A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines
A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines
Małgorzata Biernacka and Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

December 27, 2005

Abstract

We present a systematic construction of environment-based abstract machines from context-sensitive calculi of explicit substitutions, and we illustrate it with ten calculi and machines for applicative order with an abort operation, normal order with generalized reduction and call/cc, the lambda-mu-calculus, delimited continuations, stack inspection, proper tail-recursion, and lazy evaluation. Most of the machines already exist but have been obtained independently and are only indirectly related to the corresponding calculi. All of the calculi are new and they make it possible to directly reason about the execution of the corresponding machines.

In connection with the functional correspondence between evaluation functions and abstract machines initiated by Reynolds, the present syntactic correspondence makes it possible to construct reduction-free normalization functions out of reduction-
based ones, which was an open problem in the area of normalization by evaluation.

Revised version of BRICS RS-05-22.
Contents

1 Introduction
  1.1 Calculi and machines .................................. 1
  1.2 Calculi of explicit substitutions and environment-based machines ............ 2
  1.3 Calculi of explicit substitutions for computational effects and environment-based machines ........................................ 2
  1.4 Overview ............................................ 2

2 Preliminaries ............................................. 3
  2.1 Reduction semantics ..................................... 3
  2.2 Context-sensitive reduction ................................ 3
  2.3 Our base calculus of closures: \( \lambda \rho \) ........................................ 4

3 The \( \lambda \rho A \)-calculus .................................. 5
  3.1 A reduction semantics for applicative order and abort ........................... 6
    3.1.1 Decomposition ..................................... 7
    3.1.2 Context-sensitive contraction .................................... 7
    3.1.3 Plugging ........................................ 8
3.1.4 One-step reduction

3.1.5 Reduction-based evaluation

3.2 A pre-abstract machine

3.3 A staged abstract machine

3.4 An eval/apply abstract machine

3.5 Transition compression

3.6 An environment machine

3.7 Correctness

3.8 Conclusion

4 The $\lambda\hat{\rho}\mathcal{K}$-calculus

4.1 The language of $\lambda\hat{\rho}\mathcal{K}$
4.2 Notion of context-sensitive reduction.
4.3 Krivine's machine.
4.4 Formal correspondence.

5 The \( \lambda \rho \mu \)-calculus
5.1 The language of \( \lambda \rho \mu \).
5.2 Notion of context-sensitive reduction.
5.3 An eval/apply abstract machine.
5.4 Formal correspondence.

6 Delimited continuations
6.1 The \( \lambda \rho S \)-calculus.
6.1 The language of $\lambda\rho S$ ................................. 17
6.1.2 The eval/apply/meta-apply abstract machine .......................... 17
6.1.3 Notion of context-sensitive reduction ................................. 18
6.1.4 Formal correspondence ............................................ 18
6.1.5 The CPS hierarchy .................................................. 18
6.2 The $\lambda\rho F$-calculus .............................................. 19
6.2.1 The language of $\lambda\rho F$ ............................................ 19

6.2.2 Notion of context-sensitive reduction ................................. 19
6.2.3 The eval/apply abstract machine ..................................... 19
6.2.4 Formal correspondence .............................................. 19
6.2.5 A hierarchy of control delimiters ................................... 19
6.3 Conclusion ...........................................................
7 Stack inspection

7.1 The $\lambda\hat{\rho}_{\text{sec}}$-calculus

7.1.1 The language of $\lambda\hat{\rho}_{\text{sec}}$

7.1.2 Notion of context-sensitive reduction

19

20

20

20
7.1.3 An eval/apply abstract machine .................................................. 23
7.1.4 Formal correspondence ............................................................... 23
7.2 Properly tail-recursive stack inspection .......................................... 23
  7.2.1 The storeless cm machine ......................................................... 24
  7.2.2 The underlying calculus $\lambda \hat{\rho}_{sec}^{cm}$ .................................. 24
7.3 State-based properly tail-recursive stack inspection ........................... 25
  7.3.1 The unzipped storeless cm machine ......................................... 25
  7.3.2 The language of $\lambda \hat{\rho}_{sec}^{ucm}$ ........................................... 26
  7.3.3 Notion of context-sensitive reduction ........................................ 26
  7.3.4 Formal correspondence ............................................................ 27
8 A calculus for proper tail-recursion
8.1 A simplified version of Clinger’s abstract machine
8.2 The language of $\lambda\hat{\rho}_{ptr}$
8.3 Notion of context-sensitive reduction
8.4 Formal correspondence

9 A lazy calculus of closures
9.1 The language of $\lambda\hat{\rho}_1$
9.2 Notion of context-sensitive reduction
9.3 An eval/apply abstract machine
9.4 Formal correspondence

10 Conclusion
1 Introduction

How does one construct a new semantic artifact?

1.1 Calculi and machines

Sixty-five years ago, the λ-calculus was introduced [15]. Forty-five years ago, its expressive power was observed to be relevant for computing [67, 81]. Forty years ago, a first abstract machine for the λ-calculus was introduced [61]. Thirty years ago, calculi and abstract machines were formally connected [70]. Twenty years ago, a calculus format—reduction semantics—with an explicit representation of reduction contexts was introduced [40]. Today calculi and abstract machines are standard tools to study programming languages. Given a calculus, it is by now a standard activity to design a corresponding abstract machine and to prove its correctness [42].
From calculus to machine by refocusing and transition compression: Recently, Danvy and Nielsen have pointed out that the reduction strategy for a calculus actually determines the structure of the corresponding machine [36]. They present a method for constructing an abstract machine out of a reduction semantics satisfying the unique-decomposition property. In such a reduction semantics, a non-value term is reduced by

1. decomposing it (uniquely) into a redex and its context,
2. contracting the redex, and
3. plugging the contractum into the reduction context,

yielding a new term. A reduction-based evaluation function is defined using the reflexive and transitive closure of the one-step reduction function:

Danvy and Nielsen have observed that the intermediate terms, in the composition of plug and decompose, could be avoided by fusing the composition into a ‘refocus’ function:
The resulting ‘refocused’ evaluation function is defined as the reflexive and transitive closure of refocusing and contraction.

The refocus function takes the form of a state-transition function, i.e., an abstract machine. The refocused evaluation function therefore also takes the form of an abstract machine. Compressing its intermediate transitions (i.e., short-circuiting them) yields abstract machines that are often independently known: for example, for the pure $\lambda$-calculus with normal order, the resulting abstract machine is a substitution-based version of the Krivine machine (i.e., a push/enter machine); for the pure $\lambda$-calculus with applicative order, the resulting abstract machine is Felleisen et al.’s CK machine (i.e., an eval/apply machine). Refocusing has also been applied to the term language of the free monoid, yielding a reduction-free normalization function [26], and to context-based CPS transformations, improving them from quadratic time to operating in one pass [36].

1.2 Calculi of explicit substitutions and environment-based machines

Twenty years ago, Curien observed that while most calculi use actual substitutions, most implementations use closures and environments [22]. He then developed a calculus of closures, $\lambda\rho$ [23], thereby launching the study of explicit substitutions [1, 24, 51, 65, 75].

From calculus to machine by refocusing, transition compression, and closure
unfolding: Recently, we have applied the refocusing method to $\lambda\hat{\rho}$, a minimal extension of $\lambda\rho$ where one can express single-step computations; we added an unfolding step to make the machine operate not on a closure, but on a term and its environment [8]. We have shown how $\lambda\hat{\rho}$ with left-to-right applicative order directly corresponds to the CEK machine [43], how $\lambda\hat{\rho}$ with normal order directly corresponds to the Krivine machine [20, 23], how $\lambda\hat{\rho}$ with generalized reduction directly corresponds to the original version of Krivine’s machine [59], and how $\lambda\hat{\rho}$ with right-to-left applicative order directly corresponds to the ZINC abstract machine [64]. All of these machines are environment-based and use closures.

1.3 Calculi of explicit substitutions for computational effects and environment-based machines

Twenty years ago, Felleisen introduced reduction semantics—a version of small-step operational semantics with an explicit representation of reduction contexts—in order to provide calculi for control and state [40, 43]. In these calculi, reduction rules are not oblivious to their reduction context; on the contrary, they are context sensitive in that the context takes part in some reduction steps, e.g., for call/cc. Reduction semantics are in wide use today, e.g., to study the security technique of stack inspection [16, 47, 71].

From calculus to machine by refocusing, transition compression, and closure unfolding: In this article, we apply the refocusing method to context-sensitive extensions of $\lambda\hat{\rho}$ accounting for a variety of computational effects. We present ten calculi of closures and the corresponding environment-based machines. What is significant here is
that each machine is mechanically derived from the corresponding calculus (instead of designed and then proved correct) and also that each machine directly corresponds to this calculus (instead of indirectly via an ‘unload’ function at the end of each run [70] or via a compilation / decompilation scheme in the course of execution [51]).

1.4 Overview

We successively consider call by name: Krivine’s machine with call/cc (Section 4) and the λμ-calculus (Section 5); call by value: static and dynamic delimited continuations (Section 6), stack inspection (Section 7), and proper tail-recursion (Section 8); and call by need (Section 9). Towards this end, we first present the λρ-calculus and the notion of context-sensitive reduction (Section 2) and then a detailed walkthrough for the λρ-calculus with an abort operation (Section 3). The subsequent repetitiveness of Sections 4 to 9 (language, notion of context-sensitive reduction, abstract machine, and formal correspondence) is a deliberate feature, not an inadvertent presentational bug: we apply the same simple method to many situations that have been separately studied so far. Each of these sections can therefore be read independently of the others.

2 Preliminaries

2.1 Reduction semantics
A reduction semantics \cite{40, 42} 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 \cite[Figure 2, page 8]{36}:

**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 potential redex (i.e., an actual redex or a “stuck term”) but not both.

If the redexes do not overlap, the contraction rules can be implemented as a function.
2.2 Context-sensitive reduction

Traditional specifications of one-step reduction as the compatible closure of a notion of reduction provide a local characterization of a computation step in the form of a redex.\(^1\) This local characterization is not fit for non-local reductions such as one involving a control operator capturing all its surrounding context in one step, or a global state. For these, one needs a notion of context-sensitive reduction, i.e., a binary relation defined both on redexes and on their reduction context instead of only on redexes [40].

This relation is given by context-sensitive contraction rules of the form \(\langle r, C \rangle \rightarrow \langle c', C' \rangle\), where \(\langle r, C \rangle\) denotes the decomposition of a program into a potential redex \(r\) and its context \(C\).

A one-step reduction relation for a given notion of context-sensitive reduction is defined as follows: A program \(p\) reduces in one step to \(p'\) if decomposing \(p\) yields \(\langle r, C \rangle\), reducing \(\langle r, C \rangle\) yields \(\langle c', C' \rangle\), and plugging \(c'\) into \(C'\) yields \(p'\).

A context-sensitive reduction implicitly assumes a decomposition of the entire program, and therefore it cannot be used locally. One way to recover compatibility in the context-sensitive setting is to add explicit local control delimiters to the language (see Section 6 for an illustration). For a language without explicit control delimiters (as the \(\lambda\hat{\rho}\)-calculus with call/cc), there is an implicit global control delimiter around the program [41].

\(^1\)For example, a potential redex in the \(\lambda\)-calculus is the application of a value to a term. If the value is a \(\lambda\)-abstraction, the potential redex is an actual one and it can be \(\beta\)-reduced. If no reduction rule applies, the potential redex is not an actual one and the program is stuck [70].
2.3 Our base calculus of closures: $\lambda\hat{\rho}$

Since Landin [61], most abstract machines implementing variants and extensions of the $\lambda$-calculus use closures and environments, and the substitution of terms for free variables is thus delayed until a variable is reached in the evaluation process. This implementation technique has motivated the study of calculi of explicit substitutions [1, 23, 75] to mediate between the traditional abstract specifications of the $\lambda$-calculus and its traditional concrete implementations [51].

To derive an abstract machine for evaluating $\lambda$-terms, a weak calculus of explicit substitutions suffices. The first (and simplest) of such calculi was Curien's calculus of closures $\lambda\rho$ [23]. Although this calculus is not expressive enough to model full normalization, it is suitable for evaluating a $\lambda$-term, i.e., to produce the corresponding weak head normal form. Its operational semantics is specified using multi-step reductions, but its syntax is too restrictive to allow single-step computations, which is what we need to apply the refocusing method. For this reason, in our earlier work [8], we have proposed a minimal extension of $\lambda\rho$ with one-step reduction rules, the $\lambda\hat{\rho}$-calculus.

The language of $\lambda\hat{\rho}$ is as follows:

(terms) $ t ::= i \mid \lambda t \mid t \ t $

(closures) $ c ::= t[s] \mid c \ c $

(substitutions) $ s ::= \bullet \mid c \cdot s $
(For comparison, $\lambda \rho$ does not have the $cc$ production.)

We use de Bruijn indices for variables in a term ($i \geq 1$). A closure is a term equipped with a substitution, i.e., a list of closures to be substituted for free variables in the term. Programs are closures of the form $t[\bullet]$ where $t$ does not contain free variables.

The notion of reduction in the $\lambda \rho$-calculus is given by the following rules:

(Var)  \[ i[c_1 \cdots c_j] \xrightarrow{\rho} c_i \text{ if } i \leq j \]

(Beta)  \[ ((\lambda t)[s]) c \xrightarrow{\rho} t[c \cdot s] \]

(Prop)  \[ (t_0 t_1)[s] \xrightarrow{\rho} (t_0[s])(t_1[s]) \]

We write $s(i)$ for the $i$th element of the substitution $s$ considered as a list. (So $[c_1 \cdots c_j](i) = c_i$ if $1 \leq i \leq j$.)

Finally, the one-step reduction relation (i.e., the compatible closure of the notion of reduction) extends the notion of reduction with the following rules:

(L-Comp)  \[ \frac{c_0 \xrightarrow{\rho} c'_0}{c_0 c_1 \xrightarrow{\rho} c'_0 c_1} \]

(R-Comp)  \[ \frac{c_1 \xrightarrow{\rho} c'_1}{c_0 c_1 \xrightarrow{\rho} c_0 c'_1} \]

(Sub)  \[ \frac{c_i \xrightarrow{\rho} c'_i}{t[c_1 \cdots c_i \cdots c_j] \xrightarrow{\rho} t[c_1 \cdots c'_i \cdots c_j]} \text{ for } i \leq j \]
Specific, deterministic reduction strategies can be obtained by restricting the compatibility rules. In the following sections, we consider two such strategies:

1. the normal-order strategy obtained by discarding the (Sub) and (R-Comp) rules, where the grammars of values, redexes and reduction contexts are specified as follows:

   (values) \( v ::= (\lambda t)[s] \)

   (redexes) \( r ::= v c \mid i[s] \mid (t_0 \ t_1)[s] \)

   (reduction contexts) \( C ::= [] \mid C[[ ]\ c] \),

and

2. the left-to-right applicative-order strategy obtained in the usual way by discarding the (Sub) rule and restricting the (Beta) and (R-Comp) rules, where the grammars of values, redexes and reduction contexts are specified as follows:

   (values) \( v ::= (\lambda t)[s] \)

   (redexes) \( r ::= v v \mid i[s] \mid (t_0 \ t_1)[s] \)

   (reduction contexts) \( C ::= [] \mid C[[ ]\ c] \mid C[v \ ][] \)
All of the calculi presented in this article extend the $\lambda\rho$-calculus. For each of them, we define a suitable notion of reduction, denoted $\rightarrow_X$, where $X$ is a subscript identifying a particular calculus. For each of them, we then define a one-step reduction relation as the composition of: decomposing a non-value closure into a redex and a reduction context, contracting a (context-sensitive) redex, and then plugging the resulting closure into the resulting context. Finally, we define the evaluation relation (denoted $\rightarrow^*_X$) using the reflexive, transitive closure of one-step reduction, i.e., we say that $c$ evaluates to $c'$ if $c \rightarrow^*_X c'$ and $c'$ is a value closure. We define the convertibility relation between closures as the smallest equivalence relation containing $\rightarrow^*_X$. If two closures $c$ and $c'$ are convertible, they behave similarly under evaluation (i.e., either they both evaluate to the same value, or they both diverge).

3 The $\lambda\rho A$-calculus

As an illustration, we present a detailed and systematic derivation of an abstract machine for call-by-value evaluation in the $\lambda$-calculus with an abort operation, starting from the specification of the applicative-order reduction strategy in the $\lambda\rho$-calculus with an abort operation. We follow the steps outlined by Biernacka, Danvy, and Nielsen [8, 36]:

Section 3.1: We specify the applicative-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
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 $\lambda\rho$-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: Observing that in a reduction sequence, an (App) reduction step is always followed by a decomposition step, we coalesce these two steps into one. Observing that in a reduction sequence, an (Abort) reduction step is always followed by a decomposition step, we coalesce these two steps into one. This shortcut makes the resulting abstract machine dispatch on terms rather than on closures, and enables the following step.

Section 3.6: We unfold the data type of closures, making the abstract machine operate over two components—a term and a substitution—instead of over one—a closure.
The substitution component is the traditional environment of environment machines, and the resulting machine is an environment machine. This machine coincides with the CEK machine with an abort operator [42].

In Section 3.7, we state the correctness of the resulting CEK machine with respect to evaluation in the $\lambda\hat{\rho}$-calculus with an abort operation.

### 3.1 A reduction semantics for applicative order and abort

A reduction semantics for applicative-order reduction in the $\lambda\hat{\rho}A$-calculus builds on the applicative-order strategy presented in Section 2.3. The grammar of terms and closures contains additional productions for the abort operation:

\[
\begin{align*}
\text{(Term)} & \quad t ::= \ldots | A t \\
\text{(Closure)} & \quad c ::= \ldots | A c
\end{align*}
\]

The reduction semantics is context-sensitive and its contraction rules contain the contractions of the $\lambda\hat{\rho}$-calculus (here stated in the context-sensitive form):

\[
\begin{align*}
\text{(Var)} & \quad \langle i[c_1 \cdots c_j], C \rangle \rightarrow A \langle c_i, C \rangle \quad \text{if } i \leq j \\
\text{(Beta)} & \quad \langle ((\lambda t)[s]) c, C \rangle \rightarrow A \langle t[c \cdot s], C \rangle \\
\text{(Prop)} & \quad \langle (t_0 \ t_1)[s], C \rangle \rightarrow A \langle (t_0[s]) (t_1[s]), C \rangle
\end{align*}
\]

as well as two new contractions for the abort operation:
(Prop) $\langle (A\ t)[s], \ C \rangle \to_A \langle A\ (t[s]), \ C \rangle$

(Abort) $\langle A\ v, \ C' \rangle \to_A \langle v, [\ ] \rangle$

Finally, the grammar of reduction contexts reads as follows:

(Context) $C ::= [\ ] | C[[\ ] c] | C[v [\ ]] | C[A [\ ]]

This reduction semantics satisfies the conditions stated in Section 2.1. We can therefore define the following three functions:

- decompose : Closure $\to$ Value + (Redex $\times$ Context)
- contract : Redex $\times$ Context $\to$ Closure $\times$ Context
- plug : Closure $\times$ Context $\to$ Closure

3.1.1 Decomposition

We define decompose as a transition function over closures and reduction contexts:

- decompose : Closure $\to$ Value + (Redex $\times$ Context)
  - decompose $c = \text{decompose}'(c, [\ ])$

- decompose' : Closure $\times$ Context $\to$ Value + (Redex $\times$ Context)
The main decomposition function, $\text{decompose}$, uses two auxiliary transition functions that work according to Property 1:

- $\text{decompose}$ 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 $\text{decompose}_{\text{aux}}$ 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.

- $\text{decompose}_{\text{aux}}$ is passed a reduction context and a value. It dispatches on the reduc-
- if the current context is empty, then the value is the result of the function;
- if the top constructor of the context is that of a function application, the actual parameter is decomposed in a new context; and
- otherwise, a redex has been found.

The decomposition function is total.

3.1.2 Context-sensitive contraction

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

\[
\text{contract} : \text{Redex} \times \text{Context} \rightarrow \text{Closure} \times \text{Context}
\]

\[
\begin{align*}
\text{contract} (i[v_1 \cdots v_m], C) &= (v_i, C) \\
\text{contract} ((t_0 t_1)[s], C) &= ((t_0[s])(t_1[s]), C) \\
\text{contract} ((\lambda t)[s] v, C) &= (t[v \cdot s], C) \\
\text{contract} ((A t)[s], C) &= (\lambda t[s], C) \\
\text{contract} (A v, C) &= (v, [])
\end{align*}
\]

In general, the contraction function is partial because of stuck terms.

3.1.3 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:

\[
\begin{align*}
\text{plug} : \text{Closure} \times \text{Context} &\rightarrow \text{Closure} \\
\text{plug} (c, [ ]) &= c \\
\text{plug} (c_0, C[[ ] c_1]) &= \text{plug} (c_0 c_1, C) \\
\text{plug} (c_1, C[c_0 [ ]]) &= \text{plug} (c_0 c_1, C) \\
\text{plug} (c, C[A [ ]]) &= \text{plug} (A c, C)
\end{align*}
\]

The plugging function is total.

### 3.1.4 One-step reduction

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

\[
\text{reduce} : \text{Closure} \rightarrow \text{Closure} \\
\text{reduce} c = \text{case} \ \text{decompose} \ c \\
of \ v \Rightarrow v \\
| (r, C) \Rightarrow \text{plug} (c', C') \quad \text{where} \ (c', C') = \text{contract} \ (r, C)
\]

In general, the one-step reduction function is partial because of the contraction function.

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

**Proposition 1.** For any non-value closure \(c\) and for any closure \(c'\), \(c \rightarrow_A c' \iff \text{reduce} \ c = c'\).
3.1.5 Reduction-based evaluation

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

\[
\text{iterate} : \text{Value} + (\text{Redex} \times \text{Context}) \rightarrow \text{Value} \\
\text{iterate} \ v = v \\
\text{iterate} \ (r, C) = \text{iterate} \ (\text{decompose} \ (\text{plug} \ (c', C'))) \\
\text{where} \ (c', C') = \text{contract} \ (r, C)
\]

\[
\text{evaluate} : \text{Term} \rightarrow \text{Value} \\
\text{evaluate} \ t = \text{iterate} \ (\text{decompose} \ (t[\bullet]))
\]

This evaluation function is partial because of the one-step reduction function and also because a reduction sequence might not terminate.

\textbf{Proposition 2.} \ For any closed term \( t \) and any value \( v \), \( t\[\bullet\] \rightarrow_{A^*} v \iff \text{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 [36], Danvy and Nielsen pointed out that the composition of plug and decompose (which are total) could be replaced by a more efficient function, refocus (which is also total), 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 3 (Danvy and Nielsen [26, 36]).** For any closure \( c \) and reduction context \( C \),

\[
\text{decompose} \left( \text{plug} \left( c, C \right) \right) = \text{decompose}' \left( c, C \right)
\]

In words: refocusing amounts to continuing the decomposition of the given contractum in the given context.

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

\[
\text{refocus} : \text{Closure} \times \text{Context} \rightarrow \text{Value} + (\text{Redex} \times \text{Context})
\]

\[
\text{refocus} (i[s], C) = (i[s], C)
\]

\[
\text{refocus} ((\lambda t)[s], C) = \text{refocus}_{\text{aux}} (C, (\lambda t)[s])
\]

\[
\text{refocus} ((t_0 t_1)[s], C) = ((t_0 t_1)[s], C)
\]

\[
\text{refocus} (c_0 c_1, C) = \text{refocus} (c_0, C[[] c_1])
\]

\[
\text{refocus} ((A t)[s], C) = ((A t)[s], C)
\]

\[
\text{refocus} (A c, C) = \text{refocus} (c, C[A []])
\]

\[
\text{refocus}_{\text{aux}} : \text{Context} \times \text{Value} \rightarrow \text{Value} + (\text{Redex} \times \text{Context})
\]

\[
\text{refocus}_{\text{aux}} ([], v) = v
\]

\[
\text{refocus}_{\text{aux}} (C[[] c], v) = \text{refocus} (c, C[v []])
\]

\[
\text{refocus}_{\text{aux}} (C[v’ []], v) = (v’ v, C)
\]

\[
\text{refocus}_{\text{aux}} (C[A []], v) = (A v, C)
\]

In this abstract machine, the configurations are pairs containing a closure and a context; the final transitions are specified by the first, third, and fifth clauses of \text{refocus} and by the first, third, and fourth clauses of \text{refocus}_{\text{aux}}; and the initial transition is specified by two clauses of the corresponding ‘refocused’ evaluation function, which reads as follows:

\[
\text{iterate} : \text{Value} + (\text{Redex} \times \text{Context}) \rightarrow \text{Value}
\]

\[
\text{iterate} v = v
\]

\[
\text{iterate} (r, C) = \text{iterate} (\text{refocus} (c’, C’))
\]

where \((c’, C’)) = \text{contract} (r, C)
evaluate : Term → Value
   evaluate \( t \) = iterate (refocus (\( t[\mathord{\cdot}] \), [ ]))

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

This evaluation function computes the transitive closure of refocus using the auxiliary function iterate as a trampoline [48]. 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 into an abstract machine, we distribute the calls to iterate from the definitions of evaluate and of iterate to the definitions of refocus and refocus\(_{\text{aux}}\):

evaluate : Term → Value
   evaluate \( t \) = refocus (\( t[\mathord{\cdot}] \), [ ])

iterate : Value + (Redex × Context) → Value
   iterate \( v \) = \( v \)
   iterate \( (r, C) \) = refocus \( (c', C') \)
      where \( (c', C') = \text{contract} (r, C) \)

refocus : Closure × Context → Value
refocus\ (i[s], \ C) = \ iterate\ (i[s], \ C)
refocus\ ((\lambda t)[s], \ C) = \ refocus_{aux}\ (C, \ (\lambda t)[s])
refocus\ ((t_0 \ t_1)[s], \ C) = \ iterate\ ((t_0 \ t_1)[s], \ C)
refocus\ (c_0 \ c_1, \ C) = \ refocus\ (c_0, \ C[[c_1]])
refocus\ ((A \ t)[s], \ C) = \ iterate\ ((A \ t)[s], \ C)
refocus\ (A \ c, \ C) = \ refocus\ (c, \ C[A \ ])]}

refocus_{aux} : Context \times \ Value \rightarrow \ Value
refocus_{aux}\ ([], \ v) = \ iterate\ v
refocus_{aux}\ (C[[c]], \ v) = \ refocus\ (c, \ C[[v]])
refocus_{aux}\ (C[v'[[]], \ v) = \ iterate\ (v'[v, \ C')
refocus_{aux}\ (C[\ A [], \ v) = \ iterate\ (A \ v, \ C')

The resulting definitions of evaluate, iterate, refocus, and refocus_{aux} are that of four mutually recursive transition functions that form an abstract machine. In this abstract machine, the configurations are pairs of a closure and a context, the initial transition is specified by evaluate, and the final transition in the first clause of iterate. The compatibility rules are implemented by refocus and refocus_{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:

iterate : Value + (Redex \times \ Context) \rightarrow \ Value
iterate\ v = v
iterate\ (i[v_1 \cdots v_m], \ C') = \ refocus\ (v_i, \ C')
iterate\ ((t_0 \ t_1)[s], \ C') = \ refocus\ ((t_0[s]) \ (t_1[s]), \ C')
iterate (((λt)[s]) v, C') = refocus (t[v · s], C')
iterate ((A t)[s], C') = refocus (A (t[s]), C')
iterate (A v, C') = refocus (v, [])

By construction, the machine therefore separately implements the reduction rules (with \text{iterate}) and the compatibility rules (with \text{refocus} and \text{refocus}_{\text{aux}}); for this reason, we refer to it as a 'staged abstract machine' [51].

3.4 An eval/apply abstract machine

As already observed by Danvy and Nielsen in their work on refocusing, inlining \text{iterate} yields an eval/apply abstract machine [66]. Inlining the calls to \text{iterate} in the staged abstract machine yields the following eval/apply machine, where \text{refocus} (the 'eval' transition function) dispatches on closures and \text{refocus}_{\text{aux}} (the 'apply' function) dispatches on contexts:

\text{evaluate} : \text{Term} \to \text{Value}
\text{evaluate } t = \text{refocus} (t[\bullet], [])

\text{refocus} : \text{Closure} \times \text{Context} \to \text{Value}
\text{refocus} (i[v_1 \ldots v_m], C') = \text{refocus} (v_i, C')
\text{refocus} (((λt)[s], C') = \text{refocus}_{\text{aux}} (C, (λt)[s])
\text{refocus} ((t_0 t_1)[s], C') = \text{refocus} ((t_0[s])(t_1[s]), C')
\text{refocus} (c_0 c_1, C') = \text{refocus} (c_0, C[[]c_1])
refocus \(( (\mathcal{A} t)[s], C) \) = refocus \((\mathcal{A} (t[s]), C)\)

refocus \((\mathcal{A} c, C) \) = refocus \((c, C[A[ ]])\)

\[
\text{refocus}_{\text{aux}} : \text{Context} \times \text{Value} \rightarrow \text{Value}
\]

\[
\text{refocus}_{\text{aux}} ([], v) = v
\]

\[
\text{refocus}_{\text{aux}} (C[[ ] c], v) = \text{refocus} (c, C[v [ ]])
\]

\[
\text{refocus}_{\text{aux}} (C[((\lambda t)[s]) [ ]], v) = \text{refocus} (t[v \cdot s], C)
\]

\[
\text{refocus}_{\text{aux}} (C[\mathcal{A} [ ]], v) = \text{refocus} (v, [ ])
\]

### 3.5 Transition compression

In three cases, the eval/apply machine of Section 3.4 yields a configuration that uniquely determines the next one. Let us shortcut these “corridor transitions:”

- The machine only produces an application of closures through an \((\text{App})\) reduction step (third clause of \text{refocus}). We observe that in a reduction sequence, an \((\text{App})\) reduction step is always followed by a decomposition step (fourth clause of \text{refocus}). As a shortcut, we coalesce the two consecutive transitions into one, replacing the third and fourth clauses of \text{refocus} with the following one:

\[
\text{refocus} \(( (t_0 t_1)[s], C) \) = \text{refocus} \((t_0[s], C[[ ] (t_1[s]))\))
\]

- The machine only produces an application of \(\mathcal{A}\) through an \((\text{Abort})\) reduction step (second-to-last clause of \text{refocus}). We observe that in a reduction sequence, an \((\text{Abort})\) 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:

$$\text{refocus}(A(t[s]), C) = \text{refocus}(t[s], C[A[]])$$

- The configuration refocus($v, []$) always yields the configuration refocus$_{aux}([], v)$, which triggers a final transition. As a shortcut, we coalesce these three consecutive transitions into one final transition, replacing the last clause of refocus$_{aux}$ with the following one:

$$\text{refocus}_{aux}(C[A[]], v) = v$$

### 3.6 An environment machine

Because of the two first compressions of Section 3.5, refocus is now defined by structural induction over terms instead of by structural induction over closures. Its type can therefore be refined to $(\text{Term} \times \text{Substitution}) \times \text{Context} \rightarrow \text{Value} + (\text{Redex} \times \text{Context})$, in effect unfolding closures into a term and a substitution. We thus replace each closure in the compressed definition of evaluate and refocus by its unfolding. Flattening $(\text{Term} \times \text{Substitution}) \times \text{Context}$ into $\text{Term} \times \text{Substitution} \times \text{Context}$ yields the following abstract machine:

$$\begin{align*}
\text{(Value)} & \quad v ::= (\lambda t, s) \\
\text{(Context)} & \quad C ::= [] | C[#](t, s) | C[v[]] | C[A[]]
\end{align*}$$
refocus : Term × Substitution × Context

refocus (i, v_1 \ldots v_m, C)
refocus (\lambda t, s, C)
refocus (t_0 t_1, s, C)
refocus (A t, s, C)

refocus_{aux} : Context × Value
\[
\begin{align*}
&\text{refocus}_{\text{aux}} (\text{[]}, v) \\
&\text{refocus}_{\text{aux}} (C[[]] (t, s), v) \\
&\text{refocus}_{\text{aux}} (C[(\lambda t, s) []], v) \\
&\text{refocus}_{\text{aux}} (C[A []], v)
\end{align*}
\]

: \quad \text{Term} \rightarrow \text{Value}

= \quad \text{refocus} (t, \bullet, [ ])

\rightarrow \quad \text{Value}
\[
\begin{align*}
    &= \text{refocus} (v_i, C) \\
    &= \text{refocus}_{\text{aux}} (C, (\lambda t, s)) \\
    &= \text{refocus} (t_0, s, C[[t_1, s]]) \\
    &= \text{refocus} (t, s, C[A[]])
\end{align*}
\]
We observe that this machine coincides with the CEK machine extended with an abort operator—an extension that was designed as such [42]. In particular, the substitution component assumes the role of the environment.

3.7 Correctness

We state the correctness of the final result—the CEK machine—with respect to evaluation in the $\lambda \rho A$-calculus.

**Theorem 1.** For any closed term $t$ in $\lambda \rho A$,

$$t[\bullet] \to^*_{\lambda A} (\lambda t')[s'] \text{ if and only if evaluate } t = (\lambda t', s').$$

**Proof.** The proof relies on the correctness of refocusing [36], and the (trivial) meaning preservation of each of the subsequent transformations. \qed

The theorem states that the CEK machine is correct in the sense that it computes closed weak head normal forms, and that it realizes the applicative-order strategy in the $\lambda \rho A$-calculus, which makes it a call-by-value machine [70]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-value evaluation in the $\lambda \rho A$-calculus.
3.8 Conclusion

We have presented a detailed and systematic derivation of an abstract machine for call-by-value evaluation in the $\lambda$-calculus with an abort operation. We started from the specification of the applicative-order reduction strategy in the $\lambda\hat{\rho}$-calculus with an abort operation and we finished with a formal correspondence between the calculus and the abstract machine.

In the next six sections, we apply the same method in a variety of computational situations, each of which has been separately studied and reported in the literature. Our presentation is structured around four elements: language, notion of context-sensitive reduction, abstract machine, and formal correspondence. The rest is mechanical and therefore omitted.

4 The $\lambda\hat{\rho}K$-calculus

The Krivine machine is probably the best-known abstract machine implementing the normal-order reduction strategy in the $\lambda$-calculus [30]. In our previous work [8], we have pointed out that Krivine's original machine [59] does not coincide with the Krivine Machine As We Know It [21, 23] in that it implements generalized instead of ordinary $\beta$-reduction: indeed Krivine's machine reduces the term $(\lambda\lambda t) t_1 t_2$ in one step whereas the Krivine ma-
machine reduces it in two steps. Furthermore, an extension of the archival version of Krivine’s machine [60, Section 3] also caters for call/cc (noted $\mathcal{K}$ below).

In our previous work [8], we have presented the calculus corresponding to the original version of Krivine’s machine. This machine uses closures and an environment and correspondingly, the calculus is one of explicit substitutions, $\lambda\rho$.

Here, we present the calculus corresponding to the archival version of Krivine’s machine with $\mathcal{K}$. This machine also uses closures and an environment. Correspondingly, the calculus is one of explicit substitutions, $\lambda\rho\mathcal{K}$. We build on top of Krivine’s language of terms by specifying syntactic categories of closures and substitutions as shown below. The calculus is tied to a particular reduction strategy. Here, like Krivine, we consider the normal-order reduction strategy and therefore call by name [70].

### 4.1 The language of $\lambda\rho\mathcal{K}$

The abstract syntax of the language is as follows:

- **(terms)**
  \[ t ::= i \mid \lambda^n t \mid t t \mid \mathcal{K} t \]

- **(closures)**
  \[ c ::= t[s] \mid c c \mid \mathcal{K} c \mid \llbracket C \rrbracket \]

- **(values)**
  \[ v ::= (\lambda^n t)[s] \mid \llbracket C \rrbracket \]

- **(substitutions)**
  \[ s ::= \cdot \mid c \cdot s \]

- **(reduction contexts)**
  \[ C ::= [] \mid C[[ ] c] \mid C[\mathcal{K}[ ] ] \]

A nested $\lambda$-abstraction of the form $\lambda^n t$ is to be understood as a syntactic abbreviation for $\underbrace{\lambda \ldots \lambda}_{n} t$, where $t$ is not a $\lambda$-abstraction.
In $\lambda\rho\mathcal{K}$, a value is either a closure with a $\lambda$-abstraction in the term part, or the representation of a reduction context captured by $\mathcal{K}$.

### 4.2 Notion of context-sensitive reduction

The notion of reduction is specified by the set of rules shown below. The rules (Var) and (Prop) are as in the $\lambda\rho$-calculus, and (Beta$^+$) supersedes the (Beta) rule by performing a generalized $\beta$-reduction in one step:

$$(\text{Var}) \quad \langle i[c_1 \cdots c_j], C \rangle \rightarrow_\mathcal{K} \langle c_i, C \rangle \text{ if } i \leq j$$

$$(\text{Beta}^+) \quad \langle (\lambda^n t)[s], C[[[ \cdots [ ] c_n ] \cdots ] c_1] \rangle \rightarrow_\mathcal{K} \langle t[c_1 \cdots c_n \cdot s], C \rangle$$

$$(\text{Beta}_C) \quad \langle \Gamma C^n, C[[ ] c] \rangle \rightarrow_\mathcal{K} \langle c, C' \rangle$$

The three last rules account for call/cc: the first is an ordinary propagation rule, and the two others describe a continuation capture. In the first case, the current continuation is captured and passed to a function, and in the second, it is captured and passed to an already captured continuation.
4.3 Krivine’s machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

\[
\begin{align*}
\langle i, s, C \rangle & \Rightarrow_{\kappa} \\
\langle \lambda^n t, s, C \rangle & \Rightarrow_{\kappa} \\
\langle t_0 t_1, s, C \rangle & \Rightarrow_{\kappa} \\
\langle \mathcal{K} t, s, C \rangle & \Rightarrow_{\kappa}
\end{align*}
\]
\langle C, \neg C \wedge \rangle \text{ if } s(i) = \neg C
\langle C, (\lambda^n t, s) \rangle
\langle t_0, s, C[[ ]](t_1, s) \rangle
\langle t, s, C[K[ ]\rangle
\nu
\[ \langle t, c_1 \cdots c_n \cdot s, C \rangle \]
\[ \langle t, s, C' \rangle \]
\[ \langle C', \lnot C' \rangle \]
\[ \langle C'[\lnot\lnot C'], v \rangle \]

This machine coincides with the extension of Krivine's machine with \( \mathcal{K} \)—an extension which was designed as such and not connected to any calculus [60, Section 3].

4.4 Formal correspondence

Proposition 4. For any term \( t \) in the \( \lambda \rho \mathcal{K} \)-calculus,

\[ t[\bullet] \to^*_\mathcal{K} v \quad \text{if and only if} \quad \langle t, \bullet, [] \rangle \Rightarrow^*_\mathcal{K} v. \]
The $\lambda\rho K$-calculus therefore directly corresponds to the archival version of Krivine's machine with call/cc.

5 The $\lambda\rho\mu$-calculus

In this section we present a calculus of closures that extends Parigot's $\lambda\mu$-calculus [69] and the corresponding call-by-name abstract machine obtained by refocusing.

We want to compare our derived abstract machine with an existing one designed by de Groote [38] and therefore we adapt his syntax, which differs from Parigot's in that arbitrary terms can be abstracted by $\mu$ (not only named ones). In addition, de Groote presents a calculus of explicit substitutions built on top of the $\lambda\mu$-calculus, and uses it to prove the correctness of his machine. We show that a $\lambda\rho$-like calculus of closures is enough to model evaluation in the $\lambda\mu$-calculus and to derive the same abstract machine as de Groote.

The $\lambda\mu$-calculus is typed, and suitable typing rules can be given to the calculus of closures we present below. The reduction rules we show satisfy the subject reduction property, and in consequence, the machine we derive operates on typed terms. To remain concise, we omit all the typing considerations and concentrate on the syntactic correspondence between the calculus and the machine.
5.1 The language of $\lambda\rho\mu$

We use de Bruijn indices for both the $\lambda$-bound variables and the $\mu$-bound variables. The two kinds of variables are represented using the same set of indices, which leads one to an abstract machine with one environment [38]. Alternatively, we could use two separate sets of indices, which would then yield two environments in the resulting machine (one for each kind of variable).

The abstract syntax of the language is specified as follows:

\[
\begin{align*}
\text{(terms)} & : = i \\
\lambda t & \\
t t & \\
\mu t & \\
\text{(closures)} & : = t[s] \\
& \quad | \ c c \\
\text{(values)} & : = (\lambda t)[s] \\
\text{(substitutions)} & : = \bullet \\
& \quad | \ C \cdot s \\
\text{(reduction contexts)} & : = [ ] \\
& \quad | \ C[[ ] c]
\end{align*}
\]

We consider only closed $\lambda$-terms, and $i \geq 0$. Bound variables are indexed starting with 1, and a (free) occurrence of a variable 0 indicates a distinguished toplevel continuation (similar to $\tau_p$ in Ariola et al.'s setting [6]). A substitution is a non-empty sequence of closures—to be substituted for $\lambda$-bound variables, or captured reduction contexts—to be used when accessing $\mu$-bound variables.

Programs are closures of the form $t[[ ]] \cdot \bullet$, where the empty context is to be substituted for the toplevel continuation variable 0.

5.2 Notion of context-sensitive reduction
The notion of reduction extends that of the $\lambda \rho$ with two rules: (Mu), which captures the entire reduction context and stores it in the substitution, and (Rho), which reinstates a captured context when a continuation variable is applied:

(Beta) \[ \langle (\lambda t)[s], C[[c]] \rangle \to_\mu \langle t[c \cdot s], C \rangle \]
Var \[ \langle i[s], C \rangle \to_\mu \langle c, C \rangle \text{ if } s(i) = c \]

(Prop) \[ \langle (t_0 \ t_1)[s], C \rangle \to_\mu \langle (t_0[s]) (t_1[s]), C \rangle \]
Mu \[ \langle (\mu t)[s], C \rangle \to_\mu \langle t[C \cdot s], [] \rangle \]
Rho \[ \langle ([i]t)[s], [] \rangle \to_\mu \langle t[s], C \rangle \text{ if } s(i) = C \]

5.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

\[ \langle \lambda t, s, C \rangle \Rightarrow_\mu \langle C, (\lambda t, s) \rangle \]
\[ \langle i, s, C \rangle \Rightarrow_\mu \langle t', s', C \rangle \text{ if } s(i) = (t', s') \]
\[ \langle t_0 \ t_1, s, C \rangle \Rightarrow_\mu \langle t_0, s, C[[s]](t_1, s) \rangle \]
\[ \langle \mu t, s, C \rangle \Rightarrow_\mu \langle t, C \cdot s, [] \rangle \]
\[ \langle [i]t, s, [] \rangle \Rightarrow_\mu \langle t, s, C \rangle \text{ if } s(i) = C \]

\[ \langle [], v \rangle \Rightarrow_\mu v \]
\[ \langle C[[c]], (\lambda t, s) \rangle \Rightarrow_\mu \langle t, c \cdot s, C \rangle \]

This machine coincides with de Groote's final abstract machine [38, p. 24], except that instead of traversing the environment as a list, it directly fetches the right substitutee for a given index \( i \).

5.4 Formal correspondence

Proposition 5. For any term \( t \) in the \( \lambda \rho \mu \)-calculus,

\[ t[[\cdot ]] \rightarrow_\mu^* v \text{ if and only if } \langle t, [[\cdot ]], [] \rangle \Rightarrow_\mu^* v. \]

The \( \lambda \rho \mu \)-calculus therefore directly corresponds to de Groote's abstract machine for the \( \lambda \mu \)-calculus.

6 Delimited continuations

Continuations have been discovered multiple times [73], but they acquired their name for describing jumps [82], using what is now known as continuation-passing style (CPS) [80]. A full-fledged control operator, \( J \) [62,84], however, existed before CPS, providing first-class continuations in direct style. Continuations therefore existed before CPS, and so one could say that it was really CPS that was discovered multiple times.

Conversely, delimited continuations, in the form of the traditional success and failure continuations [76], have been regularly used in artificial-intelligence programming [14,54,83] for generators and backtracking. They also occur in the study of reflective towers [79],
where the notions of meta-continuation [89] and of “jumpy” vs. “pushy” continuations [34] arose. A full-fledged delimited control operator, \# (pronounced “prompt”), however, was introduced independently of CPS and of reflective towers, to support operational equivalence in \(\lambda\)-calculi with first-class control [41, 44]. Only subsequently were control delimiters connected to success and failure continuations [32].

The goal of this section is to provide a uniform account of delimited continuations. Three data points are in presence—a calculus and an abstract machine, both invented by Felleisen [41], and an extension of CPS, as discovered by Danvy and Filinski [32]:

**Calculus:** As we show below, an explicit-substitutions version of Felleisen’s calculus of dynamic delimited continuations can be refocused into his extension of the CEK machine, which uses closures and an environment.

**Abstract machine:** As we have shown elsewhere [11], Felleisen’s extension of the CEK machine is not in defunctionalized form (at least for the usual notion of defunctionalization [35, 72]); it needs some adjustment to be so, which leads one to a dynamic form of CPS that threads a state-like trail of delimited contexts.

**CPS:** Defunctionalizing Danvy and Filinski’s continuation-based evaluator yields an environment-based machine [7], and we present below the corresponding calculus of static delimited continuations.

The syntactic correspondence makes it possible to directly compare (1) the calculi of dynamic and of static delimited continuations, (2) the extended CEK machine and the machine corresponding to the calculus of static delimited continuations and to the
continuation-based evaluator, and (3) the evaluator corresponding to the extended CEK machine and the continuation-based evaluator. In other words, rather than having to relate heterogeneous semantic artifacts such as a calculus with actual substitutions, an environment-based machine, and a continuation-based evaluator, we are now in position to directly compare two calculi, two abstract machines, and two continuation-based evaluators.

We address static delimited continuations in Section 6.1 and dynamic delimited continuations in Section 6.2. In both cases, we consider the left-to-right applicative-order reduction strategy and therefore left-to-right call by value.

6.1 The $\lambda\rho S$-calculus

The standard $\lambda$-calculus is extended with the control operator shift (written $S$) that captures the current delimited continuation and with the control delimiter reset (written $\langle \cdot \rangle$) that initializes the current delimited continuation.

6.1.1 The language of $\lambda\rho S$

The abstract syntax of the language is as follows:

$$t ::= i \mid \lambda t \mid t t \mid St \mid \langle t \rangle$$

$$c ::= t[s] \mid cc \mid Sc \mid \langle c \rangle \mid \Gamma C$$
\( v ::= (\lambda t)[s] \mid [C] \)

(substitutions)  
\( s ::= \bullet \mid c \cdot s \)

(contexts)  
\( C_1 ::= [ ] \mid C_1[v][] \mid C_1[[c] \mid C_1[S][]] \)

(meta-contexts)  
\( C_2 ::= \bullet \mid C_1 \cdot C_2 \)

For readability, we write \( C_1 \cdot C_2 \) rather than \( C_2[\langle C_1[]\rangle] \).

The control operator \( S \) captures the current delimited context and replaces it with the empty context. The control delimiter \( \langle \cdot \rangle \) initializes the current delimited context, saving the then-current one onto the meta-context. When a captured delimited context is resumed, the current delimited context is saved onto the meta-context. When the current delimited context completes, the previously saved one, if there is any, is resumed; otherwise, the computation terminates. This informal description paraphrases the definitional interpreter for shift and reset, which has two layers of control—a current delimited continuation (akin to a success continuation) and a meta-continuation (akin to a failure continuation), as arises naturally when one CPS-transforms a direct-style evaluator twice \([32]\). Elsewhere \([7]\), we have defunctionalized this interpreter into an environment-based machine, which we present next.

6.1.2 The eval/apply/meta-apply abstract machine

The environment-based machine is in “eval/apply/meta-apply” form (to build on Peyton Jones’s terminology \([66]\)) because the continuation is defunctionalized into a context and the corresponding apply transition function, and the meta-continuation is defunctionalized into a meta-context (here a list of contexts) and the corresponding meta-apply transition function:
\[ \langle i, s, C_1, C_2 \rangle \Rightarrow_s \langle t', s', C_1, C_2 \rangle \text{ if } s(i) = (t', s') \]
\[ \langle \lambda t, s, C_1, C_2 \rangle \Rightarrow_s \langle C_1, (\lambda t, s), C_2 \rangle \]
\[ \langle t_0 t_1, s, C_1, C_2 \rangle \Rightarrow_s \langle t_0, s, C_1[](t_1, s), C_2 \rangle \]
\[ \langle S t, s, C_1, C_2 \rangle \Rightarrow_s \langle t, s, C_1[S[]], C_2 \rangle \]
\[ \langle \langle t \rangle, s, C_1, C_2 \rangle \Rightarrow_s \langle t, s, [], C_1 \cdot C_2 \rangle \]

17

\[ \langle [[], v, C_2 \rangle \Rightarrow_s \langle C_1[[](t, s)], v, C_2 \rangle \Rightarrow_s \]
\[ \langle C_1[(\lambda t, s)[]], v, C_2 \rangle \Rightarrow s \]
\[ \langle C_1[\neg C_1[]], v, C_2 \rangle \Rightarrow s \]
\[ \langle C_1[S[]], (\lambda t, s), C_2 \rangle \Rightarrow s \]
\[ \langle C_1[S[]], \neg C_1[], C_2 \rangle \Rightarrow s \]
\[ \langle \bullet, v \rangle \Rightarrow s \]
\[ \langle C_1 \cdot C_2, v \rangle \Rightarrow s \]
\[ \langle C_2, v \rangle \]
\[ \langle t, s, C_1[v[[]]], C_2 \rangle \]
\[ \langle t, v \cdot s, C_1, C_2 \rangle \]
\[ \langle C'_1, v, C_1 \cdot C_2 \rangle \]
\[ \langle t, \lceil C_1 \rceil \cdot s, [\text{ }], C_2 \rangle \]
\[ \langle C'_1, \lceil C_1 \rceil, [\text{ }], \cdot C_2 \rangle \]
We have observed that this machine is in the range of refocusing, transition compression, and closure unfolding for the following calculus $\lambda \rho S$.

6.1.3 Notion of context-sensitive reduction

The $\lambda \rho S$-calculus uses two layers of contexts: $C_1$ and $C_2$. A non-value closure is decomposed into a redex, a context $C_1$, and a meta-context $C_2$, and the notion of reduction is specified by the following rules:
Since none of the contractions depends on the meta-context, it is evident that the notion of reduction $\rightarrow_S$ is compatible with meta-contexts, but it is not compatible with contexts, due to $S_\lambda$ and $S_C$. The $\langle \cdot \rangle$ construct therefore delimits the parts of non-value closures in which context-sensitive reductions may occur, and partially restores the compatibility of reductions. In particular, $\langle \langle t[s] \rangle, C_1, C_2 \rangle$ is decomposed into $\langle t[s], [], C_1 \cdot C_2 \rangle$ in the course of decomposition towards a context-sensitive redex.

6.1.4 Formal correspondence

Proposition 6. For any term $t$ in the $\lambda\rho S$-calculus,
The $\lambda\rho S$-calculus therefore directly corresponds to the abstract machine for shift and reset.

6.1.5 The CPS hierarchy

Iterating the CPS transformation on a direct-style evaluator for the $\lambda$-calculus gives rise to a family of CPS evaluators. At each iteration, one can add shift and reset to the new inner layer. The result forms a CPS hierarchy of static delimited continuations [32, 37] which Filinski has shown to be able to represent layered monads [46]. Recently, Kameyama has proposed an axiomatization of the CPS hierarchy [56]. Elsewhere [7], we have studied its defunctionalized counterpart and the corresponding hierarchy of calculi.

6.2 The $\lambda\rho F$-calculus

The standard $\lambda$-calculus is extended with the control operator $F$ that captures a segment of the current context and with the control delimiter prompt (noted #) that initializes a new segment in the current context.

6.2.1 The language of $\lambda\rho F$

The abstract syntax of the language is as follows:

\[ t[\bullet] \rightarrow^*_S v \quad \text{if and only if} \quad \langle t, \bullet, [], \bullet \rangle \Rightarrow^*_S v. \]
(terms) \[ t ::= i | \lambda t | t t | \mathcal{F} t | \# t \]
(closures) \[ c ::= t[s] | c c | \mathcal{F} c | \# c | \neg C \]
(values) \[ v ::= (\lambda t)[s] | \neg C \]
(substitutions) \[ s ::= \bullet | c \cdot s \]
(reduction contexts) \[ C ::= [ ] | C[[ ] c] | C[v []] | C[\mathcal{F} []] | C[# []] \]

6.2.2 Notion of context-sensitive reduction

The control operator \( \mathcal{F} \) captures a segment of the current context up to a mark. The control delimiter \( \# \) sets a mark on the current context. When a captured segment is resumed, it is composed with the current context. For the rest, the notion of reduction is as usual:\(^2\)

(Var) \[ \langle i[c_1 \cdots c_j], C \rangle \rightarrow_{\mathcal{F}} \langle c_i, C \rangle \text{ if } i \leq j \]
(Beta) \[ \langle ((\lambda t)[s]) v, C \rangle \rightarrow_{\mathcal{F}} \langle t[v \cdot s], C \rangle \]
(Beta_C) \[ \langle \neg C' \neg v, C \rangle \rightarrow_{\mathcal{F}} \langle C'[v], C \rangle \]
(Prop) \[ \langle (t_0 t_1)[s], C \rangle \rightarrow_{\mathcal{F}} \langle (t_0[s]) (t_1[s]), C \rangle \]
(Prop_F) \[ \langle (\mathcal{F} t)[s], C \rangle \rightarrow_{\mathcal{F}} \langle \mathcal{F} (t[s]), C \rangle \]
(Prop_#) \[ \langle (#t)[s], C \rangle \rightarrow_{\mathcal{F}} \langle # (t[s]), C \rangle \]
(\( \mathcal{F}_\lambda \)) \[ \langle \mathcal{F}((\lambda t)[s]), C[# C'] \rangle \rightarrow_{\mathcal{F}} \langle t[\neg C' \neg \cdot s], C \rangle \]
\quad \text{if } C' \text{ contains no mark}
(\( \mathcal{F}_C \)) \[ \langle \neg C' \neg \neg, C[# C'] \rangle \rightarrow_{\mathcal{F}} \langle \neg C' \neg C' \neg, C \rangle \]
\quad \text{if } C' \text{ contains no mark}
Alternatively, we could specify the reduction rules using two layers of contexts, similarly to the $\lambda\rho S$-calculus [7, 10, 11]. The difference between the two calculi would then be only in the rule (Beta$_C$):

$$(\text{Beta}_C) \quad \langle \neg C'_1 \neg v, C_1, C_2 \rangle \rightarrow_{\mathcal{F}} \langle C'_1[v], C_1, C_2 \rangle$$

where there is no delimiter around $C'[v]$. As in the previous case of the $\lambda\rho S$-calculus, such two-layered decomposition makes it evident that the contraction rules are compatible with the meta-context, since it is isolated by the use of a control delimiter.

2The original version of $\mathcal{F}$ does not reduce its argument first, but its followers do. The present version of $\mathcal{F}$ does likewise here, for a more direct comparison with $S$.

### 6.2.3 The eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

$$\langle i, s, C \rangle \Rightarrow_{\mathcal{F}} \langle t', s', C \rangle \text{ if } s(i) = (t', s')$$

$$\langle \lambda t, s, C \rangle \Rightarrow_{\mathcal{F}} \langle C, (\lambda t, s) \rangle$$

$$\langle t_0 \ t_1, s, C \rangle \Rightarrow_{\mathcal{F}} \langle t_0, s, C[[ ]] (t_1, s) \rangle$$

$$\langle \mathcal{F} \ t, s, C \rangle \Rightarrow_{\mathcal{F}} \langle t, s, C[\mathcal{F}[ ]] \rangle$$
\[
\langle \#t, s, C \rangle \Rightarrow_f \langle t, s, C[\#[]] \rangle
\]

\[
\langle C[[ \, (t, s)], v \rangle \Rightarrow_f \langle t, s, C[v [ ]] \rangle
\]

\[
\langle C[[\lambda t, s \, ]], v \rangle \Rightarrow_f \langle t, v \cdot s, C \rangle
\]

\[
\langle C[\Box C'' \, [ ]], v \rangle \Rightarrow_f \langle C' \circ C, v \rangle
\]

\[
\langle C[[\# C'[F[ ]]], \lambda t, s \rangle \Rightarrow_f \langle t, \Box C'' \cdot s, C \rangle
\]

where \( C'' \) contains no mark

\[
\langle C[[\# C'[F[ ]]], \Box C'' \rangle \Rightarrow_f \langle C'' \circ C, \Box C'' \rangle
\]

where \( C' \) contains no mark

\[
\langle C[[\# [ ]], v \rangle \Rightarrow_f \langle C, v \rangle
\]

This machine coincides with Felleisen's extension of the CEK machine—an extension which was designed as such [41, Section 3].

### 6.2.4 Formal correspondence

**Proposition 7.** For any term \( t \) in the \( \lambda \rho F \)-calculus,

\[
t[\bullet] \rightarrow^*_f v \quad \text{if and only if} \quad \langle t, \bullet, [ ] \rangle \Rightarrow^*_f v.
\]

This proposition parallels Felleisen's second correspondence theorem [41, p. 186]. The \( \lambda \rho F \)-calculus therefore directly corresponds to Felleisen's extension of the CEK machine.

### 6.2.5 A hierarchy of control delimiters
As described by Sitaram and Felleisen [78], one could have not one but several marks in the context and have control operators capture segments of the current context up to a particular mark. For these marks not to interfere in programming practice, they need to be organized hierarchically, forming a hierarchy of control delimiters [78, Section 5]. Alternatively, one could iterate Biernacki et al.’s dynamic CPS transformation [11] to give rise to a hierarchy of dynamic delimited continuations with a functional (CPS) counterpart. Except for the work of Gunter et al. [50] and more recently of Dybvig et al. [39], this area is little explored.

6.3 Conclusion

The syntactic correspondence has made it possible to exhibit the calculus corresponding to static delimited continuations as embodied in the functional idiom of success and failure continuations and more generally in the CPS hierarchy, and to show that (the explicit-substitutions version of) Felleisen’s calculus of dynamic delimited continuations corresponds to his extension of the CEK machine [41]. Elsewhere, we present the abstract machine [7] and the evaluator [32] corresponding to static delimited continuations and an evaluator [11] corresponding to dynamic delimited continuations. We are now in position to compare them pointwise:

- From a calculus point of view, it seems to us that one is better off with layered contexts because it is immediately obvious whether a notion of reduction is compat-

20
ible with them (see Section 6.1.3); a context containing marks is less easy to treat.

Otherwise, the difference between static and dynamic delimited continuations is tiny (see Section 6.2.2), and located in the rule (BetaC).

- From a machine point of view, separating between the current delimited context and the other ones is also simpler, as it avoids linear searches, copies, and concatenations (in this respect, efficient implementations, e.g., with an Algol-style display, in effect separate between the current delimited context and the other ones).

- From the point of view of CPS, the abstract machine for dynamic delimited continuations is not in defunctionalized form whereas the abstract machine for static delimited continuations is (and corresponds to an evaluator in CPS). Conversely, defunctionalizing a CPS evaluator provides design guidelines, whereas without CPS, one is on one's own, and locally plausible choices may have unforeseen global consequences which are then taken as the norm. Two cases in point:

1. in Lisp, it was locally plausible to push both formal and actual parameters at function-call time, and to pop them at return time, but this led to dynamic scope since variable lookup then follows the dynamic link; and

2. here, it was locally plausible to concatenate a control-stack segment to the current control stack ("From this, we learn that an empty context adds no information." [45, p. 58]), but this led to dynamic delimited continuations since capturing a segment of a concatenated context then gives access to beyond the concatenation point.
Granted, a degree of dynamism makes it possible to write compact programs (e.g., a compositional breadth-first traversal without a data-queue accumulator and in direct style [12]), but it is very difficult to reason about them and they are not necessarily more efficient.

- From the point of view of expressiveness, for example, in Lisp, one can simulate the static scope of Scheme by making each lambda-abstraction a “funarg” and in Scheme, one can simulate the dynamic scope of Lisp by threading an environment of fluid variables in a state-monad fashion. Similarly, static delimited continuations can be simulated using dynamic ones by delimiting the extent of each captured continuation [10], and dynamic delimited continuations can be simulated using static ones by threading a trail of contexts in a state-monad fashion [11, 39, 58, 77]. As to which should be used by default, the question then reduces to which behavior is the norm and which should be simulated if it is needed.

In summary, the calculi, the abstract machines, and the evaluators all differ. In one approach, continuations are dynamically composed by concatenating their representations [45] and in the other, continuations are statically composed through a meta-continuation. These differences result from distinct designs: Felleisen and his colleagues started from a calculus and wanted to go “beyond continuations” [44], and therefore beyond CPS, whereas Danvy and Filinski were aiming at a CPS account of delimited control, one that has turned out to be not without practical, semantical, and logical content.
7 Stack inspection

This section addresses Fournet and Gordon's $\lambda_{sec}$-calculus, which formalizes security enforcement by stack inspection [47]. We first present a calculus of closures built on top of the $\lambda_{sec}$-calculus, and we construct the corresponding environment-based machine. This machine is a storeless version of the fg machine presented by Clements and Felleisen [16, Figure 1]. (We consider the issue of store-based machines in Section 8.) This machine is not properly tail-recursive, and so Clements and Felleisen presented another machine—the cm machine—which does implement stack inspection in a properly tail-recursive manner [16, Figure 2]. The cm machine builds on Clinger's formalization of proper tail-recursion (see Section 8) and it is therefore store-based; we considered its storeless version here, and we present the corresponding calculus of closures. We show how the tail-optimization of the cm machine is reflected in the calculus. Finally, we turn to the unzipped version of the cm machine [4] and we present the corresponding state-based calculus of closures.

7.1 The $\lambda\rho_{sec}$-calculus

7.1.1 The language of $\lambda\rho_{sec}$

(terms) $t ::= i \mid \lambda t \mid tt \mid \text{grant } R \text{ in } t \mid \text{test } R \text{ then } t \text{ else } t \mid R[t] \mid \text{fail}$

(closures) $c ::= t[s] \mid cc \mid \text{grant } R \text{ in } c \mid \text{test } R \text{ then } c \text{ else } c \mid R[c]$  

(values) $v ::= (\lambda t)[s] \mid \text{fail}$

(substitutions) $s ::= \bullet \mid c \cdot s$
The set of terms consists of $\lambda$-terms and four constructs for handling different levels of security specified in a set $\mathcal{P}$: $\text{grant } R \text{ in } t$ grants the permissions $R$ to $t$; $\text{test } R \text{ then } t_0 \text{ else } t_1$ proceeds to evaluate $t_0$ if permissions $R$ are available, and otherwise $t_1$; a frame $R[t]$ restricts the permissions of $t$ to $R$; and finally, fail aborts the computation.

### 7.1.2 Notion of context-sensitive reduction

Given the predicate $\mathcal{OK}_\text{sec}(R, C)$ checking whether the permissions $R$ are available within the context $C$,

\[
\begin{align*}
\mathcal{OK}_\text{sec}(\emptyset, C) & \quad \mathcal{OK}_\text{sec}(R, \emptyset) \\
\mathcal{OK}_\text{sec}(R, C) & \quad \mathcal{OK}_\text{sec}(R, C[[]\ c]) \\
\mathcal{OK}_\text{sec}(R, C[[\ ]\ v]) & \quad \mathcal{OK}_\text{sec}(R, C[\text{grant } R \text{ in } \emptyset]) \\
R \subseteq R' & \quad \mathcal{OK}_\text{sec}(R, C) \quad \mathcal{OK}_\text{sec}(R \setminus R', C) \\
\end{align*}
\]

the notion of reduction is given by the following set of rules:

- **(Var)** \( \langle i[v_1 \cdots v_j], C \rangle \rightarrow_{\text{sec}} \langle v_i, C \rangle \text{ if } i \leq j \)

- **(Beta)** \( \langle (((\lambda t)[s]) v), C' \rangle \rightarrow_{\text{sec}} \langle t[v \cdot s], C' \rangle \)

- **(Prop)** \( \langle (t_0 \ t_1)[s], C \rangle \rightarrow_{\text{sec}} \langle (t_0[s]) (t_1[s]), C \rangle \)
\[(\text{Prop}_G) \ (\langle \text{grant } R \text{ in } t \rangle[s], C) \rightarrow_{\text{sec}} \langle \text{grant } R \text{ in } t[s], C \rangle \]
\[(\text{Prop}_F) \ (\langle R[t]\rangle[s], C) \rightarrow_{\text{sec}} \langle R[t[s], C \rangle \]

22

\[(\text{Prop}_T) \ (\langle \text{test } R \text{ then } t_0 \text{ else } t_1 \rangle[s], C) \rightarrow_{\text{sec}} \langle \text{test } R \text{ then } t_0[s] \text{ else } t_1[s], C \rangle \]
\[(\text{Frame}) \ (\langle R[v], C \rangle) \rightarrow_{\text{sec}} \langle v, C \rangle \]
\[(\text{Grant}) \ (\langle \text{grant } R \text{ in } v, C \rangle) \rightarrow_{\text{sec}} \langle v, C \rangle \]
\[(\text{Test}_1) \ (\langle \text{test } R \text{ then } c_1 \text{ else } c_2, C \rangle) \rightarrow_{\text{sec}} \langle c_1, C \rangle \text{ if } \text{OK}_{\text{sec}}(R, C) \]
\[(\text{Test}_2) \ (\langle \text{test } R \text{ then } c_1 \text{ else } c_2, C \rangle) \rightarrow_{\text{sec}} \langle c_2, C \rangle \text{ otherwise} \]
\[(\text{Fail}) \ (\langle \text{fail}[s], C \rangle) \rightarrow_{\text{sec}} \langle \text{fail}, [] \rangle \]

The only context-sensitive rules are (Test$_1$) and (Test$_2$), which perform a reduction step after inspecting the entire context $C$, and (Fail) which aborts the computation.

7.1.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

\[\langle i, s, C \rangle \Rightarrow_{\text{sec}} \langle C, s(i) \rangle\]
\[ \langle \lambda t, s, C \rangle \Rightarrow_{\text{sec}} \langle C, (\lambda t, s) \rangle \]
\[ \langle t_0 \ t_1, s, C \rangle \Rightarrow_{\text{sec}} \langle t_0, s, C[[[ (t_1, s)]]) \rangle \]
\[ \langle \text{grant } R \ \text{in } t, s, C \rangle \Rightarrow_{\text{sec}} \langle t, s, C[\text{grant } R \ \text{in } [ ]]) \rangle \]
\[ \langle \text{test } R \ \text{then } t_0 \ \text{else } t_1, s, C \rangle \Rightarrow_{\text{sec}} \langle t_0, s, C \rangle \ 	ext{if } \text{OK}_{\text{sec}}(R, C) \]
\[ \langle \text{test } R \ \text{then } t_0 \ \text{else } t_1, s, C \rangle \Rightarrow_{\text{sec}} \langle t_1, s, C \rangle \ 	ext{otherwise} \]
\[ \langle \text{fail}, s, C \rangle \Rightarrow_{\text{sec}} \text{fail} \]
\[ \langle R[t], s, C \rangle \Rightarrow_{\text{sec}} \langle t, s, C[R[[ ]]]) \rangle \]

\[ \langle [], v \rangle \Rightarrow_{\text{sec}} v \]
\[ \langle C[[[ (t, s)], v \rangle \Rightarrow_{\text{sec}} \langle t, s, C[v [ ]]]) \rangle \]
\[ \langle C[(\lambda t, s) [ ]], v \rangle \Rightarrow_{\text{sec}} \langle t, v \cdot s, C \rangle \]
\[ \langle C[\text{grant } R \ \text{in } [ ]], v \rangle \Rightarrow_{\text{sec}} \langle C, v \rangle \]
\[ \langle C[R[[ ]]], v \rangle \Rightarrow_{\text{sec}} \langle C, v \rangle \]

This machine is a storeless version of Clements and Felleisen's fg machine [16, Figure 1].

### 7.1.4 Formal correspondence

**Proposition 8.** For any term \( t \) in the \( \lambda \hat{\rho}_{\text{sec}} \)-calculus,

\[ t[\bullet] \Rightarrow^*_{\text{sec}} v \text{ if and only if } \langle t, \bullet, [] \rangle \Rightarrow^*_{\text{sec}} v. \]

The \( \lambda \hat{\rho}_{\text{sec}} \)-calculus therefore directly corresponds to the storeless version of the fg machine.
7.2 Properly tail-recursive stack inspection

On the ground that the fg machine is not properly tail-recursive, Clements and Felleisen presented a new, properly tail-recursive, machine—the cm machine [16, Figure 2]—thereby debunking the folklore that stack inspection is incompatible with proper tail recursion. Below, we consider the storeless version of the cm machine and we present the underlying calculus of closures.

23

7.2.1 The storeless cm machine

The cm machine operates on a \( \lambda_{sec} \)-term, an environment, and an evaluation context enriched with updatable permission tables (written \( m \) below):

\[
C ::= m[ ] | C[] (c, m) | C[(v, m)[ ]]
\]

A permission table is a partial function with a finite domain from a set of permissions \( \mathcal{P} \) to the set \( \{ \bot = \text{not granted}, \top = \text{granted} \} \). A permission table with the empty domain is written \( \varepsilon \).

Given the predicate \( \mathcal{OK}_{sec}^{cm}(R, C) \),

\[
R \cap m^{-1}(\bot) = \emptyset
\]
the transitions of the storeless cm machine read as follows:

\[
\begin{align*}
    \langle i, s, C \rangle &\xrightarrow{\text{cm sec}} \langle C, s(i) \rangle \\
    \langle \lambda t, s, C \rangle &\xrightarrow{\text{cm sec}} \langle C, (\lambda t, s) \rangle \\
    \langle t_0 t_1, s, C \rangle &\xrightarrow{\text{cm sec}} \langle t_0, s, C[[\cdot((t_1, s), \varepsilon)\cdot]] \rangle \\
    \langle \text{grant } R \text{ int}, s, C \rangle &\xrightarrow{\text{cm sec}} \langle t, s, C[R \leftarrow T] \rangle \\
    \langle \text{test } R \text{ then } t_0 \text{ else } t_1, s, C \rangle &\xrightarrow{\text{cm sec}} \langle t_0, s, C \rangle \text{ if } \mathcal{OK}^\text{cm}_\text{sec}(R, C) \\
    \langle \text{test } R \text{ then } t_0 \text{ else } t_1, s, C \rangle &\xrightarrow{\text{cm sec}} \langle t_1, s, C \rangle \text{ otherwise} \\
    \langle \text{fail}, s, C \rangle &\xrightarrow{\text{cm sec}} \text{fail} \\
    \langle R[t], s, C \rangle &\xrightarrow{\text{cm sec}} \langle t, s, C[R \leftarrow \bot] \rangle
\end{align*}
\]

where \( \overline{R} = P \setminus R \) and \( C[R \leftarrow v] \) is a modification of the permission table in the context \( C \) obtained by granting or restricting the permissions \( R \), depending on \( v \).

The following proposition states the equivalence of the fg machine and the cm machine
with respect to the values they compute:

**Proposition 9.** For any term $t$ in the $\lambda\hat{\rho}_{\sec}$-calculus,

\[
\langle t, \bullet, [] \rangle \Rightarrow^*_{\sec} v \text{ if and only if } \langle t, \bullet, [] \rangle (\Rightarrow^*_{\sec}) v.
\]

Moreover, it can be shown that each step of the fg machine is simulated by at most one step of the cm machine [16]. At the level of the calculus, this is reflected in the fact that the reduction semantics implemented by the cm machine has fewer reductions than $\lambda\hat{\rho}_{\sec}$.

### 7.2.2 The underlying calculus $\lambda\hat{\rho}_{\sec}^{cm}$

The calculus corresponding to the storeless cm machine is very close to the $\lambda\hat{\rho}_{\sec}$-calculus. The grammar of terms, closures and substitutions is the same, but the reduction contexts (which correspond to the stack frames in the machine) contain permission tables.

Consequently, the functions plug and decompose are defined in a non-standard way:

\[
\begin{align*}
\text{plug} (c, m[]) &= \text{build} (m, c) \\
\text{plug} (c_0, C[[[]](c_1, m)]) &= \text{plug} (\text{build} (m, c_0 c_1), C) \\
\text{plug} (c, C[(v, m)[[]]]) &= \text{plug} (\text{build} (m, v c), C)
\end{align*}
\]

where the auxiliary function build conservatively constructs a closure based on the permission table of the reduction context:
\[
\begin{align*}
\text{build}_G (m, c) &= \begin{cases} 
    c & \text{if } m^{-1}(\top) = \emptyset \\
    \text{grant } m^{-1}(\top) \text{ in } c & \text{otherwise}
\end{cases} \\
\text{build}_F (m, c) &= \begin{cases} 
    c & \text{if } m^{-1}(\bot) = \emptyset \\
    m^{-1}(\bot)[c] & \text{otherwise}
\end{cases} \\
\text{build} (m, c) &= \text{build}_F (m, \text{build}_G (m, c))
\end{align*}
\]

Any closure that is not already a value or a potential redex, can be further decomposed as follows:

\[
\begin{align*}
\text{decompose } (c_0 c_1, C) &= \text{decompose } (c_0, C[[[]](c_1, \varepsilon)]) \\
\text{decompose } (\text{grant } R \text{ in } c, C) &= \text{decompose } (c, C[R \mapsto \top]) \\
\text{decompose } (R[c], C) &= \text{decompose } (c, C[R \mapsto \bot]) \\
\text{decompose } (v, C[[[]](c, m))] &= \text{decompose } (c, C[(v, \varepsilon)][[]])
\end{align*}
\]

The notion of reduction includes most rules of the \(\lambda\rho_{\text{sec}}\)-calculus, except for (Frame) and (Grant).

From a calculus standpoint, Clements and Felleisen therefore obtained proper tail recursion by changing the computational model (witness the change from \(\mathcal{OK}_{\text{sec}}\) to \(\mathcal{OK}_{\text{cm}}\)) and by simplifying the reduction rules and modifying the compatibility rules.

### 7.3 State-based properly tail-recursive stack inspection

On the observation that the stack of the cm machine can be unzipped into the usual control stack of the CEK machine and a state-like list of permission tables, Ager et al. have presented an unzipped version of the cm machine (characterizing properly tail-recursive
7.3.1 The unzipped storeless cm machine

The unzipped cm machine operates on a $\lambda_{sec}$-term, an environment, and an ordinary evaluation context. In addition, the machine has a read-write security register $m$ holding the current permission table and a read-only security register $ms$ holding a list of outer permission tables. Given the predicate $OK_{sec}^{ucm}(R, m, ms)$,

\[
\begin{align*}
OK_{sec}^{ucm}(\emptyset, m, ms) & \quad R \cap m^{-1}(\bot) = \emptyset \\
OK_{sec}^{ucm}(R, m, \bullet) & \quad R \cap m^{-1}(\bot) = \emptyset \\
OK_{sec}^{ucm}(R \setminus m^{-1}(\top), m', ms) & \quad OK_{sec}^{ucm}(R, m, m' \cdot ms)
\end{align*}
\]

the transitions of the unzipped storeless cm machine read as follows:

\[
\begin{align*}
\langle i, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle C, s(i), ms \rangle \\
\langle \lambda t, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle C, (\lambda t, s), ms \rangle \\
\langle t_0 \; t_1, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle t_0, s, \varepsilon, m \cdot ms, C[[\;] (t_1, s)] \rangle \\
\langle grant \; R \; in \; t, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle t, s, m[R \rightarrow \top], ms, C \rangle \\
\langle test \; R \; then \; t_0 \; else \; t_1, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle t_0, s, m, ms, C \rangle \quad \text{if } OK_{sec}^{ucm}(R, m, ms) \\
\langle test \; R \; then \; t_0 \; else \; t_1, s, m, ms, C \rangle & \Rightarrow_{sec}^{ucm} \langle t_1, s, m, ms, C \rangle \quad \text{otherwise}
\end{align*}
\]
\[ \langle R[t], s, m, ms, C \rangle \Rightarrow_{sec}^{ucm} \langle t, s, m[R\leftarrow\bot], ms, C \rangle \]

\[ \langle \text{fail}, s, m, ms, C \rangle \Rightarrow_{sec}^{ucm} \text{fail} \]

\[ \langle [[], v, \bullet] \rangle \Rightarrow_{sec}^{ucm} v \]

\[ \langle C[[](t, s)], v, ms \rangle \Rightarrow_{sec}^{ucm} \langle t, s, \varepsilon, ms, C[v[]] \rangle \]

\[ \langle C[(\lambda t, s)[[]], v, m \cdot ms \rangle \Rightarrow_{sec}^{ucm} \langle t, v \cdot s, m, ms, C \rangle \]

The following proposition states the equivalence of the cm machine and the unzipped cm machine:

**Proposition 10.** For any term \( t \) in the \( \lambda \rho_{sec} \)-calculus,

\[ \langle t, \bullet, [] \rangle (\Rightarrow_{sec}^{cm})^* v \quad \text{if and only if} \quad \langle t, \bullet, \varepsilon, \bullet, [] \rangle (\Rightarrow_{sec}^{ucm})^* v. \]

Moreover, it can be shown that each step of the cm machine is simulated by one step of the unzipped cm machine.

### 7.3.2 The language of \( \lambda \rho_{sec}^{ucm} \)

(terms) \( t ::= i \mid \lambda t \mid tt \mid \text{grant } R \text{ in } t \mid \text{test } R \text{ then } t \text{ else } t \mid R[t] \mid \text{fail} \)

(closures) \( c ::= t[s] \)

(values) \( v ::= (\lambda t)[s] \mid \text{fail} \)

(substitutions) \( s ::= \bullet \mid c \cdot s \)

(reduction contexts) \( C ::= [] \mid C[[c] \mid C[v[]] \)

(annotated closures) \( \tilde{c} ::= c[m, ms] \mid c\tilde{c} \mid \tilde{c}c \mid \text{fail} \)
7.3.3 Notion of context-sensitive reduction

The notion of reduction is specified by the rules below. Compared to the rules of Section 7.1.2, the current permission table and the list of outer permission tables are propagated locally to each closure being evaluated. When a value is consumed, the current permission table is discarded.

The (Prop) reduction rule illustrates the propagation to a subclosure to be evaluated:

$$\langle (t_0 \ t_1)[s][m, ms], \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle (t_0[s][\varepsilon, m \cdot ms]) (t_1[s]), \ C \rangle$$

So do half of the other reduction rules:

(Var) \[ \langle i[c_1 \cdots c_j][m, ms], \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle c_i[m, ms], \ C \rangle \quad \text{if } i \leq j \]

(Test1) \[ \langle (\text{test } R \text{ then } t_0 \text{ else } t_1)[s][m, ms], \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle t_0[s][m, ms], \ C \rangle \quad \text{if } \text{OK}_{\text{sec}}^{\text{ucm}}(R, m, ms) \]

(Test2) \[ \langle (\text{test } R \text{ then } t_0 \text{ else } t_1)[s][m, ms], \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle t_1[s][m, ms], \ C \rangle \quad \text{otherwise} \]

(Fail) \[ \langle \text{fail}[s][m, ms], \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle \text{fail}, [\ ] \rangle \]

A new reduction rule, however, is now necessary to go from one evaluated subclosure to a subclosure to evaluate:

(Switch) \[ \langle (v[m, ms]) c, \ C \rangle \rightarrow_{\text{sec}}^{\text{ucm}} \langle v(c[\varepsilon, ms]), \ C \rangle \]

$\tilde{v} ::= v[m, ms] \mid \text{fail}$
The (Beta) rule doubles up with discarding the permission table of the actual parameter:

\[(\text{Beta}) \langle (((\lambda t)[s]) (v[m, m' \cdot ms]), C) \xrightarrow{\text{sec}} \text{ucm} \langle t[v \cdot s][m', ms], C]\]

Finally, the (Frame) and (Grant) rules embody the state counterpart of Clements and Felleisen’s design to enable proper tail recursion:

\[(\text{Frame}) \langle R[t][s][m, ms], C) \xrightarrow{\text{sec}} \text{ucm} \langle t[s][m[\overline{R} \leftarrow \bot], ms], C]\]

\[(\text{Grant}) \langle (\text{grant } R \text{ in } t)[s][m, ms], C) \xrightarrow{\text{sec}} \text{ucm} \langle t[s][m[R \leftarrow \top], ms], C]\]

7.3.4 Formal correspondence

**Proposition 11.** For any term \( t \) in the \( \lambda_{\text{sec}} \)-calculus,

\[ t[\bullet](\xrightarrow{\text{sec}}^{\text{ucm}})^* v \quad \text{if and only if} \quad \langle t, \bullet, \varepsilon, \bullet, [] \rangle (\Rightarrow_{\text{sec}}^{\text{ucm}})^* v.\]

7.4 Conclusion

We have presented three corresponding calculi of closures and machines for stack inspection, showing first how the storeless fg machine reflects the \( \lambda_{\rho_{\text{sec}}} \)-calculus, second, how the \( \lambda_{\rho_{\text{sec}}^{\text{cm}}} \)-calculus reflects the storeless cm machine, and third, how the \( \lambda_{\rho_{\text{sec}}^{\text{cm}}} \)-calculus reflects the unzipped storeless cm machine. In doing so, we have provided a calculus account of machine design and optimization for stack inspection.
8 A calculus for proper tail-recursion

At PLDI'98 [19], Clinger presented a properly tail-recursive semantics for Scheme in the form of a store-based abstract machine. This machine models the memory-allocation behavior of function calls in Scheme, and Clinger used it to specify in which sense an implementation should not run out of memory when processing a tail-recursive program (such as a program in CPS).

We first present a similar machine for the $\lambda$-calculus with left-to-right call-by-value evaluation and assignments. This machine is in the range of refocusing, transition compression, and closure unfolding, and so we next present the corresponding store-based calculus, $\lambda\widehat{\rho}_{ptr}$.

8.1 A simplified version of Clinger's abstract machine

Our simplified version of Clinger's abstract machine is an eval/apply machine with an environment and a store:

$$\langle x, s, C, \sigma \rangle \Rightarrow_{ptr} \langle C, \sigma(s(x)), \sigma \rangle$$

$$\langle \lambda x.t, s, C, \sigma \rangle \Rightarrow_{ptr} \langle C, (\lambda x.t, s), \sigma \rangle$$

$$\langle t_0 t_1, s, C, \sigma \rangle \Rightarrow_{ptr} \langle t_0, s, C[[\ ]](t_1, s), \sigma \rangle$$

$$\langle x := t, s, C, \sigma \rangle \Rightarrow_{ptr} \langle t, s, C[\text{upd}(s(x), [\ ])], \sigma \rangle$$

$$\langle [\ ] , v, \sigma \rangle \Rightarrow_{ptr} (v, \sigma)$$
(terms)
(closures)
(values)
(substitutions)
(red. contexts)
(store)
(store closures)
\[ \tilde{C} ::= C[\sigma] \]

### 8.3 Notion of context-sensitive reduction

In the rules below, (Var) dereferences the store; (Beta) allocates a fresh location, and extends both the substitution and the store with it; (Prop) is context-insensitive and therefore essentially as in the \(\lambda\rho\)-calculus; and (Upd) updates the store.

- **(Var)** \( \langle x_i[(x_1, l_1) \cdots (x_j, l_j)], C[\sigma] \rangle \rightarrow_{\text{ptr}} \langle \sigma(l_i), C[\sigma] \rangle \) if \( i \leq j \)
- **(Beta)** \( \langle ((\lambda x.t)[s]) v, C[\sigma] \rangle \rightarrow_{\text{ptr}} \langle t[(x, l) \cdot s], C[\sigma[l \mapsto v]] \rangle \)
  - if \( l \) does not occur within \( s, C, v, \sigma \)
- **(Prop)** \( \langle (t_0 \ t_1)[s], C[\sigma] \rangle \rightarrow_{\text{ptr}} \langle (t_0[s]) \ (t_1[s]), C[\sigma] \rangle \)
- **(Upd)** \( \langle \text{upd}(l, v), C[\sigma] \rangle \rightarrow_{\text{ptr}} \langle \sigma(l), C[\sigma[l \mapsto v]] \rangle \)

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the abstract machine of Section 8.1.

### 8.4 Formal correspondence

**Proposition 12.** For any term \( t \) in the \(\lambda\rho_{\text{ptr}}\)-calculus,

\[ t[\bullet][\bullet] \rightarrow^*_{\text{ptr}} v[\sigma] \text{ if and only if } \langle t, \bullet, [], \bullet \rangle \Rightarrow^*_{\text{ptr}} (v, \sigma). \]
The \( \lambda\rho_{\text{ptr}} \)-calculus therefore directly corresponds to the simplified version of Clinger's properly tail-recursive machine.

In Section 7, we showed storeless variants of two machines for stack inspection (the fg and the cm machines). The original versions of these machines use a store in the Clinger fashion [16], and we can exhibit their underlying calculi with an explicit representation of the store, as straightforward extensions of the storeless calculi. For conciseness, we do not include them here.

9 A lazy calculus of closures

The store-based account of proper tail-recursion from Section 8 suggests the following lazy calculus of closures, \( \lambda\rho_1 \).

9.1 The language of \( \lambda\rho_1 \)

The abstract syntax of the language is as follows:
(terms) (closures) (values) (substitutions)
(reduction contexts)

(store)

(store closures)

(store values)
\[
\begin{align*}
\text{store contexts} & \quad t ::= i \mid \lambda t \mid tt \\
& \quad c ::= t[s] \mid c[l] \mid \text{upd}(l, c) \\
& \quad v ::= (\lambda t)[s] \\
& \quad s ::= \bullet \cdot \\
& \quad C ::= [] \mid C[l] \\
\end{align*}
\]
\[\sigma ::= \bullet \mid \sigma[\ell \mapsto v]\]
\[\widetilde{c} ::= c[\sigma]\]
\[\widetilde{v} ::= v[\sigma]\]
\[\widetilde{C} ::= C[\sigma]\]

9.2 Notion of context-sensitive reduction

The notion of reduction is specified by the five rules shown below.

(\text{Var}_1) \quad \langle i[l_1 \cdots l_j], C[\sigma] \rangle \rightarrow_1 \langle v, C[\sigma] \rangle \quad \text{if } \sigma(l_i) = v

(\text{Var}_2) \quad \langle i[l_1 \cdots l_j], C[\sigma] \rangle \rightarrow_1 \langle \text{upd}(l_i, c), C[\sigma] \rangle \quad \text{if } \sigma(l_i) = c

(\text{Beta}) \quad \langle ((\lambda t)[s]) \ell, C[\sigma] \rangle \rightarrow_1 \langle t[l \cdot s], C[\sigma] \rangle

(\text{App}) \quad \langle (t_0 t_1)[s], C[\sigma] \rangle \rightarrow_1 \langle (t_0[s]) \ell, C[\sigma[l \mapsto t_1[s]]] \rangle \quad \text{where } \ell \text{ does not occur in } s, C, \sigma

(\text{Upd}) \quad \langle \text{upd}(\ell, v), C[\sigma] \rangle \rightarrow_1 \langle v, C[\sigma[l \mapsto v]] \rangle
Variables denote locations, and have two reduction rules, depending on whether the store holds a value or not at that location. In the former case—handled by (Var₁)—the result is this value, the current context, and the current store. In the latter case—handled by (Var₂)—a special closure upd(ℓ, c) is created, indicating that c is a shared computation. When this computation completes and yields a value, the store at location ℓ should be updated with this value, which is achieved by (Upd). Since every argument to an application can potentially be shared, (App) conservatively allocates a new location in the store for such shared closures. (Beta) extends the substitution with this location.

9.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following store-based machine:³

\[
\begin{align*}
\langle i, s, C, \sigma \rangle \Rightarrow_1 & \langle C, (\lambda t', s'), \sigma \rangle \quad \text{where } s(i) = ℓ \text{ and } \sigma(ℓ) = (\lambda t')[s'] \\
\langle i, s, C, \sigma \rangle \Rightarrow_1 & \langle t', s', C[\text{upd}(ℓ, [ ])], \sigma \rangle \quad \text{where } s(i) = ℓ \text{ and } \sigma(ℓ) = t'[s'] \\
\langle \lambda t, s, C, \sigma \rangle \Rightarrow_1 & \langle C, (\lambda t, s), \sigma \rangle \\
\langle t_0 \ t_1, s, C, \sigma \rangle \Rightarrow_1 & \langle t_0, s, C[[] \ ℓ], \sigma[ℓ \mapsto (t_1, s)] \rangle \quad \text{where } ℓ \text{ does not occur in } s, C, \sigma
\end{align*}
\]

³When a shared closure is to be evaluated, the current context is extended with what is known as an 'update marker' in the Three Instruction Machine (denoted C[upd(ℓ, [ ])] here).

\[
\langle [], v, \sigma \rangle \Rightarrow_1 (v, \sigma)
\]
This lazy abstract machine coincides with the one derived by Ager et al. out of a call-by-need interpreter for the $\lambda$-calculus [3], thereby connecting the present syntactic correspondence between calculi and abstract machines with the functional correspondence between abstract machines and evaluators [2, 4, 7, 28].

### 9.4 Formal correspondence

**Proposition 13.** For any term $t$ in the $\lambda\rho_1$-calculus,

$$t[\bullet][\bullet] \rightarrow^* v[\sigma] \quad \text{if and only if} \quad \langle t, \bullet, [\ ], \bullet \rangle \rightarrow^*_1 (v, \sigma).$$

The $\lambda\rho_1$-calculus therefore directly corresponds to call-by-need evaluation [85].

In $\lambda\rho_1$, sharing is made possible through a global heap where actual parameters are stored. On the other hand, a number of other calculi modeling call by need extend the set of terms with a *local* let-like construct, either by statically translating the source language into an intermediate language with explicit indications of sharing (as in Launchbury's approach [63]), or by providing dynamic reduction rules to the same effect (as in Ariola et al.'s calculus [5]). A sequence of let constructs binding variables to shared computations is a local version of a global heap where shared computations are bound to locations; extra reductions are then needed to propagate all the let constructs to the top level.

Another specificity of $\lambda\rho_1$ is that allocation occurs early, i.e., a new cell is allocated in the store every time an application is evaluated. Allocation, however, occurs late in Ariola
et al.'s semantics, i.e., a new binding is created only when the operator of the application is known to be a $\lambda$-abstraction. Delaying allocation is useful in the presence of strict functions, which we do not consider here.

We can construct a local version of our calculus with either of the store propagated inside closures or of late allocation. From there, one can mechanically derive the corresponding abstract machines.

### 10 Conclusion

We have presented a series of calculi and abstract machines accounting for a variety of computational effects, making it possible to directly reason about a computation in the calculus and in the corresponding abstract machine (horizontally in the diagram below) and to directly account for actual and explicit substitutions both in the world of calculi and in the world of abstract machines (vertically in the diagram below, where $\sigma$ maps a closure into the corresponding $\lambda$-term and an environment-machine configuration into a configuration in the corresponding machine with actual substitutions):

```
λ-calculus ← syntactic correspondence → machine with actual substitutions
   \sigma
   ↑
λ\hat{ρ}-calculus ← syntactic correspondence → environment-based machine
   \sigma
   ↑
```


The correspondence between each calculus and each abstract machine is simple and each can be mechanically built from the other. All of the calculi are new. Many of the abstract machines are known and have been independently designed and proved correct.

The work reported here leads us to drawing the following conclusions.

**Curien's calculus of closures:** Once extended to account for one-step reduction, Curien's calculus of closures directly corresponds to the notions of evaluation (i.e., weak-head normalization) accounted for by environment-based machines, even in the presence of computational effects (state and control).

**Refocusing:** Despite its pragmatic origin—fusing a plug function and a decomposition function in a reduction-based evaluation function to improve its efficiency [36], and in combination with compressing intermediate transitions and unfolding closures, refocusing proves consistently useful to construct reduction-free evaluation functions in the form of abstract machines, even in the presence of computational effects.

**Defunctionalization:** Despite its practical origin—representing a higher-order function as a data type and a dispatch function [72], defunctionalization proves consistently useful, witness the next item and also the fact that except for the abstract machines for $\lambda\rho F$ and the cm machine, all the abstract machines in this article are in defunctionalized form.
Reduction contexts and evaluation contexts: There are three objective reasons—one extensional and two intensional—why contexts are useful as well as, in some sense, unavoidable:

- reduction contexts are in one-to-one correspondence with the compatibility rules of a calculus;

- reduction contexts are the data type of the defunctionalized continuation of a one-step reduction function (as used in a reduction-based (weak-head) normalization function); and

- evaluation contexts are the data type of the defunctionalized continuation of an evaluation function (as used in a reduction-free (weak-head) normalization function).

If nothing else, each of these three reasons has practical value as a guideline for writing the grammar of reduction / evaluation contexts (which can be tricky in practice). But more significantly [27], reduction contexts and evaluation contexts coincide, which means that as a data type, they mediate between one-step reduction and evaluation, given an appropriate dispatch function:
Indeed, as initiated by Reynolds [29, 72], defunctionalizing a continuation-passing evaluator yields an abstract machine [2–4, 7], and as already pointed out above, a vast number of abstract machines are in defunctionalized form [9, 11, 28].

Together, the syntactic correspondence between calculi and abstract machines and the functional correspondence between abstract machines and evaluators therefore connect apparently distinct approaches to the same computational situations. We already illustrated this connection in Section 6 with delimited continuations; let us briefly illustrate it further with the simpler example of call/cc:

Call/cc was introduced in Scheme [17] as a Church encoding of Reynolds's escape operator [72]. A typed version of it is available in Standard ML of New Jersey [52] and Griffin has identified its logical content [49]. It is endowed with a variety of specifications: a CPS transformation [33], a CPS interpreter [53, 72], a denotational semantics [57], a computational monad [86], a big-step operational semantics [52], the CEK machine [43], calculi in the form of reduction semantics [42], and a number of implementation techniques [18, 25, 55]—not to mention its call-by-name version that allegedly accounts for the axiom of
choice in the archival version of Krivine’s machine [60].

Question: How do we know that all the artifacts in this semantic jungle define the same call/cc?

The elements of answer we contribute here are that the syntactic correspondence links calculi and abstract machines, and the functional correspondence links abstract machines and evaluators. So by construction, all these specifications are inter-derivable and therefore they are consistent.

**Normalization by evaluation:** Finally, refocusing provides a guideline for constructing reduction-free normalization functions out of reduction-based ones [26]. The reduction-free normalization functions take the form of eval/apply abstract machines, which usually are in defunctionalized form, which paves the way to writing normalization functions as usually encountered in the area of normalization by evaluation [31]. We have illustrated the method here with weak reduction and weak-head normalization (i.e., evaluation), but it also works for strong reduction and normalization, thus linking one-step reduction, abstract machines for strong reduction, and normalization functions.

**Acknowledgments**

We are grateful to Julia Lawall and Kristian Støvring for commenting a preliminary version of this article.

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-
References


University of Aarhus, Aarhus, Denmark, May 2005.


[26] 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


[43] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine,


[58] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, March 2005.


[84] Hayo Thielecke. An introduction to Landin’s “A generalization of jumps and labels”.


Recent BRICS Report Series Publications


cessing Letters. This version supersedes BRICS RS-05-14.