **Mikkel Baun Kjærgaard and Jonathan Bunde-Pedersen.** ***A Formal Model for Context-Awareness*****. February 2006. 26 pp.**

**RS-06-1** **Luca Aceto, Taolue Chen, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***On the Axiomatizability of Priority*****. January 2006. 25 pp.**

**RS-05-38** **Małgorzata Biernacka and Olivier Danvy.** ***A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*****. December 2005. iii+39 pp. Revised version of BRICS RS-05-22.**

**RS-05-37** **Gerth Stølting Brodal, Kanela Kaligosi, Irit Katriel, and Martin Kutz.** ***Faster Algorithms for Computing Longest Common Increasing Subsequences*****. December 2005. 16 pp.**

**RS-05-36** **Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan.** ***On the Static and Dynamic Extents of Delimited Continuations*****. December 2005. ii+33 pp. To appear in the journal** ***Science of Computer Programming*****. Supersedes BRICS RS-05-13.**

**RS-05-35** **Kristian Støvring.** ***Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative*****. November 2005. 19 pp. To appear in** ***Logical Methods in Computer Science*****.**

**RS-05-34** **Henning Korsholm Rohde.** ***Formal Aspects of Polyvariant Specialization*****. November 2005. 27 pp.**

**RS-05-33** **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Sumit Nain.** ***Bisimilarity is not Finitely Based over BPA with Interrupt*****. October 2005. 33 pp. This paper supersedes BRICS Report RS-04-24. An extended abstract of this paper appeared in** ***Algebra and Coalgebra in Computer Science, 1st Conference, CALCO 2005*****, Swansea, Wales, 3–6 September 2005, Lecture Notes in Computer Science 3629, pp. 54–68, Springer-Verlag, 2005.**

**RS-05-32** **Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach.** ***Static Validation of XSL Transformations*****. October 2005. 50 pp.**