A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines

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

Version of December 17, 2006 at 16:15

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 they 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.

To appear in TCS (revised version of BRICS RS-05-22).

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

Contents

1 Introduction

How does one construct a new semantic artifact?

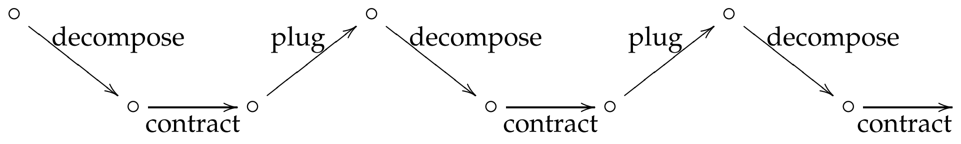
1.1 Calculi and machines

Sixty-five years ago, the λ-calculus was introduced [15]. Forty-five years ago, its expressive power was observed to be relevant for computing [68, 84]. Forty years ago, a first abstract machine for the λ-calculus was introduced [62]. Thirty years ago, calculi and abstract machines were formally connected [71]. Twenty years ago, a calculus format—reduction semantics—with an explicit representation of reduction contexts was introduced [41]. 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 [43].

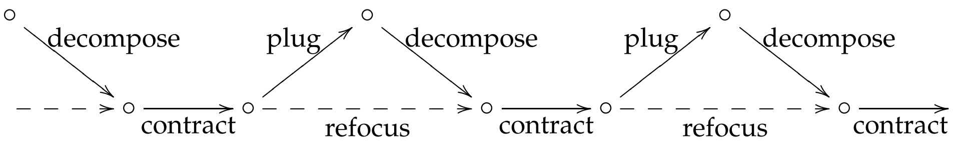
From calculus to machine by refocusing and transition compression: Recently, Danvy and Nielsen have pointed out that the reduction strategy for a calculus actually determines the structure of the corresponding machine [36]. They present a method for constructing an abstract machine out of a reduction 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 by iterating 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 by iterating 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 λ-calculus with normal-order reduction, the resulting abstract machine is a substitution-based version of the Krivine machine (i.e., a push/enter machine); for the pure λ-calculus with applicative-order reduction, 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, 52, 66, 76].

From calculus to machine by refocusing, transition compression, and closure unfolding: Recently, we have applied the refocusing method to $\lambda\widehat{\rho}$, 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 $\lambda\widehat{\rho}$ with left-to-right applicative order directly corresponds to the CEK machine [44], how $\lambda\widehat{\rho}$ with normal order directly corresponds to the Krivine machine [20, 23], how $\lambda\widehat{\rho}$ with normal order and generalized reduction directly corresponds to the original version of Krivine's machine [60], and how $\lambda\widehat{\rho}$ with right-to-left applicative order and generalized reduction directly corresponds to the ZINC abstract machine [65]. 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 [41, 44]. In these calculi, reduction rules are not oblivious to their reduction context; on the contrary, they are <u>context</u> <u>sensitive</u> 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, 48, 72].

From calculus to machine by refocusing, transition compression, and closure unfolding: In this article, we apply the refocusing method to context-sensitive extensions of $\lambda\widehat{\rho}$ 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 [71] or via a compilation / decompilation scheme in the course of execution [52]).

1.4 Overview

We successively consider call by name: Krivine's machine with call/cc (Section 4) and the $\lambda\mu$-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 $\lambda\widehat{\rho}$-calculus and the notion of context-sensitive reduction (Section 2) and then a detailed walkthrough for the $\lambda\widehat{\rho}$-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 [41,43] consists in 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 [36, Figure 2, page 8]:

Property 1. *For each syntactic construct building a term out of n subterms, there is a number $0 \leq i \leq n$ and a fixed traversal order of subterms encoded in the grammar of the reduction contexts for this construct such that the holes of these contexts are precisely the positions of the i subterms. Furthermore, a term with all the chosen i subterms reduced to values is either a value or a 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 [41].

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 $\lambda\widehat{\rho}$-calculus with call/cc), there is an implicit global control delimiter around the program [42].

[1]For example, a potential redex in the λ-calculus is the application of a value to a term. If the value is a λ-abstraction, the potential redex is an actual one and it can be β-reduced. If no reduction rule applies, the potential redex is not an actual one and the program is stuck [71].

2.3 Our base calculus of closures: $\lambda\widehat{\rho}$

Since Landin [62], most abstract machines implementing variants and extensions of the λ-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, 76] to mediate between the traditional abstract specifications of the λ-calculus and its traditional concrete implementations [52].

To derive an abstract machine for evaluating λ-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 λ-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 $\lambda\widehat{\rho}$-calculus.

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

$$\begin{array}{ll} \text{(terms)} & t ::= i \mid \lambda t \mid t\, t \\ \text{(closures)} & c ::= t[s] \mid c\, c \\ \text{(substitutions)} & s ::= \varnothing \mid c \cdot s \end{array}$$

(For comparison, $\lambda\rho$ does not have the $c\,c$ 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 $t[\varnothing]$ where t does not contain free variables.

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

$$\begin{array}{lll} \text{(Var)} & i[c_1 \cdots c_j] \overset{\widehat{\rho}}{\to} c_i & \text{if } i \leq j \\ \text{(Beta)} & ((\lambda t)[s])\, c \overset{\widehat{\rho}}{\to} t[c \cdot s] & \\ \text{(Prop)} & (t_0\, t_1)[s] \overset{\widehat{\rho}}{\to} (t_0[s])\, (t_1[s]) & \end{array}$$

We write $s(i)$ for the ith 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:

$$\begin{array}{lcl} \text{(L-Comp)} & \dfrac{c_0 \overset{\widehat{\rho}}{\to} c_0'}{c_0\, c_1 \overset{\widehat{\rho}}{\to} c_0'\, c_1} & \\ \text{(R-Comp)} & \dfrac{c_1 \overset{\widehat{\rho}}{\to} c_1'}{c_0\, c_1 \overset{\widehat{\rho}}{\to} c_0\, c_1'} & \\ \text{(Sub)} & \dfrac{c_i \overset{\widehat{\rho}}{\to} c_i'}{t[c_1 \cdots c_i \cdots c_j] \overset{\widehat{\rho}}{\to} t[c_1 \cdots c_i' \cdots c_j]} & \text{for } i \leq j \end{array}$$

These rules bijectively correspond to the following grammar of reduction contexts (minus the first production):

$$\text{(reduction contexts)} \quad C ::= [\,] \mid C[[\,]\, c] \mid C[c\, [\,]] \mid C[t[c_1 \cdots [\,] \cdots c_j]]$$

Specific, deterministic reduction strategies can be obtained by restricting the compatibility rules and thus the grammar of reduction contexts. In the following sections, we consider two such strategies:

the normal-order strategy: it is obtained in the usual way by discarding the (Sub) and (R-Comp) rules and by phrasing the grammars of values, redexes and reduction contexts as follows:

$$\begin{array}{ll}
\text{(values)} & v ::= (\lambda t)[s] \\
\text{(redexes)} & r ::= i[s] \mid v\,c \mid (t_0\,t_1)[s] \\
\text{(reduction contexts)} & C ::= [\,] \mid C[[\,]\,c]
\end{array}$$

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

$$\begin{array}{ll}
\text{(values)} & v ::= (\lambda t)[s] \\
\text{(substitutions)} & s ::= \varnothing \mid v \cdot s \\
\text{(redexes)} & r ::= i[s] \mid v\,v \mid (t_0\,t_1)[s] \\
\text{(reduction contexts)} & C ::= [\,] \mid C[[\,]\,c] \mid C[v\,[\,]]
\end{array}$$

All of the calculi presented in this article extend the $\lambda\widehat{\rho}$-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 $\lambda\widehat{\rho}\mathcal{A}$-calculus

As an illustration, we present a detailed and systematic derivation of an abstract machine for call-by-value evaluation in the λ-calculus with an abort operation, starting from the specification of the applicative-order reduction strategy in the $\lambda\widehat{\rho}$-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 iteration of one-step reduction.

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

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

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

Section 3.5: 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 [43].

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

3.1 A reduction semantics for applicative order and abort

A reduction semantics for applicative-order reduction in the $\lambda\widehat{\rho}\mathcal{A}$-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:

$$\begin{array}{ll} (\mathsf{Term}) & t ::= \ldots \mid \mathcal{A}\, t \\ (\mathsf{Closure}) & c ::= \ldots \mid \mathcal{A}\, c \end{array}$$

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

$$\begin{array}{lrcll} (\mathrm{Var}) & \langle i[v_1 \cdots v_j],\, C\rangle & \rightarrow_{\mathcal{A}} & \langle v_i,\, C\rangle & \text{if } i \leq j \\ (\mathrm{Beta}) & \langle((\lambda t)[s])\, v,\, C\rangle & \rightarrow_{\mathcal{A}} & \langle t[v \cdot s],\, C\rangle & \\ (\mathrm{Prop}) & \langle(t_0\, t_1)[s],\, C\rangle & \rightarrow_{\mathcal{A}} & \langle(t_0[s])\, (t_1[s]),\, C\rangle & \end{array}$$

as well as two new contractions for the abort operation:

$$\begin{array}{lrcl} (\mathrm{Prop}_{\mathcal{A}}) & \langle(\mathcal{A}\, t)[s],\, C\rangle & \rightarrow_{\mathcal{A}} & \langle\mathcal{A}\,(t[s]),\, C\rangle \\ (\mathrm{Abort}) & \langle\mathcal{A}\, v,\, C\rangle & \rightarrow_{\mathcal{A}} & \langle v,\, [\,]\rangle \end{array}$$

The last contraction resets the context.

Finally, the grammar of reduction contexts reads as follows:

$$(\mathsf{Context}) \qquad C ::= [\,] \mid C[[\,]\, c] \mid C[v\, [\,]] \mid C[\mathcal{A}\, [\,]]$$

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

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

3.1.1 Decomposition

We define $\mathsf{decompose}$ as a state-transition function over closures and reduction contexts:

$$\begin{aligned}
\mathsf{decompose} : \mathsf{Closure} &\to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{decompose}\; c &= \mathsf{decompose}'\,(c,\, [\,])
\end{aligned}$$

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

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

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

- $\mathsf{decompose}'$ is passed a closure and a reduction context. It dispatches on the closure and iteratively builds the reduction context:
  - if the current closure is a value, then $\mathsf{decompose}'_{\mathsf{aux}}$ is called to inspect the context;
  - if the current closure is a redex, then a decomposition is found; and
  - otherwise, a subclosure of the current closure is chosen to be visited in a new context.
- $\mathsf{decompose}'_{\mathsf{aux}}$ is passed a reduction context and a value. It dispatches on the reduction context:
  - if the current context is empty, then the value is the result of the function;
  - 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. (It is also in defunctionalized form [26].)

3.1.2 Context-sensitive contraction

We define $\mathsf{contract}$ by cases, as a straightforward implementation of the contraction rules:

$$\begin{aligned}
\mathsf{contract} : \mathsf{Redex} \times \mathsf{Context} &\to \mathsf{Closure} \times \mathsf{Context} \\
\mathsf{contract}\,(i[v_1 \cdots v_m],\, C) &= (v_i,\, C) \\
\mathsf{contract}\,((t_0\, t_1)[s],\, C) &= ((t_0[s])\, (t_1[s]),\, C) \\
\mathsf{contract}\,(((\lambda t)[s])\, v,\, C) &= (t[v \cdot s],\, C) \\
\mathsf{contract}\,((\mathcal{A}\, t)[s],\, C) &= (\mathcal{A}\, (t[s]),\, C) \\
\mathsf{contract}\,(\mathcal{A}\, v,\, C) &= (v,\, [\,])
\end{aligned}$$

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 state-transition function:

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

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:

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

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', $c \rightarrow_{\mathcal{A}} c' \Leftrightarrow \mathsf{reduce}\, c = c'$.

3.1.5 Reduction-based evaluation

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

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

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

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

Proposition 2. *For any closed term* t *and any value* v, $t[\varnothing] \rightarrow_{\mathcal{A}}^{*} v \Leftrightarrow \mathsf{evaluate}\, t = v$.

3.2 A pre-abstract machine

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

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & \\
& \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & \circ
\end{tikzcd}$$

At each step, an intermediate term is constructed by the function plug; it is then immediately decomposed by the subsequent call to decompose. In their 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:

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & \\
{} \arrow[r, dashed] & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & \circ
\end{tikzcd}$$

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

Proposition 3 (Danvy & Nielsen [26,36]). *For any closure* c *and reduction context* C,

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

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:

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

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

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 refocus and by the first, third, and fourth clauses of $\text{refocus}_{\text{aux}}$ which all lead to an accepting state; and the initial transition is specified by two clauses of the corresponding 'refocused' evaluation function, which reads as follows:

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

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

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

This evaluation function computes the iteration of refocus and contract using the auxiliary function iterate as a trampoline [49]. Due to the non-tail call to refocus in iterate, we refer to this evaluation function as a 'pre-abstract machine' in that it is a trivial state machine with two configurations (one with a value and one pairing a redex and a context) and a state-transition function, the composition of refocus and contract.

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 $\text{refocus}_{\text{aux}}$:

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

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

$$\begin{aligned}
\text{refocus} : \text{Closure} \times \text{Context} &\rightarrow \text{Value}\\
\text{refocus}\ (i[s],\ C) &= \text{iterate}\ (i[s],\ C)\\
\text{refocus}\ ((\lambda t)[s],\ C) &= \text{refocus}_{\text{aux}}\ (C,\ (\lambda t)[s])\\
\text{refocus}\ ((t_0\, t_1)[s],\ C) &= \text{iterate}\ ((t_0\, t_1)[s],\ C)\\
\text{refocus}\ (c_0\, c_1,\ C) &= \text{refocus}\ (c_0,\ C[[\,]\, c_1])\\
\text{refocus}\ ((\mathcal{A}\, t)[s],\ C) &= \text{iterate}\ ((\mathcal{A}\, t)[s],\ C)\\
\text{refocus}\ (\mathcal{A}\, c,\ C) &= \text{refocus}\ (c,\ C[\mathcal{A}\, [\,]])
\end{aligned}$$

$$\begin{aligned}
\text{refocus}_{\text{aux}} : \text{Context} \times \text{Value} &\rightarrow \text{Value}\\
\text{refocus}_{\text{aux}}\ ([\,],\ v) &= \text{iterate}\ v\\
\text{refocus}_{\text{aux}}\ (C[[\,]\, c],\ v) &= \text{refocus}\ (c,\ C[v\, [\,]])\\
\text{refocus}_{\text{aux}}\ (C[v'\, [\,]],\ v) &= \text{iterate}\ (v'\, v,\ C)\\
\text{refocus}_{\text{aux}}\ (C[\mathcal{A}\, [\,]],\ v) &= \text{iterate}\ (\mathcal{A}\, v,\ C)
\end{aligned}$$

The resulting definitions of evaluate, iterate, refocus, and $\text{refocus}_{\text{aux}}$ are that of four mutually recursive state-transition functions that form an abstract machine, where 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. In this abstract machine, the compatibility rules are implemented by refocus and $\text{refocus}_{\text{aux}}$, and the reduction rules by the call to contract in the second clause of iterate. We can make this last point even more manifest by inlining contract in the definition of iterate:

$$\begin{aligned}
\text{iterate} : \text{Value} + (\text{Redex} \times \text{Context}) &\rightarrow \text{Value}\\
\text{iterate}\ v &= v\\
\text{iterate}\ (i[v_1 \cdots v_m],\ C) &= \text{refocus}\ (v_i,\ C)\\
\text{iterate}\ ((t_0\, t_1)[s],\ C) &= \text{refocus}\ ((t_0[s])\ (t_1[s]),\ C)\\
\text{iterate}\ (((\lambda t)[s])\ v,\ C) &= \text{refocus}\ (t[v \cdot s],\ C)\\
\text{iterate}\ ((\mathcal{A}\, t)[s],\ C) &= \text{refocus}\ (\mathcal{A}\ (t[s]),\ C)\\
\text{iterate}\ (\mathcal{A}\, v,\ C) &= \text{refocus}\ (v,\ [\,])
\end{aligned}$$

By construction, the machine therefore separately implements the reduction rules (with iterate) and the compatibility rules (with refocus and $\text{refocus}_{\text{aux}}$); for this reason, we refer to it as a 'staged abstract machine' [52].

3.4 An eval/apply abstract machine

As already observed by Danvy and Nielsen in their work on refocusing, inlining $\mathsf{iterate}$ yields an eval/apply abstract machine [67]. Inlining the calls to $\mathsf{iterate}$ in the staged abstract machine yields the following eval/apply machine, where $\mathsf{refocus}$ (the 'eval' transition function) dispatches on closures and $\mathsf{refocus_{aux}}$ (the 'apply' function) dispatches on contexts:

$$\begin{aligned}
\mathsf{evaluate} : \mathsf{Term} &\to \mathsf{Value}\\
\mathsf{evaluate}\ t &= \mathsf{refocus}\ (t[\varnothing],\ [\,])\\[1ex]
\mathsf{refocus} : \mathsf{Closure} \times \mathsf{Context} &\to \mathsf{Value}\\
\mathsf{refocus}\ (i[v_1 \cdots v_m],\ C) &= \mathsf{refocus}\ (v_i,\ C)\\
\mathsf{refocus}\ ((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\ (C,\ (\lambda t)[s])\\
\mathsf{refocus}\ ((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\ ((t_0[s])\ (t_1[s]),\ C)\\
\mathsf{refocus}\ (c_0\, c_1,\ C) &= \mathsf{refocus}\ (c_0,\ C[[\,]\ c_1])\\
\mathsf{refocus}\ ((\mathcal{A}\ t)[s],\ C) &= \mathsf{refocus}\ (\mathcal{A}\ (t[s]),\ C)\\
\mathsf{refocus}\ (\mathcal{A}\ c,\ C) &= \mathsf{refocus}\ (c,\ C[\mathcal{A}\ [\,]])\\[1ex]
\mathsf{refocus_{aux}} : \mathsf{Context} \times \mathsf{Value} &\to \mathsf{Value}\\
\mathsf{refocus_{aux}}\ ([\,],\ v) &= v\\
\mathsf{refocus_{aux}}\ (C[[\,]\ c],\ v) &= \mathsf{refocus}\ (c,\ C[v\ [\,]])\\
\mathsf{refocus_{aux}}\ (C[((\lambda t)[s])\ [\,]],\ v) &= \mathsf{refocus}\ (t[v \cdot s],\ C)\\
\mathsf{refocus_{aux}}\ (C[\mathcal{A}\ [\,]],\ v) &= \mathsf{refocus}\ (v,\ [\,])
\end{aligned}$$

3.5 Transition compression

In four cases, the eval/apply machine of Section 3.4 yields a configuration that uniquely determines the subsequent transition. Let us shortcut these "corridor transitions:"

- Substitutions contain only values and therefore the (Var) reduction step (first clause of $\mathsf{refocus}$) always produces a value and is thus always followed by a shift to $\mathsf{refocus_{aux}}$ (second clause of $\mathsf{refocus}$). As a shortcut, we coalesce the two consecutive transitions into one, replacing the first clause of $\mathsf{refocus}$ with the following one:

$$\mathsf{refocus}\ (i[v_1 \cdots v_m],\ C) = \mathsf{refocus_{aux}}\ (C,\ v_i)$$

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

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

- The machine only produces an application of $\mathcal{A}$ through an (Abort) reduction step (second-to-last clause of $\mathsf{refocus}$). We observe that in a reduction sequence, an (Abort) reduction step is always followed by a decomposition step (last clause of $\mathsf{refocus}$). As a shortcut, we coalesce the two consecutive transitions into one, replacing the last two clauses of $\mathsf{refocus}$ with the following one:

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

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

$$\mathsf{refocus_{aux}}\ (C[\mathcal{A}\ [\,]],\ v) = v$$

3.6 An environment machine

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

$$
\begin{array}{lrl}
(\mathsf{Value}) & v ::= & (\lambda t,\, s) \\
(\mathsf{Context}) & C ::= & [\,] \mid C[[\,]\,(t,\, s)] \mid C[v\,[\,]] \mid C[\mathcal{A}\,[\,]]
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathsf{refocus} : \mathsf{Term} \times \mathsf{Substitution} \times \mathsf{Context} & \rightarrow & \mathsf{Value} \\
\mathsf{refocus}\,(i,\, v_1 \cdots v_m,\, C) & = & \mathsf{refocus_{aux}}\,(C,\, v_i) \\
\mathsf{refocus}\,(\lambda t,\, s,\, C) & = & \mathsf{refocus_{aux}}\,(C,\, (\lambda t,\, s)) \\
\mathsf{refocus}\,(t_0\, t_1,\, s,\, C) & = & \mathsf{refocus}\,(t_0,\, s,\, C[[\,]\,(t_1,\, s)]) \\
\mathsf{refocus}\,(\mathcal{A}\, t,\, s,\, C) & = & \mathsf{refocus}\,(t,\, s,\, C[\mathcal{A}\,[\,]])
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{refocus_{aux}} : \mathsf{Context} \times \mathsf{Value} & \rightarrow & \mathsf{Value} \\
\mathsf{refocus_{aux}}\,([\,],\, v) & = & v \\
\mathsf{refocus_{aux}}\,(C[[\,]\,(t,\, s)],\, v) & = & \mathsf{refocus}\,(t,\, s,\, C[v\,[\,]]) \\
\mathsf{refocus_{aux}}\,(C[(\lambda t,\, s)\,[\,]],\, v) & = & \mathsf{refocus}\,(t,\, v \cdot s,\, C) \\
\mathsf{refocus_{aux}}\,(C[\mathcal{A}\,[\,]],\, v) & = & v
\end{array}
$$

We observe that this machine coincides with the CEK machine extended with an abort operator—an extension that was designed as such [43] and is in defunctionalized form [2]. 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 $\lambda\widehat{\rho}\mathcal{A}$-calculus.

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

$$t[\varnothing] \rightarrow_{\mathcal{A}}^{*} (\lambda t')[s'] \quad \textit{if and only if} \quad \mathsf{evaluate}\; t = (\lambda t',\, s').$$

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

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 $\lambda\widehat{\rho}\mathcal{A}$-calculus, which makes it a call-by-value machine [71]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-value evaluation in the $\lambda\widehat{\rho}\mathcal{A}$-calculus.

3.8 Conclusion

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

The Krivine machine is probably the best-known abstract machine implementing the normal-order reduction strategy in the λ-calculus [30]. In our previous work [8], we have pointed out that Krivine's original machine [60] does not coincide with the Krivine Machine As We Know It [21, 23] in that it implements generalized instead of ordinary β-reduction: indeed Krivine's machine reduces the term $(\lambda\lambda t)\, t_1\, t_2$ in one step whereas the Krivine machine reduces it in two steps. Furthermore, an extension of the archival version of Krivine's machine [61, Section 3] also caters for a call-by-name variant of call/cc (noted $\mathcal{K}$ below).

In our previous work [8], we have presented the calculus and the reduction strategy 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, $\lambda\widehat{\rho}$.

Here, we present the calculus corresponding to the archival version of Krivine's machine extended with $\mathcal{K}$. This machine also uses closures and an environment. Correspondingly, the calculus is one of explicit substitutions, $\lambda\widehat{\rho}\mathcal{K}$. We build on top of Krivine's language of terms by specifying syntactic categories of closures and substitutions as shown below. Like Krivine, we consider the normal-order reduction strategy and therefore call by name [71].

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

The abstract syntax of the language is as follows:

$$\begin{array}{lr}
\text{(terms)} & t ::= i \mid \lambda^n t \mid t\, t \mid \mathcal{K}\, t \\
\text{(closures)} & c ::= t[s] \mid c\; c \mid \mathcal{K}\, c \mid \ulcorner C \urcorner \\
\text{(values)} & v ::= (\lambda^n t)[s] \mid \ulcorner C \urcorner \\
\text{(substitutions)} & s ::= \varnothing \mid c \cdot s \\
\text{(reduction contexts)} & C ::= [\,] \mid C[[\,]\, c] \mid C[\mathcal{K}[\,]]
\end{array}$$

A nested λ-abstraction of the form $\lambda^n t$ is to be understood as a syntactic abbreviation for $\underbrace{\lambda\lambda\ldots\lambda}_{n} t$, where t is not a λ-abstraction.
In $\lambda\widehat{\rho}\mathcal{K}$, a value is either a closure with a λ-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 rules shown below. (Var) and (Prop) are as in the $\lambda\widehat{\rho}$-calculus, and (Beta$^+$) supersedes (Beta) in that it performs a generalized β-reduction in one step:

$$\begin{array}{lr}
\text{(Var)} & \langle i[c_1 \cdots c_j],\, C\rangle \to_{\mathcal{K}} \langle c_i,\, C\rangle \ \text{ if } i \leq j \\
\text{(Beta}^+\text{)} & \langle (\lambda^n t)[s],\, C[[\ldots[[\,]\, c_n]\, \ldots]\, c_1]\rangle \to_{\mathcal{K}} \langle t[c_n \cdots c_1 \cdot s],\, C\rangle \\
\text{(Beta}_C\text{)} & \langle \ulcorner C' \urcorner,\, C[[\,]\, c]\rangle \to_{\mathcal{K}} \langle c,\, C'\rangle \\
\text{(Prop)} & \langle (t_0\, t_1)[s],\, C\rangle \to_{\mathcal{K}} \langle (t_0[s])\, (t_1[s]),\, C\rangle
\end{array}$$

$$\begin{array}{lrcl}
(\mathrm{Prop}_{\mathcal{K}}) & \langle(\mathcal{K}\, t)[s],\, C\rangle & \rightarrow_{\mathcal{K}} & \langle\mathcal{K}\,(t[s]),\, C\rangle \\
(\mathcal{K}_{\lambda}) & \langle\mathcal{K}\,((\lambda t)[s]),\, C\rangle & \rightarrow_{\mathcal{K}} & \langle t[\ulcorner C\urcorner \cdot s],\, C\rangle \\
(\mathcal{K}_{C}) & \langle\mathcal{K}\,\ulcorner C'\urcorner,\, C\rangle & \rightarrow_{\mathcal{K}} & \langle\ulcorner C'\urcorner\,\ulcorner C\urcorner,\, C\rangle
\end{array}$$

The three last rules account for call/cc: the first is an ordinary propagation rule, and the two others describe the capture of a context. In the first case, the current context is captured and passed to a function, and in the second, it is captured and passed to an already captured context. In (Beta$_C$), a captured context is reinstated in place of the current context, which is tossed away.

4.3 Krivine's machine

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

$$\begin{array}{rcl}
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathcal{K}} & \langle t',\, s',\, C\rangle \ \text{ if } s(i) = (t',\, s') \\
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathcal{K}} & \langle C,\, \ulcorner C'\urcorner\rangle \ \text{ if } s(i) = \ulcorner C'\urcorner \\
\langle \lambda^{n} t,\, s,\, C\rangle & \Rightarrow_{\mathcal{K}} & \langle C,\, (\lambda^{n} t,\, s)\rangle \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathcal{K}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle \\
\langle \mathcal{K}\, t,\, s,\, C\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\, s,\, C[\mathcal{K}[\,]]\rangle \\
\\
\langle [\,],\, v\rangle & \Rightarrow_{\mathcal{K}} & v \\
\langle C[[\ldots[[\,]\, c_n]\, \ldots]\, c_1],\, (\lambda^{n} t,\, s)\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\, c_n \cdots c_1 \cdot s,\, C\rangle \\
\langle C[[\,]\,(t,\, s)],\, \ulcorner C'\urcorner\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\, s,\, C'\rangle \\
\langle C[[\,]\,\ulcorner C''\urcorner],\, \ulcorner C'\urcorner\rangle & \Rightarrow_{\mathcal{K}} & \langle C',\, \ulcorner C''\urcorner\rangle \\
\langle C[\mathcal{K}[\,]],\, v\rangle & \Rightarrow_{\mathcal{K}} & \langle C[[\,]\,\ulcorner C\urcorner],\, v\rangle
\end{array}$$

This machine evaluates a closed term t by starting in the configuration $\langle t,\, \varnothing,\, [\,]\rangle$. It halts with value v if it reaches a configuration $\langle [\,],\, v\rangle$. Its definition coincides with that of the extension of Krivine's machine with $\mathcal{K}$—an extension which was designed as such and not connected to any calculus [61, Section 3]. Because of the generalized reduction, it is not in defunctionalized form.

4.4 Formal correspondence

Proposition 4. *For any term* t *in the* $\lambda\widehat{\rho}\mathcal{K}$*-calculus,*

$$t[\varnothing] \rightarrow^{*}_{\mathcal{K}} v \quad \textit{if and only if} \quad \langle t,\, \varnothing,\, [\,]\rangle \Rightarrow^{*}_{\mathcal{K}} v.$$

The $\lambda\widehat{\rho}\mathcal{K}$-calculus therefore directly corresponds to the archival version of Krivine's machine with call/cc.

5 The $\lambda\widehat{\rho}\mu$-calculus

In this section we present a calculus of closures that extends Parigot's $\lambda\mu$-calculus [70] 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 μ (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 $\lambda\widehat{\rho}$-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, though, we omit all the typing considerations and concentrate on the syntactic correspondence between the calculus and the machine.

5.1 The language of $\lambda\widehat{\rho}\mu$

We use de Bruijn indices for both the λ-bound variables and the μ-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:

$$\begin{array}{ll}
\text{(terms)} & t ::= i \mid \lambda t \mid t\,t \mid \mu t \mid [i]t \\
\text{(closures)} & c ::= t[s] \mid c\,c \\
\text{(values)} & v ::= (\lambda t)[s] \\
\text{(substitutions)} & s ::= \varnothing \mid C \cdot s \mid c \cdot s \\
\text{(reduction contexts)} & C ::= [\,] \mid C[[\,]\,c]
\end{array}$$

We consider only closed λ-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 `tp` in Ariola et al.'s setting [6]). A substitution is a non-empty sequence of either closures—to be substituted for λ-bound variables, or captured reduction contexts—to be used when accessing μ-bound variables.

Programs are closures of the form $t[[\,] \cdot \varnothing]$, 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 $\lambda\widehat{\rho}$ 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 μ-bound variable is applied in an empty context:

$$\begin{array}{lrcl}
\text{(Beta)} & \langle(\lambda t)[s],\, C[[\,]\,c]\rangle & \rightarrow_\mu & \langle t[c \cdot s],\, C\rangle \\
\text{(Var)} & \langle i[s],\, C\rangle & \rightarrow_\mu & \langle c,\, C\rangle \quad \text{if } s(i) = c \\
\text{(Prop)} & \langle(t_0\,t_1)[s],\, C\rangle & \rightarrow_\mu & \langle(t_0[s])\,(t_1[s]),\, C\rangle \\
\text{(Mu)} & \langle(\mu t)[s],\, C\rangle & \rightarrow_\mu & \langle t[C \cdot s],\, [\,]\rangle \\
\text{(Rho)} & \langle([i]t)[s],\, [\,]\rangle & \rightarrow_\mu & \langle t[s],\, C\rangle \quad \text{if } s(i) = C
\end{array}$$

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:

$$\begin{array}{rcl}
\langle \lambda t,\, s,\, C\rangle & \Rightarrow_\mu & \langle C,\, (\lambda t,\, s)\rangle \\
\langle i,\, s,\, C\rangle & \Rightarrow_\mu & \langle t',\, s',\, C\rangle \quad \text{if } s(i) = (t',\, s') \\
\langle t_0\,t_1,\, s,\, C\rangle & \Rightarrow_\mu & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle \\
\langle \mu t,\, s,\, C\rangle & \Rightarrow_\mu & \langle t,\, C \cdot s,\, [\,]\rangle \\
\langle [i]t,\, s,\, [\,]\rangle & \Rightarrow_\mu & \langle t,\, s,\, C\rangle \quad \text{if } s(i) = C
\end{array}$$

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

This machine evaluates a closed term t by starting in the configuration $\langle t,\ [\,] \cdot \varnothing,\ [\,] \rangle$. It halts with value v if it reaches a configuration $\langle [\,], v \rangle$. Its definition coincides with that of 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. It is in defunctionalized form.

5.4 Formal correspondence

Proposition 5. *For any term* t *in the* $\lambda\widehat{\rho}\mu$*-calculus,*

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

The $\lambda\widehat{\rho}\mu$-calculus therefore directly corresponds to de Groote's abstract machine for the $\lambda\mu$-calculus, and a similar story can be told for an applicative-order reduction strategy and the corresponding call-by-value machine.

6 Delimited continuations

Continuations have been discovered multiple times [73], but they acquired their name for describing jumps [85], using what is now known as continuation-passing style (CPS) [83]. A full-fledged control operator, J [63, 88], 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 [34].

Conversely, delimited continuations, in the form of the traditional success and failure continuations and to CPS [77], have been regularly used in artificial-intelligence programming [14, 55, 86] for generators and backtracking. They also occur in the study of reflective towers [82], where the notions of meta-continuation [93] and of "jumpy" vs. "pushy" continuations [33] arose. A full-fledged delimited control operator, # (pronounced "prompt"), however, was introduced independently of CPS and of reflective towers, to support operational equivalence in λ-calculi with first-class control [42,45]. Only subsequently were control delimiters connected to success and failure continuations [31].

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 [42], and an extension of CPS, as discovered by Danvy and Filinski [31]:

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, 74]); 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 the left-to-right call-by-value evaluation strategy.

6.1 The $\lambda\widehat{\rho}\mathcal{S}$-calculus

The standard λ-calculus is extended with the control operator shift (written $\mathcal{S}$) 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 $\lambda\widehat{\rho}\mathcal{S}$

The abstract syntax of the language is as follows:

$$\begin{array}{ll}
\text{(terms)} & t ::= i \mid \lambda t \mid t\,t \mid \mathcal{S}\,t \mid \langle t\rangle \\
\text{(closures)} & c ::= t[s] \mid c\,c \mid \mathcal{S}\,c \mid \langle c\rangle \mid \ulcorner C_1 \urcorner \\
\text{(values)} & v ::= (\lambda t)[s] \mid \ulcorner C_1 \urcorner \\
\text{(substitutions)} & s ::= \varnothing \mid v \cdot s \\
\text{(contexts)} & C_1 ::= [\,] \mid C_1[[\,]\,c] \mid C_1[v\,[\,]] \mid C_1[\mathcal{S}[\,]] \\
\text{(meta-contexts)} & C_2 ::= \bullet \mid C_1 \cdot C_2
\end{array}$$

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

The control operator $\mathcal{S}$ 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 [31]. 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 [67]) 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:

$$\begin{array}{rcl}
\langle i,\ s,\ C_1,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\ v,\ C_2\rangle \quad \text{if } s(i) = v \\
\langle \lambda t,\ s,\ C_1,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\ (\lambda t,\ s),\ C_2\rangle \\
\langle t_0\,t_1,\ s,\ C_1,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t_0,\ s,\ C_1[[\,]\ (t_1,\ s)],\ C_2\rangle \\
\langle \mathcal{S}\,t,\ s,\ C_1,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\ s,\ C_1[\mathcal{S}[\,]],\ C_2\rangle \\
\langle \langle t\rangle,\ s,\ C_1,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\ s,\ [\,],\ C_1 \cdot C_2\rangle \\
 & & \\
\langle [\,],\ v,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_2,\ v\rangle \\
\langle C_1[[\,]\ (t,\ s)],\ v,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\ s,\ C_1[v\ [\,]],\ C_2\rangle \\
\langle C_1[(\lambda t,\ s)\ [\,]],\ v,\ C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\ v \cdot s,\ C_1,\ C_2\rangle
\end{array}$$

$$
\begin{aligned}
\langle C_1[\ulcorner C_1' \urcorner\, [\,]],\, v,\, C_2\rangle &\Rightarrow_{\mathcal{S}} \langle C_1',\, v,\, C_1 \cdot C_2\rangle\\
\langle C_1[\mathcal{S}[\,]],\, (\lambda t,\, s),\, C_2\rangle &\Rightarrow_{\mathcal{S}} \langle t,\, \ulcorner C_1 \urcorner \cdot s,\, [\,],\, C_2\rangle\\
\langle C_1[\mathcal{S}[\,]],\, \ulcorner C_1' \urcorner,\, C_2\rangle &\Rightarrow_{\mathcal{S}} \langle C_1',\, \ulcorner C_1 \urcorner,\, [\,] \cdot C_2\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \bullet,\, v\rangle &\Rightarrow_{\mathcal{S}} v\\
\langle C_1 \cdot C_2,\, v\rangle &\Rightarrow_{\mathcal{S}} \langle C_1,\, v,\, C_2\rangle
\end{aligned}
$$

This machine evaluates a closed term t by starting in the configuration $\langle t,\, \varnothing,\, [\,],\, \bullet\rangle$. It halts with value v if it reaches a configuration $\langle \bullet,\, v\rangle$. As pointed out initially, it is in defunctionalized form.

We have observed that this machine is in the range of refocusing, transition compression, and closure unfolding for the following calculus $\lambda\widehat{\rho}\mathcal{S}$.

6.1.3 Notion of context-sensitive reduction

The $\lambda\widehat{\rho}\mathcal{S}$-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:

$$
\begin{array}{lrl}
(\mathrm{Var}) & \langle i[v_1 \cdots v_j],\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle v_i,\, C_1,\, C_2\rangle \quad \text{if } i \leq j\\
(\mathrm{Beta}) & \langle((\lambda t)[s])\, v,\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle t[v \cdot s],\, C_1,\, C_2\rangle\\
(\mathrm{Beta}_C) & \langle \ulcorner C_1' \urcorner v\, ,\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle \langle C_1'[v]\rangle,\, C_1,\, C_2\rangle\\
(\mathrm{Prop}) & \langle (t_0\, t_1)[s],\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle (t_0[s])\, (t_1[s]),\, C_1,\, C_2\rangle\\
(\mathrm{Prop}_{\mathcal{S}}) & \langle (\mathcal{S}\, t)[s],\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle \mathcal{S}\, (t[s]),\, C_1,\, C_2\rangle\\
(\mathrm{Prop}_{\langle\cdot\rangle}) & \langle \langle t\rangle[s],\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle \langle t[s]\rangle,\, C_1,\, C_2\rangle\\
(\mathcal{S}_\lambda) & \langle \mathcal{S}\, ((\lambda t)[s]),\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle t[\ulcorner C_1 \urcorner \cdot s],\, [\,],\, C_2\rangle\\
(\mathcal{S}_C) & \langle \mathcal{S}\, \ulcorner C_1' \urcorner,\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle \ulcorner C_1' \urcorner\, \ulcorner C_1 \urcorner,\, [\,],\, C_2\rangle\\
(\mathrm{Reset}) & \langle \langle v\rangle,\, C_1,\, C_2\rangle \rightarrow_{\mathcal{S}} & \langle v,\, C_1,\, C_2\rangle
\end{array}
$$

Since none of the contractions depends on the meta-context, it is evident that the notion of reduction $\rightarrow_{\mathcal{S}}$ is compatible with meta-contexts. It is, however, not compatible with contexts, due to $\mathcal{S}_\lambda$ and $\mathcal{S}_C$. 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* $\lambda\widehat{\rho}\mathcal{S}$*-calculus,*

$$
t[\varnothing] \rightarrow_{\mathcal{S}}^{*} v \quad \textit{if and only if} \quad \langle t,\, \varnothing,\, [\,],\, \bullet\rangle \Rightarrow_{\mathcal{S}}^{*} v.
$$

The $\lambda\widehat{\rho}\mathcal{S}$-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 λ-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 [31, 37] which Filinski has shown to be able to represent layered monads [47]. Recently, Kameyama has proposed an axiomatization of the CPS hierarchy [57]. Elsewhere [7], we have studied its defunctionalized counterpart and the corresponding hierarchy of calculi.

6.2 The $\lambda\widehat{\rho}\mathcal{F}$-calculus

The standard λ-calculus is extended with the control operator $\mathcal{F}$ 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 $\lambda\widehat{\rho}\mathcal{F}$

The abstract syntax of the language is as follows:

$$\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \mid \mathcal{F}\,t \mid \#t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \mid \mathcal{F}\,c \mid \#\,c \mid \ulcorner C \urcorner \\
\text{(values)} & v & ::= & (\lambda t)[s] \mid \ulcorner C \urcorner \\
\text{(substitutions)} & s & ::= & \varnothing \mid v \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \mid C[\mathcal{F}[\,]] \mid C[\#[\,]]
\end{array}$$

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]

$$\begin{array}{lrcll}
(\text{Var}) & \langle i[v_1 \cdots v_j],\, C\rangle & \rightarrow_{\mathcal{F}} & \langle v_i,\, C\rangle & \text{if } i \leq j \\
(\text{Beta}) & \langle ((\lambda t)[s])\, v,\, C\rangle & \rightarrow_{\mathcal{F}} & \langle t[v \cdot s],\, C\rangle & \\
(\text{Beta}_C) & \langle \ulcorner C' \urcorner v\,,\, C\rangle & \rightarrow_{\mathcal{F}} & \langle C'[v],\, C\rangle & \\
(\text{Prop}) & \langle (t_0\, t_1)[s],\, C\rangle & \rightarrow_{\mathcal{F}} & \langle (t_0[s])\,(t_1[s]),\, C\rangle & \\
(\text{Prop}_{\mathcal{F}}) & \langle (\mathcal{F}\, t)[s],\, C\rangle & \rightarrow_{\mathcal{F}} & \langle \mathcal{F}\,(t[s]),\, C\rangle & \\
(\text{Prop}_{\#}) & \langle (\#t)[s],\, C\rangle & \rightarrow_{\mathcal{F}} & \langle \#\,(t[s]),\, C\rangle & \\
(\mathcal{F}_\lambda) & \langle \mathcal{F}\,((\lambda t)[s]),\, C[\#C']\rangle & \rightarrow_{\mathcal{F}} & \langle t[\ulcorner C' \urcorner \cdot s],\, C\rangle & \text{if } C' \text{ contains no mark} \\
(\mathcal{F}_C) & \langle \mathcal{F}\, \ulcorner C'' \urcorner,\, C[\#C']\rangle & \rightarrow_{\mathcal{F}} & \langle \ulcorner C'' \urcorner\, \ulcorner C' \urcorner,\, C\rangle & \text{if } C' \text{ contains no mark} \\
(\text{Prompt}) & \langle \#v,\, C\rangle & \rightarrow_{\mathcal{F}} & \langle v,\, C\rangle &
\end{array}$$

Alternatively, we could specify the reduction rules using two layers of contexts, similarly to the $\lambda\widehat{\rho}\mathcal{S}$-calculus [7,10,11]. The difference between the two calculi would then be only in the rule (Beta_C):

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

where there is no delimiter around $C_1'[v]$. As in the previous case of the $\lambda\widehat{\rho}\mathcal{S}$-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.

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:

$$\begin{array}{rcll}
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, v\rangle & \text{if } s(i) = v \\
\langle \lambda t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, (\lambda t,\, s)\rangle & \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle & \\
\langle \mathcal{F}\, t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\mathcal{F}[\,]]\rangle & \\
\langle \#t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\#[\,]]\rangle &
\end{array}$$

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

$$
\begin{array}{rcll}
\langle [\,], v\rangle & \Rightarrow_{\mathcal{F}} & v & \\
\langle C[[\,]\,(t, s)], v\rangle & \Rightarrow_{\mathcal{F}} & \langle t, s, C[v\,[\,]]\rangle & \\
\langle C[(\lambda t, s)\,[\,]], v\rangle & \Rightarrow_{\mathcal{F}} & \langle t, v \cdot s, C\rangle & \\
\langle C[\ulcorner C' \urcorner\,[\,]], v\rangle & \Rightarrow_{\mathcal{F}} & \langle C' \circ C, v\rangle & \\
\langle C[\#C'[\mathcal{F}[\,]]], (\lambda t, s)\rangle & \Rightarrow_{\mathcal{F}} & \langle t, \ulcorner C' \urcorner \cdot s, C\rangle & \text{where } C' \text{ contains no mark} \\
\langle C[\#C'[\mathcal{F}[\,]]], \ulcorner C'' \urcorner\rangle & \Rightarrow_{\mathcal{F}} & \langle C'' \circ C, \ulcorner C' \urcorner\rangle & \text{where } C' \text{ contains no mark} \\
\langle C[\#[\,]], v\rangle & \Rightarrow_{\mathcal{F}} & \langle C, v\rangle &
\end{array}
$$

where $\circ$ concatenates contexts: ...

This machine evaluates a closed term t by starting in the configuration $\langle t, \varnothing, [\,]\rangle$. It halts with value v if it reaches a configuration $\langle [\,], v\rangle$. Its definition coincides with that of Felleisen's extension of the CEK machine—an extension which was designed as such [42, Section 3] and is not in defunctionalized form because of the concatenation of contexts [11].

6.2.4 Formal correspondence

Proposition 7. *For any term* t *in the* $\lambda\widehat{\rho}\mathcal{F}$*-calculus,*

$$t[\varnothing] \rightarrow^*_{\mathcal{F}} v \quad \textit{if and only if} \quad \langle t, \varnothing, [\,]\rangle \Rightarrow^*_{\mathcal{F}} v.$$

This proposition parallels Felleisen's second correspondence theorem [42, p. 186]. The $\lambda\widehat{\rho}\mathcal{F}$-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 [81], 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 [81, 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. [51] 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 [42]. Elsewhere, we present the abstract machine [7] and the evaluator [31] 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 compatible 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 (Beta_C).
- 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 a 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." [46, 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, 59, 80]. 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 [46] 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" [45], 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 λ_{sec}-calculus, which formalizes security enforcement by stack inspection [48]. We first present a calculus of closures built on top of the λ_{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 $\lambda\widehat{\rho}_{\text{sec}}$-calculus

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

(terms)	$t ::= i \mid \lambda t \mid t\ t \mid \texttt{grant}\ R\ \texttt{in}\ t \mid \texttt{test}\ R\ \texttt{then}\ t\ \texttt{else}\ t \mid R[t] \mid \texttt{fail}$
(closures)	$c ::= t[s] \mid c\ c \mid \texttt{grant}\ R\ \texttt{in}\ c \mid \texttt{test}\ R\ \texttt{then}\ c\ \texttt{else}\ c \mid R[c]$
(values)	$v ::= (\lambda t)[s] \mid \texttt{fail}$
(substitutions)	$s ::= \varnothing \mid v \cdot s$
(reduction contexts)	$C ::= [\,] \mid C[[\,]\ c] \mid C[v\ [\,]] \mid C[\texttt{grant}\ R\ \texttt{in}\ [\,]] \mid C[R[[\,]]]$
(permissions)	$R \subseteq \mathcal{P}$

The set of terms consists of λ-terms and four constructs for handling different levels of security specified in a set $\mathcal{P}$: $\texttt{grant}\ R\ \texttt{in}\ t$ grants the permissions R to t; $\texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1$ 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, $\texttt{fail}$ 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,

$$\frac{}{\mathcal{OK}_{\text{sec}}(\emptyset, C)} \qquad \frac{}{\mathcal{OK}_{\text{sec}}(R, [\,])}$$

$$\frac{\mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[[\,]\ c])} \qquad \frac{\mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[v\ [\,]])}$$

$$\frac{R \subset R' \quad \mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[R'[[\,]]])} \qquad \frac{\mathcal{OK}_{\text{sec}}(R \setminus R', C)}{\mathcal{OK}_{\text{sec}}(R, C[\texttt{grant}\ R'\ \texttt{in}\ [\,]])}$$

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

(Var)	$\langle i[v_1 \cdots v_j],\ C\rangle \to_{\text{sec}} \langle v_i,\ C\rangle$	if $i \leq j$
(Beta)	$\langle ((\lambda t)[s])\ v,\ C\rangle \to_{\text{sec}} \langle t[v \cdot s],\ C\rangle$	
(Prop)	$\langle (t_0\ t_1)[s],\ C\rangle \to_{\text{sec}} \langle (t_0[s])\ (t_1[s]),\ C\rangle$	
(Prop$_G$)	$\langle (\texttt{grant}\ R\ \texttt{in}\ t)[s],\ C\rangle \to_{\text{sec}} \langle \texttt{grant}\ R\ \texttt{in}\ t[s],\ C\rangle$	
(Prop$_F$)	$\langle (R[t])[s],\ C\rangle \to_{\text{sec}} \langle R[t[s]],\ C\rangle$	
(Prop$_T$)	$\langle (\texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1)[s],\ C\rangle \to_{\text{sec}} \langle \texttt{test}\ R\ \texttt{then}\ t_0[s]\ \texttt{else}\ t_1[s],\ C\rangle$	
(Frame)	$\langle R[v],\ C\rangle \to_{\text{sec}} \langle v,\ C\rangle$	
(Grant)	$\langle \texttt{grant}\ R\ \texttt{in}\ v,\ C\rangle \to_{\text{sec}} \langle v,\ C\rangle$	
(Test$_1$)	$\langle \texttt{test}\ R\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ C\rangle \to_{\text{sec}} \langle c_1,\ C\rangle$	if $\mathcal{OK}_{\text{sec}}(R, C)$ holds
(Test$_2$)	$\langle \texttt{test}\ R\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ C\rangle \to_{\text{sec}} \langle c_2,\ C\rangle$	otherwise
(Fail)	$\langle \texttt{fail}[s],\ C\rangle \to_{\text{sec}} \langle \texttt{fail},\ [\,]\rangle$	

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:

$$
\begin{array}{rcll}
\langle \mathrm{i},\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\ \nu \rangle & \text{if } \mathrm{s}(\mathrm{i}) = \nu \\
\langle \lambda \mathrm{t},\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\ (\lambda \mathrm{t},\ \mathrm{s}) \rangle & \\
\langle \mathrm{t}_0\, \mathrm{t}_1,\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t}_0,\ \mathrm{s},\ C[[\,]\ (\mathrm{t}_1,\ \mathrm{s})] \rangle & \\
\langle \texttt{grant}\ \mathrm{R}\ \texttt{in}\ \mathrm{t},\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t},\ \mathrm{s},\ C[\texttt{grant}\ \mathrm{R}\ \texttt{in}\ [\,]] \rangle & \\
\langle \texttt{test}\ \mathrm{R}\ \texttt{then}\ \mathrm{t}_0\ \texttt{else}\ \mathrm{t}_1,\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t}_0,\ \mathrm{s},\ C \rangle & \text{if } \mathcal{OK}_{\mathrm{sec}}(\mathrm{R}, C) \text{ holds} \\
\langle \texttt{test}\ \mathrm{R}\ \texttt{then}\ \mathrm{t}_0\ \texttt{else}\ \mathrm{t}_1,\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t}_1,\ \mathrm{s},\ C \rangle & \text{otherwise} \\
\langle \texttt{fail},\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \texttt{fail} & \\
\langle \mathrm{R}[\mathrm{t}],\ \mathrm{s},\ C \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t},\ \mathrm{s},\ C[\mathrm{R}[[\,]]] \rangle & \\
& & & \\
\langle [\,],\ \nu \rangle & \Rightarrow_{\mathrm{sec}} & \nu & \\
\langle C[[\,]\ (\mathrm{t},\ \mathrm{s})],\ \nu \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t},\ \mathrm{s},\ C[\nu\ [\,]] \rangle & \\
\langle C[(\lambda \mathrm{t},\ \mathrm{s})\ [\,]],\ \nu \rangle & \Rightarrow_{\mathrm{sec}} & \langle \mathrm{t},\ \nu \cdot \mathrm{s},\ C \rangle & \\
\langle C[\texttt{grant}\ \mathrm{R}\ \texttt{in}\ [\,]],\ \nu \rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\ \nu \rangle & \\
\langle C[\mathrm{R}[[\,]]],\ \nu \rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\ \nu \rangle &
\end{array}
$$

This machine evaluates a closed term t by starting in the configuration $\langle \mathrm{t},\ \varnothing,\ [\,] \rangle$. It halts with value ν if it reaches a configuration $\langle [\,],\ \nu \rangle$. It is a storeless version of Clements and Felleisen's fg machine [16, Figure 1], and is in defunctionalized form.

7.1.4 Formal correspondence

Proposition 8. *For any term* t *in the* $\lambda\widehat{\rho}_{\mathrm{sec}}$*-calculus,*

$$\mathrm{t}[\varnothing] \rightarrow^*_{\mathrm{sec}} \nu \quad \textit{if and only if} \quad \langle \mathrm{t},\ \varnothing,\ [\,] \rangle \Rightarrow^*_{\mathrm{sec}} \nu.$$

The $\lambda\widehat{\rho}_{\mathrm{sec}}$-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.

7.2.1 The storeless cm machine

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

$$\text{(stack frames)} \quad C ::= m[\,] \mid C[[\,](\mathrm{c}, m)] \mid C[(\nu, m)[\,]]$$

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 ε.

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

$$\frac{}{\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\emptyset, C)} \qquad \frac{\mathrm{R} \cap m^{-1}(\bot) = \emptyset}{\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathrm{R}, m[\,])}$$

$$\frac{\mathsf{R} \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathsf{R} \setminus m^{-1}(\top), C)}{\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathsf{R}, C[[\,](\mathsf{c}, m)])} \qquad \frac{\mathsf{R} \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathsf{R} \setminus m^{-1}(\top), C)}{\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathsf{R}, C[(\nu, m)[\,]])}$$

the transitions of the storeless cm machine read as follows:

$$\begin{array}{rcll}
\langle \mathsf{i},\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle C,\ \nu\rangle & \text{if } \mathsf{s}(\mathsf{i}) = \nu \\
\langle \lambda\mathsf{t},\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle C,\ (\lambda\mathsf{t},\ \mathsf{s})\rangle & \\
\langle \mathsf{t}_0\,\mathsf{t}_1,\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t}_0,\ \mathsf{s},\ C[[\,]((\mathsf{t}_1,\ \mathsf{s}), \varepsilon)]\rangle & \\
\langle \mathtt{grant}\ \mathsf{R}\ \mathtt{in}\ \mathsf{t},\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t},\ \mathsf{s},\ C[\mathsf{R} \mapsto \top]\rangle & \\
\langle \mathtt{test}\ \mathsf{R}\ \mathtt{then}\ \mathsf{t}_0\ \mathtt{else}\ \mathsf{t}_1,\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t}_0,\ \mathsf{s},\ C\rangle & \text{if } \mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}(\mathsf{R}, C) \text{ holds} \\
\langle \mathtt{test}\ \mathsf{R}\ \mathtt{then}\ \mathsf{t}_0\ \mathtt{else}\ \mathsf{t}_1,\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t}_1,\ \mathsf{s},\ C\rangle & \text{otherwise} \\
\langle \mathtt{fail},\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \mathtt{fail} & \\
\langle \mathsf{R}[\mathsf{t}],\ \mathsf{s},\ C\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t},\ \mathsf{s},\ C[\overline{\mathsf{R}} \mapsto \bot]\rangle & \\
\\
\langle m[\,],\ \nu\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \nu & \\
\langle C[[\,]((\mathsf{t},\ \mathsf{s}), m)],\ \nu\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t},\ \mathsf{s},\ C[(\nu, \varepsilon)[\,]]\rangle & \\
\langle C[((\lambda\mathsf{t},\ \mathsf{s}), m)[\,]],\ \nu\rangle & \Rightarrow^{\mathrm{cm}}_{\mathrm{sec}} & \langle \mathsf{t},\ \nu \cdot \mathsf{s},\ C\rangle &
\end{array}$$

where $\overline{\mathsf{R}} = \mathcal{P} \setminus \mathsf{R}$ and $C[\mathsf{R} \mapsto \nu]$ is a modification of the permission table in the context C obtained by granting or restricting the permissions R, depending on ν. This machine evaluates a closed term t by starting in the configuration $\langle \mathsf{t},\ \varnothing,\ [\,]\rangle$. It halts with value ν if it reaches a configuration $\langle m[\,],\ \nu\rangle$. It is not in defunctionalized form because of the modification of the permission tables in the contexts.

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* $\lambda\widehat{\rho}_{\mathrm{sec}}$*-calculus,*

$$\langle \mathsf{t},\ \varnothing,\ [\,]\rangle \Rightarrow^{*}_{\mathrm{sec}} \nu \quad \textit{if and only if} \quad \langle \mathsf{t},\ \varnothing,\ [\,]\rangle\ (\Rightarrow^{\mathrm{cm}}_{\mathrm{sec}})^{*}\ \nu.$$

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

7.2.2 The underlying calculus $\lambda\widehat{\rho}^{\mathrm{cm}}_{\mathrm{sec}}$

The calculus corresponding to the storeless cm machine is very close to the $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus. The grammars of terms, closures and substitutions are 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:

$$\begin{array}{lcl}
\mathsf{plug}\ (\mathsf{c}, m[\,]) & = & \mathsf{build}\ (m, \mathsf{c}) \\
\mathsf{plug}\ (\mathsf{c}_0, C[[\,](\mathsf{c}_1, m)]) & = & \mathsf{plug}\ (\mathsf{build}\ (m, \mathsf{c}_0\,\mathsf{c}_1), C) \\
\mathsf{plug}\ (\mathsf{c}, C[(\nu, m)[\,]]) & = & \mathsf{plug}\ (\mathsf{build}\ (m, \nu\,\mathsf{c}), C)
\end{array}$$

where the auxiliary function build conservatively constructs a closure based on the permission table of the reduction context:

$$\begin{array}{lcl}
\mathsf{build}_{\mathsf{G}}\ (m, \mathsf{c}) & = & \begin{cases} \mathsf{c} & \text{if } m^{-1}(\top) = \emptyset \\ \mathtt{grant}\ m^{-1}(\top)\ \mathtt{in}\ \mathsf{c} & \text{otherwise} \end{cases} \\
\mathsf{build}_{\mathsf{F}}\ (m, \mathsf{c}) & = & \begin{cases} \mathsf{c} & \text{if } m^{-1}(\bot) = \emptyset \\ \overline{m^{-1}(\bot)}[\mathsf{c}] & \text{otherwise} \end{cases} \\
\mathsf{build}\ (m, \mathsf{c}) & = & \mathsf{build}_{\mathsf{F}}\ (m, \mathsf{build}_{\mathsf{G}}\ (m, \mathsf{c}))
\end{array}$$

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

$$\begin{array}{lcl}
\text{decompose}\ (c_0\, c_1, C) & = & \text{decompose}\ (c_0, C[[\,](c_1, \varepsilon)]) \\
\text{decompose}\ (\texttt{grant}\ R\ \texttt{in}\ c, C) & = & \text{decompose}\ (c, C[R \mapsto \top]) \\
\text{decompose}\ (R[c], C) & = & \text{decompose}\ (c, C[\overline{R} \mapsto \bot]) \\
\text{decompose}\ (v, C[[\,](c, m)]) & = & \text{decompose}\ (c, C[(v, \varepsilon)[\,]])
\end{array}$$

The notion of reduction includes most rules of the $\lambda\widehat{\rho}_{\text{sec}}$-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 $\mathcal{OK}^{\text{cm}}_{\text{sec}}$) 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 stack inspection as a monad, in passing) [4]. We first present this machine, and then the corresponding calculus of closures.

7.3.1 The unzipped storeless cm machine

The unzipped cm machine operates on a λ_{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 $\mathcal{OK}^{\text{ucm}}_{\text{sec}}(R, m, ms)$,

$$\frac{}{\mathcal{OK}^{\text{ucm}}_{\text{sec}}(\emptyset, m, ms)} \qquad \frac{R \cap m^{-1}(\bot) = \emptyset}{\mathcal{OK}^{\text{ucm}}_{\text{sec}}(R, m, \bullet)} \qquad \frac{R \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\text{ucm}}_{\text{sec}}(R \setminus m^{-1}(\top), m', ms)}{\mathcal{OK}^{\text{ucm}}_{\text{sec}}(R, m, m' \cdot ms)}$$

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

$$\begin{array}{rcll}
\langle i,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle C,\ v,\ ms\rangle & \text{if } s(i) = v \\
\langle \lambda t,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle C,\ (\lambda t,\ s),\ ms\rangle & \\
\langle t_0\, t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t_0,\ s,\ \varepsilon, m \cdot ms,\ C[[\,]\ (t_1,\ s)]\rangle & \\
\langle \texttt{grant}\ R\ \texttt{in}\ t,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t,\ s,\ m[R \mapsto \top], ms,\ C\rangle & \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t_0,\ s,\ m, ms,\ C\rangle & \text{if } \mathcal{OK}^{\text{ucm}}_{\text{sec}}(R, m, ms) \text{ holds} \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t_1,\ s,\ m, ms,\ C\rangle & \text{otherwise} \\
\langle R[t],\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t,\ s,\ m[\overline{R} \mapsto \bot], ms,\ C\rangle & \\
\langle \texttt{fail},\ s,\ m, ms,\ C\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \texttt{fail} & \\
\\
\langle [\,],\ v,\ \bullet\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & v & \\
\langle C[[\,]\ (t,\ s)],\ v,\ ms\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t,\ s,\ \varepsilon, ms,\ C[v\ [\,]]\rangle & \\
\langle C[(\lambda t,\ s)\ [\,]],\ v,\ m \cdot ms\rangle & \Rightarrow^{\text{ucm}}_{\text{sec}} & \langle t,\ v \cdot s,\ m, ms,\ C\rangle &
\end{array}$$

This machine evaluates a closed term t by starting in the configuration $\langle t,\ \varnothing,\ \varepsilon, \bullet,\ [\,]\rangle$. It halts with value v if it reaches a configuration $\langle [\,],\ v,\ \bullet\rangle$. It is in defunctionalized form.

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

Proposition 10. *For any term* t *in the* $\lambda\widehat{\rho}_{\text{sec}}$*-calculus,*

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

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 $\lambda\widehat{\rho}^{\mathrm{ucm}}_{\mathrm{sec}}$

(terms)	$t ::= i \mid \lambda t \mid t\,t \mid \texttt{grant}\,R\,\texttt{in}\,t \mid \texttt{test}\,R\,\texttt{then}\,t\,\texttt{else}\,t \mid R[t] \mid \texttt{fail}$
(closures)	$c ::= t[s]$
(values)	$v ::= (\lambda t)[s] \mid \texttt{fail}$
(substitutions)	$s ::= \varnothing \mid v \cdot s$
(reduction contexts)	$C ::= [\,] \mid C[[\,]\,c] \mid C[v\,[\,]]$
(annotated closures)	$\widetilde{c} ::= c[m, ms] \mid c\,\widetilde{c} \mid \widetilde{c}\,c \mid \texttt{fail}$
(annotated values)	$\widetilde{v} ::= v[m, ms] \mid \texttt{fail}$

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:

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

So do half of the other reduction rules:

(Var)	$\langle i[c_1 \cdots c_j][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle c_i[m, ms],\, C\rangle \quad \text{if } i \leq j$
(Test$_1$)	$\langle (\texttt{test}\,R\,\texttt{then}\,t_0\,\texttt{else}\,t_1)[s][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle t_0[s][m, ms],\, C\rangle$ if $\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, ms)$ holds
(Test$_2$)	$\langle (\texttt{test}\,R\,\texttt{then}\,t_0\,\texttt{else}\,t_1)[s][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle t_1[s][m, ms],\, C\rangle$ otherwise
(Fail)	$\langle \texttt{fail}[s][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle \texttt{fail},\, [\,]\rangle$

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

$$\text{(Switch)} \qquad \langle (v[m, ms])\,c,\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle v\,(c[\varepsilon, ms]),\, C\rangle$$

The (Beta) rule doubles up with discarding the permission table of the actual parameter:

$$\text{(Beta)}\ \langle ((\lambda t)[s])\,(v[m, m' \cdot ms]),\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle t[v \cdot s][m', ms],\, C\rangle$$

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

$$\text{(Frame)} \qquad \langle R[t][s][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle t[s][m[\overline{R} \mapsto \bot], ms],\, C\rangle$$

$$\text{(Grant)}\ \langle (\texttt{grant}\,R\,\texttt{in}\,t)[s][m, ms],\, C\rangle \rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} \langle t[s][m[R \mapsto \top], ms],\, C\rangle$$

7.3.4 Formal correspondence

Proposition 11. *For any term* t *in the* λ_{sec}*-calculus,*

$$t[\varnothing](\rightarrow^{\mathrm{ucm}}_{\mathrm{sec}})^* v \quad \textit{if and only if} \quad \langle t,\, \varnothing,\, \varepsilon, \bullet,\, [\,]\rangle\, (\Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}})^*\, v.$$

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 $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus; second, how the $\lambda\widehat{\rho}^{\mathrm{cm}}_{\mathrm{sec}}$-calculus reflects the storeless cm machine; and third, how the $\lambda\widehat{\rho}^{\mathrm{ucm}}_{\mathrm{sec}}$-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 λ-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, $\lambda\widehat{\rho}_{\mathrm{ptr}}$.

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:

$$
\begin{array}{rcll}
\langle x,\ s,\ C,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle C,\ v,\ \sigma\rangle & \text{if } s(x)=\ell \text{ and } \sigma(\ell)=v\\
\langle \lambda x.t,\ s,\ C,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle C,\ (\lambda x.t,\ s),\ \sigma\rangle & \\
\langle t_0\, t_1,\ s,\ C,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle t_0,\ s,\ C[[\,]\ (t_1,\ s)],\ \sigma\rangle & \\
\langle x := t,\ s,\ C,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle t,\ s,\ C[\mathtt{upd}(\ell,[\,])],\ \sigma\rangle & \text{if } s(x)=\ell\\
\\
\langle [\,],\ v,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & (v,\ \sigma) & \\
\langle C[[\,]\ (t,\ s)],\ v,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle t,\ s,\ C[v\ [\,]],\ \sigma\rangle & \\
\langle C[(\lambda x.t,\ s)\ [\,]],\ v,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle t,\ (x,\ \ell)\cdot s,\ C,\ \sigma[\ell\mapsto v]\rangle & \text{if } \ell \text{ does not occur within } s, C, v, \sigma\\
\langle C[\mathtt{upd}(\ell,[\,])],\ v,\ \sigma\rangle & \Rightarrow_{\mathrm{ptr}} & \langle C,\ v',\ \sigma[\ell\mapsto v]\rangle & \text{if } \sigma(\ell)=v'
\end{array}
$$

Locations ℓ range over an unspecified set of locations. A store σ is a finite mapping from locations to value closures. Denotable values are locations. This machine evaluates a closed term t by starting in the configuration $\langle t,\ \varnothing,\ [\,],\ \bullet\rangle$. It halts with value v and store σ if it reaches a configuration $\langle [\,],\ v,\ \sigma\rangle$. It is in defunctionalized form.

Clinger's machine also has a garbage-collection rule [19, Figure 5 and Section 3], but for simplicity we ignore it here.

8.2 The language of $\lambda\widehat{\rho}_{\mathrm{ptr}}$

The abstract syntax of the language is as follows:

$$
\begin{array}{ll}
\text{(terms)} & t ::= x \mid \lambda x.t \mid t\,t \mid x := t\\
\text{(closures)} & c ::= t[s] \mid c\,c\\
\text{(values)} & v ::= (\lambda x.t)[s]\\
\text{(substitutions)} & s ::= \varnothing \mid (x,\ \ell)\cdot s\\
\text{(red. contexts)} & C ::= [\,] \mid C[[\,]\ c] \mid C[v\ [\,]] \mid C[\mathtt{upd}(\ell,[\,])]\\
\text{(store)} & \sigma ::= \bullet \mid \sigma[\ell\mapsto v]\\
\text{(store closures)} & \widetilde{c} ::= c[\sigma]\\
\text{(store values)} & \widetilde{v} ::= v[\sigma]\\
\text{(store contexts)} & \widetilde{C} ::= C[\sigma]
\end{array}
$$

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 $\lambda\widehat{\rho}$-calculus; and (Upd) updates the store.

$$
\begin{array}{lll}
(\mathrm{Var}) & \langle x_i[(x_1,\,\ell_1)\cdots(x_j,\,\ell_j)],\,C[\sigma]\rangle \rightarrow_{\mathrm{ptr}} \langle\sigma(\ell_i),\,C[\sigma]\rangle & \text{if } i \le j \\
(\mathrm{Beta}) & \langle((\lambda x.t)[s])\,v,\,C[\sigma]\rangle \rightarrow_{\mathrm{ptr}} \langle t[(x,\,\ell)\cdot s],\,C[\sigma[\ell\mapsto v]]\rangle & \text{if } \ell \text{ does not occur within } s, C, v, \sigma \\
(\mathrm{Prop}) & \langle(t_0\,t_1)[s],\,C[\sigma]\rangle \rightarrow_{\mathrm{ptr}} \langle(t_0[s])\,(t_1[s]),\,C[\sigma]\rangle &
\end{array}
$$

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* $\lambda\widehat{\rho}_{\mathrm{ptr}}$*-calculus,*

$$t[\varnothing][\bullet] \rightarrow^*_{\mathrm{ptr}} v[\sigma] \quad \textit{if and only if} \quad \langle t,\,\varnothing,\,[\,],\,\bullet\rangle \Rightarrow^*_{\mathrm{ptr}} (v,\,\sigma).$$

The $\lambda\widehat{\rho}_{\mathrm{ptr}}$-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, $\lambda\widehat{\rho}_{\mathrm{l}}$.

9.1 The language of $\lambda\widehat{\rho}_{\mathrm{l}}$

The abstract syntax of the language is as follows:

$$
\begin{array}{ll}
\text{(terms)} & t ::= i \mid \lambda t \mid t\,t \\
\text{(closures)} & c ::= t[s] \mid c\,\ell \mid \mathtt{upd}(\ell, c) \\
\text{(values)} & v ::= (\lambda t)[s] \\
\text{(substitutions)} & s ::= \varnothing \mid \ell\cdot s \\
\text{(reduction contexts)} & C ::= [\,] \mid C[[\,]\,\ell] \mid C[\mathtt{upd}(\ell,[\,])] \\
\text{(store)} & \sigma ::= \bullet \mid \sigma[\ell\mapsto v] \\
\text{(store closures)} & \widetilde{c} ::= c[\sigma] \\
\text{(store values)} & \widetilde{v} ::= v[\sigma] \\
\text{(store contexts)} & \widetilde{C} ::= C[\sigma]
\end{array}
$$

9.2 Notion of context-sensitive reduction

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

$$
\begin{array}{lll}
(\mathrm{Var}_1) & \langle i[\ell_1\cdots\ell_j],\,C[\sigma]\rangle \rightarrow_{\mathrm{l}} \langle v,\,C[\sigma]\rangle & \text{if } \sigma(\ell_i) = v \\
(\mathrm{Var}_2) & \langle i[\ell_1\cdots\ell_j],\,C[\sigma]\rangle \rightarrow_{\mathrm{l}} \langle \mathtt{upd}(\ell_i, c),\,C[\sigma]\rangle & \text{if } \sigma(\ell_i) = c \\
(\mathrm{Beta}) & \langle((\lambda t)[s])\,\ell,\,C[\sigma]\rangle \rightarrow_{\mathrm{l}} \langle t[\ell\cdot s],\,C[\sigma]\rangle & \\
(\mathrm{App}) & \langle(t_0\,t_1)[s],\,C[\sigma]\rangle \rightarrow_{\mathrm{l}} \langle(t_0[s])\,\ell,\,C[\sigma[\ell\mapsto t_1[s]]]\rangle & \text{where } \ell \text{ does not occur in } s, C, \sigma \\
(\mathrm{Upd}) & \langle\mathtt{upd}(\ell, v),\,C[\sigma]\rangle \rightarrow_{\mathrm{l}} \langle v,\,C[\sigma[\ell\mapsto v]]\rangle &
\end{array}
$$

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 (Var_1)—the result is this value, the current context, and the current store. In the latter case—handled by (Var_2)—a special closure $\mathtt{upd}(\ell, c)$ 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]

$$\begin{array}{rcll}
\langle i,\ s,\ C,\ \sigma\rangle & \Rightarrow_l & \langle C,\ (\lambda t',\ s'),\ \sigma\rangle & \text{if } s(i) = \ell \text{ and } \sigma(\ell) = (\lambda t',\ s') \\
\langle i,\ s,\ C,\ \sigma\rangle & \Rightarrow_l & \langle t',\ s',\ C[\mathtt{upd}(\ell, [\,])],\ \sigma\rangle & \text{if } s(i) = \ell \text{ and } \sigma(\ell) = (t',\ s') \\
\langle \lambda t,\ s,\ C,\ \sigma\rangle & \Rightarrow_l & \langle C,\ (\lambda t,\ s),\ \sigma\rangle & \\
\langle t_0\, t_1,\ s,\ C,\ \sigma\rangle & \Rightarrow_l & \langle t_0,\ s,\ C[[\,]\,\ell],\ \sigma[\ell \mapsto (t_1,\ s)]\rangle & \text{if } \ell \text{ does not occur in } s, C, \sigma \\
\langle [\,],\ v,\ \sigma\rangle & \Rightarrow_l & (v,\ \sigma) & \\
\langle C[[\,]\,\ell],\ (\lambda t,\ s),\ \sigma\rangle & \Rightarrow_l & \langle t,\ \ell \cdot s,\ C,\ \sigma\rangle & \\
\langle C[\mathtt{upd}(\ell, [\,])],\ v,\ \sigma\rangle & \Rightarrow_l & \langle C,\ v,\ \sigma[\ell \mapsto v]\rangle &
\end{array}$$

This machine evaluates a closed term t by starting in the configuration $\langle t,\ \varnothing,\ [\,],\ \bullet\rangle$. It halts with value v and store σ if it reaches a configuration $\langle [\,],\ v,\ \sigma\rangle$. It is in defunctionalized form, and coincides with the lazy abstract machine derived by Ager et al. out of a call-by-need interpreter for the λ-calculus [3].

9.4 Formal correspondence

Proposition 13. *For any term* t *in the* $\lambda\widehat{\rho}_l$*-calculus,*

$$t[\varnothing][\bullet] \rightarrow_l^* v[\sigma] \quad \textit{if and only if} \quad \langle t,\ \varnothing,\ [\,],\ \bullet\rangle \Rightarrow_l^* (v,\ \sigma).$$

The $\lambda\widehat{\rho}_l$-calculus therefore directly corresponds to call-by-need evaluation [89].

In $\lambda\widehat{\rho}_l$, 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 and Sestoft's approaches [64, 78]), 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 $\lambda\widehat{\rho}_l$ 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 λ-abstraction. Delaying allocation is useful in the presence of strict functions, which we do not consider here.

We can construct a local version of the $\lambda\widehat{\rho}_l$-calculus with either of the store propagated inside closures or of late allocation. From there, one can mechanically derive the corresponding abstract machines.

[3]When 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 [40]. The update marker is denoted $C[\mathtt{upd}(\ell, [\,])]$ here.

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 σ maps a closure into the corresponding λ-term and an environment-machine configuration into a configuration in the corresponding machine with actual substitutions [8, Section 2.5]):

$$\begin{tikzcd}
\lambda\text{-calculus} \arrow[r, leftrightarrow, "\text{syntactic}", "\text{correspondence}"'] & \text{machine with actual substitutions} \\
\lambda\widehat{\rho}\text{-calculus} \arrow[u, dotted, "\sigma"] \arrow[r, leftrightarrow, "\text{syntactic}", "\text{correspondence}"'] & \text{environment-based machine} \arrow[u, dotted, "\sigma"']
\end{tikzcd}$$

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

Defunctionalization: Defunctionalization has a practical origin: representing a higher-order function as a data type and a dispatch function [74]. It proves consistently useful, witness the next item and also the fact that except for the abstract machines for $\lambda\widehat{\rho}\mathcal{F}$ 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 is known to 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:

$$\begin{tikzcd}
& \text{context} \arrow[dl, "\text{decompose}'_{\text{aux}} \,\&\, \text{plug}"'] \arrow[dr, "\text{refocus}_{\text{aux}}"] & \\
\text{state-transition functions} & & \text{eval/apply abstract machine} \\
\text{one-step reduction function in CPS} \arrow[u, dashed, "\text{defunctionalization}"] & & \text{evaluation function in CPS} \arrow[u, dashed, "\text{defunctionalization}"']
\end{tikzcd}$$

Indeed, as initiated by Reynolds [29,74], 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 (the left part of the diagram just above) and the functional correspondence between abstract machines and evaluators (the right part of the diagram) 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 [74]. A typed version of it is available in Standard ML of New Jersey [53] and Griffin has identified its logical content [50]. It is endowed with a variety of specifications: a CPS transformation [32], a CPS interpreter [54,74], a denotational semantics [58], a computational monad [90], a big-step operational semantics [53], the CEK machine [44], calculi in the form of reduction semantics [43], and a number of implementation techniques [18,25,56]—not to mention its call-by-name variant in the archival version of Krivine's machine [61].
>
> 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.

Acknowledgments

We are grateful to Julia Lawall, Johan Munk, and Kristian Støvring for commenting a preliminary version of this article. The second author would also like to thank the Romantik restaurant in Bornholm for its quiet hospitality in July 2005. Finally, thanks are due to the anonymous reviewers and to our editor, Don Sannella.

This work is partially supported by the Danish Natural Science Research Council, Grant no. 21-03-0545 and by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`). We dedicate it with respect and affection to John Reynolds for his 70th birthday. May Mary and him live long and continue to prosper.

References

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

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

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

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

[5] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, January 1995. ACM Press.

[6] Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium (ICALP 2003)*, number 2719 in Lecture Notes in Computer Science, pages 871–885, Eindhoven, The Netherlands, July 2003. Springer.

[7] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

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

[9] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[10] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 16(3):269–280, 2006.

[11] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-06-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006. Revised version of BRICS RS-05-16.

[12] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60:274–297, 2006.

[13] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

[14] Eugene Charniak, Christopher Riesbeck, and Drew McDermott. *Artificial Intelligence Programming*. Lawrence Earlbaum Associates, 1980.

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

[16] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[17] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[18] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[19] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[20] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [92], pages 333–340.

[21] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. In Danvy [30]. To appear. Journal version of [20].

[22] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*, volume 1 of *Research Notes in Theoretical Computer Science*. Pitman, 1986.

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

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

[25] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[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)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

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

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

[29] Olivier Danvy. Defunctionalized interpreters for higher-order programming languages. In *Preliminary proceedings of the 21st Conference on Mathematical Foundations of Programming Semantics*, Birmingham, UK, May 2005. John Reynolds session.

[30] Olivier Danvy, editor. *Special Issue on the Krivine Abstract Machine*, Higher-Order and Symbolic Computation. Springer, 2007. In preparation.

[31] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [92], pages 151–160.

[32] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[33] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [13], pages 327–341.

[34] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number 4015 in Lecture Notes in Computer Science, pages 55–73, Dublin, Ireland, September 2005. Springer-Verlag. Extended version available as the technical report BRICS RS-06-4 (February 2006).

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

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

[37] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[38] Philippe de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structures in Computer Science*, 8:637–669, 1998.

[39] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. To appear in the Journal of Functional Programming. Available at `<http://www.cs.indiana.edu/~sabry/research.html>`, May 2006.

[40] Jon Fairbairn and Stuart Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, Oregon, September 1987. Springer-Verlag.

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

[42] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

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

[44] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[45] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[46] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [13], pages 52–62.

[47] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[48] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

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

[50] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[51] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

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

[53] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[54] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press.

[55] Carl Hewitt. Control structure as patterns of passing messages. In Patrick Henry Winston and Richard Henry Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2, pages 434–465. The MIT Press, 1979.

[56] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[57] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[58] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[59] 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.

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

[61] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In Danvy [30]. To appear. Available online at `<http://www.pps.jussieu.fr/~krivine/>`.

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

[63] Peter J. Landin. A generalization of jumps and labels. Research report, UNIVAC Systems Programming Research, 1965. Reprinted in Higher-Order and Symbolic Computation 11(2):125–143, 1998, with a foreword [88].

[64] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

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

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

[67] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, SIGPLAN Notices, Vol. 39, No. 9, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

[68] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[69] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

[70] Michel Parigot. λμ-calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, number 624 in Lecture Notes in Artificial Intelligence, pages 190–201, St. Petersburg, Russia, July 1992. Springer-Verlag.

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

[72] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2), 2005.

[73] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[74] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [75].

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

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

[77] Erik Sandewall. An early use of continuations and partial evaluation for compiling rules written in FOPC. *Higher-Order and Symbolic Computation*, 12(1):105–113, 1999.

[78] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[79] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[80] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 2007. Journal version of [79]. To appear.

[81] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[82] Brian C. Smith. Reflection and semantics in Lisp. In Ken Kennedy, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984. ACM Press.

[83] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[84] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000, with a foreword [69].

[85] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [91].

[86] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998, with a foreword [87].

[87] Gerald J. Sussman and Guy L. Steele Jr. The first report on Scheme revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, 1998.

[88] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[89] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[90] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[91] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.

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

[93] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, May 1988. A preliminary version was presented at the 1986 ACM Conference on Lisp and Functional Programming (LFP 1986).