Basic Research in Computer Science

A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines

Małgorzata Biernacka
Olivier Danvy

BRICS Report Series **RS-05-22**

ISSN 0909-0878 **July 2005**

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

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/05/22/`

A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines

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

July 2005

Abstract

We present a systematic construction of environment-based abstract machines from context-sensitive calculi of explicit substitutions, and we illustrate it with a series of calculi and machines: Krivine's machine with call/cc, the $\lambda\mu$-calculus, delimited continuations, i/o, stack inspection, proper tail-recursion, and lazy evaluation. Most of the machines already exist but have been obtained independently and are only indirectly related to the corresponding calculi. All of the calculi are new and they make it possible to directly reason about the execution of the corresponding machines. In connection with the functional correspondence between evaluation functions and abstract machines initiated by Reynolds, the present syntactic correspondence makes it possible to construct reduction-free normalization functions out of reduction-based ones, which was an open problem in the area of normalization by evaluation.

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

Contents

1 Introduction

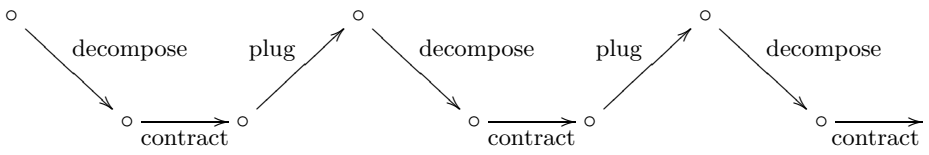
1.1 Calculi and machines

Sixty-five years ago, the λ-calculus was introduced [15]. Forty-five years ago, its expressive power was observed to be relevant for computing [60, 72]. Forty years ago, a first abstract machine for the λ-calculus was introduced [56]. Thirty years ago, calculi and abstract machines were formally connected [62]. Twenty years ago, a calculus format—reduction semantics—with an explicit representation of reduction contexts was introduced [35]. 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 [37].

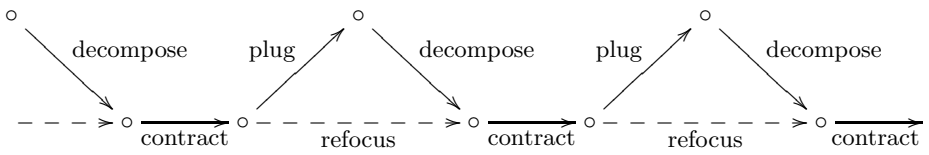
From calculus to machine by refocusing and transition compression: Recently, Danvy and Nielsen have pointed out that the reduction strategy for a calculus actually determines the structure of the corresponding machine [32]. They present an algorithm to construct an abstract machine out of a reduction 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. An evaluation function is defined using the reflexive and transitive closure of the one-step reduction function:



Danvy and Nielsen have observed that the intermediate terms, in the composition of plug and decompose, could be avoided by fusing the composition into a 'refocus' function:



The resulting 'refocused' evaluation function is defined as the reflexive and transitive closure of refocusing and contraction.

Danvy and Nielsen's algorithm yields a refocus function in 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, 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, 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 [25], and to context-based CPS transformations, improving them from quadratic time to operating in one pass [32].

1.2 Calculi of explicit substitution 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 calculi of explicit substitutions [1, 24, 45].

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 [38], how $\lambda\widehat{\rho}$ with normal order directly corresponds to the Krivine machine [20, 23], how $\lambda\widehat{\rho}$ with generalized reduction directly corresponds to the original version of Krivine's machine [53], and how $\lambda\widehat{\rho}$ with right-to-left applicative order directly corresponds to the ZINC abstract machine [58]. All of these machines are environment-based and use closures.

1.3 Calculi 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 [35, 38]. In these calculi, reduction rules are not oblivious to their reduction context; on the contrary, they are context sensitive in that the context takes part in some reduction steps, e.g., for call/cc. Reduction semantics are in wide use today, e.g., to study the security technique of stack inspection [16, 42, 63].

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 a variety of 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 [62] or via a compilation / decompilation scheme in the course of execution [45]).

1.4 Overview

We successively consider call by name: Krivine's machine with call/cc (Section 3) and the $\lambda\mu$-calculus (Section 4); call by value: static and dynamic delimited continuations (Section 5), i/o (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).

2 Preliminaries

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

Since Landin [56], 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 motivated the study of calculi of explicit substitutions [1, 23, 66] to mediate between the traditional abstract specifications of the λ-calculus and its traditional concrete implementations [45].

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 λ-terms. 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 algorithm. 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:

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

(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 $t[\bullet]$ where t does not contain free variables.

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

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

We denote by $s(i)$ 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:

$$\text{(L-Comp)} \qquad \frac{c_0 \to_{\widehat{\rho}} c_0'}{c_0\, c_1 \to_{\widehat{\rho}} c_0'\, c_1}$$

$$\text{(R-Comp)} \qquad \frac{c_1 \to_{\widehat{\rho}} c_1'}{c_0\, c_1 \to_{\widehat{\rho}} c_0\, c_1'}$$

$$\text{(Sub)} \qquad \frac{c_i \to_{\widehat{\rho}} c_i'}{t[c_1 \cdots c_i \cdots c_j] \to_{\widehat{\rho}} t[c_1 \cdots c_i' \cdots c_j]} \quad \text{for } i \leq j$$

Specific, deterministic reduction strategies can be obtained by restricting the compatibility rules. In the following sections, we consider two such strategies: the normal-order strategy obtained by discarding the (R-Comp) and (Sub) rules, and the left-to-right applicative-order strategy obtained in the usual way by restricting the (Beta) and (R-Comp) rules, and discarding the (Sub) rule.

All of the calculi presented in this article are syntactic extensions of the $\lambda\widehat{\rho}$-calculus.

2.2 Notion of 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 (potential) 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.

A one-step reduction relation for a given notion of context-sensitive reduction is defined as follows, assuming this notion to be specified by reduction rules of the form $\langle r,\, C\rangle \to \langle c,\, C'\rangle$, where $\langle r,\, C\rangle$ denotes the decomposition of a program into a potential redex r and its context C. A 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'.

[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 [62].

Any standard notion of reduction can be trivially transformed into context-sensitive form. For example, here is the corresponding notion of reduction for the $\lambda\widehat{\rho}$-calculus:

$$\begin{array}{lrcll} \text{(Var)} & \langle i[c_1 \cdots c_j],\, C\rangle & \to_{\widehat{\rho}} & \langle c_i,\, C\rangle & \text{if } i \leq j \\ \text{(Beta)} & \langle((\lambda t)[s])\; c,\, C\rangle & \to_{\widehat{\rho}} & \langle t[c \cdot s],\, C\rangle & \\ \text{(Prop)} & \langle(t_0\; t_1)[s],\, C\rangle & \to_{\widehat{\rho}} & \langle(t_0[s])\; (t_1[s]),\, C\rangle & \end{array}$$

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 5 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.

For each of the calculi considered in the remainder of this article, we define a suitable notion of reduction, denoted $\to_X$, 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 $\to_X^*$) using the reflexive, transitive closure of one-step reduction, i.e., we say that c evaluates to c' if $c \to_X^* c'$ and c' is a value closure. We define the convertibility relation between closures as the smallest equivalence relation containing $\to_X^*$. 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 $\lambda\widehat{\rho}\mathcal{K}$-calculus

The Krivine machine is probably the most well-known abstract machine implementing the normal-order reduction strategy in the λ-calculus [28]. In our previous work [8], we have pointed out that Krivine's original machine [53] 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, the archival version of Krivine's machine [54] also handles call/cc (noted $\mathcal{K}$ below).

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

Here, we present the calculus corresponding to the archival version of Krivine's machine. This machine also uses closures and an environment. In addition to generalized β-reduction, it also features $\mathcal{K}$. 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. The

calculus is tied to a particular reduction strategy. Here, like Krivine, we consider the normal-order reduction strategy and therefore call by name [62].

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

The abstract syntax of the language is as follows:

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

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 $\mathcal{K}$.

3.2 Notion of context-sensitive reduction

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

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

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

3.3 Krivine's machine

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

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

This machine coincides with the extension of Krivine's machine with $\mathcal{K}$—an extension which was designed as such [54].

3.4 Formal correspondence

Proposition 1. *For any term t in the $\lambda\widehat{\rho}\mathcal{K}$-calculus,*

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

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

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

In this section we present a calculus of closures that extends Parigot's $\lambda\mu$-calculus [61] 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 [33] 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. For lack of space, however, we omit all the typing considerations, focusing on the syntactic correspondence between the calculus and the machine.

4.1 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 [33]. Alternatively, we could use two separate sets of indices, which would then lead us to two environments in the resulting machine (one for each kind of variable).

The abstract syntax of the language is specified as follows:

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

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

4.2 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 continuation variable is applied:

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

4.3 An eval/apply abstract machine

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

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

This machine coincides with de Groote's final abstract machine [33, p. 24], except that instead of traversing the environment as a list, we directly fetch the right substitutee for a given index i.

4.4 Formal correspondence

Proposition 2. *For any term t in the $\lambda\widehat{\rho}\mu$-calculus,*

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

The $\lambda\widehat{\rho}\mu$-calculus therefore directly corresponds to de Groote's abstract machine for the $\lambda\mu$-calculus.

5 Delimited continuations

Continuations have been discovered multiple times [64], but they acquired their name for describing jumps [73], using what is now known as continuation-passing style (CPS) [71]. A full-fledged control operator, J [55, 75], however, existed before CPS, providing first-class continuations in direct style. Continuations therefore existed before CPS, and so one could say that it was really CPS that was discovered multiple times.

Conversely, delimited continuations, in the form of the traditional success and failure continuations [67], have been regularly used in artificial-intelligence programming [14, 48, 74] for generators and backtracking. They also occur in the study of reflective towers [70], where the notions of meta-continuation [78] and of "jumpy" vs. "pushy" continuations [30] 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 [36, 39]. Only subsequently were control delimiters connected to success and failure continuations [29].

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 [36], and CPS, as discovered by Danvy and Filinski [29].

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 [31, 65]); 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 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 5.1 and dynamic delimited continuations in Section 5.2. In both cases, we consider the left-to-right applicative-order reduction strategy and therefore left-to-right call by value.

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

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

5.1.1 The language of $\lambda\widehat{\rho}\mathcal{S}$

The abstract syntax of the language is as follows:

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

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 [29]. Elsewhere [7], we have defunctionalized this interpreter into an environment-based machine, which we present next.

5.1.2 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 [59]) 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:

$$
\begin{array}{rcl}
\langle i,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t',\, s',\, C_1,\, C_2\rangle \quad \text{if } s(i) = (t',\, s') \\
\langle \lambda t,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\, (\lambda t,\, s),\, C_2\rangle \\
\langle t_0\, t_1,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t_0,\, s,\, C_1[[\,]\,(t_1,\, s)],\, C_2\rangle \\
\langle \mathcal{S}\, t,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, C_1[\mathcal{S}[\,]],\, C_2\rangle \\
\langle \langle t\rangle,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, [\,],\, C_1 \cdot C_2\rangle \\
\\
\langle [\,],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_2,\, v\rangle \\
\langle C_1[[\,]\,(t,\, s)],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, C_1[v\,[\,]],\, C_2\rangle \\
\langle C_1[(\lambda t,\, s)\,[\,]],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, v \cdot s,\, C_1,\, C_2\rangle \\
\langle C_1[\ulcorner C_1' \urcorner\,[\,]],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1',\, v,\, C_1 \cdot C_2\rangle \\
\langle C_1[\mathcal{S}[\,]],\, (\lambda t,\, s),\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, \ulcorner C_1 \urcorner \cdot s,\, [\,],\, C_2\rangle \\
\langle C_1[\mathcal{S}[\,]],\, \ulcorner C_1' \urcorner,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1',\, \ulcorner C_1 \urcorner,\, [\,] \cdot C_2\rangle \\
\\
\langle \bullet,\, v\rangle & \Rightarrow_{\mathcal{S}} & v \\
\langle C_1 \cdot C_2,\, v\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\, v,\, C_2\rangle
\end{array}
$$

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

5.1.3 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:

$$
\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle c_i,\ C_1,\ C_2\rangle \quad \text{if } i \le j \\
\text{(Beta)} & \langle((\lambda t)[s])\ v,\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle t[v \cdot s],\ C_1,\ C_2\rangle \\
(\text{Beta}_C) & \langle \ulcorner C_1' \urcorner v\ ,\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle\langle C_1'[v]\rangle,\ C_1,\ C_2\rangle \\
\text{(Prop)} & \langle(t_0\ t_1)[s],\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle(t_0[s])\ (t_1[s]),\ C_1,\ C_2\rangle \\
(\text{Prop}_{\mathcal{S}}) & \langle(\mathcal{S}\ t)[s],\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle\mathcal{S}\ (t[s]),\ C_1,\ C_2\rangle \\
(\text{Prop}_{\langle\cdot\rangle}) & \langle\langle t\rangle[s],\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle\langle t[s]\rangle,\ C_1,\ C_2\rangle \\
(\mathcal{S}_\lambda) & \langle\mathcal{S}\ ((\lambda t)[s]),\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle t[\ulcorner C_1 \urcorner \cdot s],\ [\,],\ C_2\rangle \\
(\mathcal{S}_C) & \langle\mathcal{S}\ulcorner C_1'\urcorner,\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle\ulcorner C_1'\urcorner\ulcorner C_1\urcorner,\ [\,],\ C_2\rangle \\
\text{(Reset)} & \langle\langle v\rangle,\ C_1,\ C_2\rangle & \to_{\mathcal{S}} & \langle v,\ C_1,\ C_2\rangle
\end{array}
$$

Since none of the contractions depends on the meta-context, it is evident that the notion of reduction $\to_{\mathcal{S}}$ is compatible with meta-contexts, but it is 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.

5.1.4 Formal correspondence

Proposition 3. *For any term t in the $\lambda\widehat{\rho}\mathcal{S}$-calculus,*

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

The $\lambda\widehat{\rho}\mathcal{S}$-calculus therefore directly corresponds to the abstract machine for shift and reset.

5.1.5 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 [29] which Filinski has shown to be able to represent layered monads [41]. Recently, Kameyama has proposed an axiomatization of the CPS hierarchy [50]. Elsewhere [7], we have studied its defunctionalized counterpart and the corresponding hierarchy of calculi.

5.2 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.

5.2.1 The language of $\lambda\widehat{\rho}\mathcal{F}$

The abstract syntax of the language is as follows:

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

5.2.2 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]

$$
\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\, C\rangle & \to_{\mathcal{F}} & \langle c_i,\, C\rangle \text{ if } i \le j \\
\text{(Beta)} & \langle((\lambda t)[s])\,v,\, C\rangle & \to_{\mathcal{F}} & \langle t[v \cdot s],\, C\rangle \\
(\text{Beta}_C) & \langle \ulcorner C' \urcorner v\,,\, C\rangle & \to_{\mathcal{F}} & \langle C'[v],\, C\rangle \\
\text{(Prop)} & \langle(t_0\,t_1)[s],\, C\rangle & \to_{\mathcal{F}} & \langle(t_0[s])\,(t_1[s]),\, C\rangle \\
(\text{Prop}_{\mathcal{F}}) & \langle(\mathcal{F}\,t)[s],\, C\rangle & \to_{\mathcal{F}} & \langle\mathcal{F}\,(t[s]),\, C\rangle \\
(\text{Prop}_{\#}) & \langle(\#t)[s],\, C\rangle & \to_{\mathcal{F}} & \langle\#\,(t[s]),\, C\rangle \\
(\mathcal{F}_\lambda) & \langle\mathcal{F}\,((\lambda t)[s]),\, C[\#\,C']\rangle & \to_{\mathcal{F}} & \langle t[\ulcorner C' \urcorner \cdot s],\, C\rangle \\
 & & & \quad \text{if } C' \text{ contains no mark} \\
(\mathcal{F}_C) & \langle\mathcal{F}\ulcorner C'' \urcorner,\, C[\#\,C']\rangle & \to_{\mathcal{F}} & \langle\ulcorner C'' \urcorner\ulcorner C \urcorner,\, C\rangle \\
 & & & \quad \text{if } C' \text{ contains no mark} \\
\text{(Prompt)} & \langle\#\,v,\, C\rangle & \to_{\mathcal{F}} & \langle v,\, C\rangle
\end{array}
$$

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):

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

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

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

5.2.3 The eval/apply abstract machine

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

$$\begin{array}{rcl}
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t',\, s',\, C\rangle \quad \text{if } s(i) = (t',\, s') \\
\langle \lambda t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, (\lambda t,\, s)\rangle \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle \\
\langle \mathcal{F}\, t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\mathcal{F}[\,]]\rangle \\
\langle \#t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\#[\,]]\rangle \\
\\
\langle C[[\,]\,(t,\, s)],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[v\,[\,]]\rangle \\
\langle C[(\lambda t,\, s)\,[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, v \cdot s,\, C\rangle \\
\langle C[\ulcorner C' \urcorner\,[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle C' \circ C,\, v\rangle \\
\langle C[\#C'[\mathcal{F}[\,]]],\, (\lambda t,\, s)\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, \ulcorner C' \urcorner \cdot s,\, C\rangle \\
 & & \quad \text{where } C' \text{ contains no mark} \\
\langle C[\#C'[\mathcal{F}[\,]]],\, \ulcorner C'' \urcorner\rangle & \Rightarrow_{\mathcal{F}} & \langle C'' \circ C,\, \ulcorner C' \urcorner\rangle \\
 & & \quad \text{where } C' \text{ contains no mark} \\
\langle C[\#[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, v\rangle
\end{array}$$

This environment-based machine coincides with Felleisen's extension of the CEK machine—an extension which was designed as such [36, Section 3].

5.2.4 Formal correspondence

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

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

This proposition parallels Felleisen's second correspondence theorem [36, p. 186]. The $\lambda\widehat{\rho}\mathcal{F}$-calculus therefore directly corresponds to Felleisen's extension of the CEK machine.

5.2.5 A hierarchy of control delimiters

As described by Sitaram and Felleisen [69], 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 [69, Section 7]. 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. [44] and more recently of Dybvig et al. [34], this area is little explored.

5.3 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 [36]. Elsewhere, we present the abstract machine [7] and the evaluator [29] 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 5.1.3); a context containing marks is less easy to deal with. Otherwise, the difference between static and dynamic delimited continuations is tiny (see Section 5.2.2), 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 a 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." [40, 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 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, 34, 52, 68]. 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 composed by dynamically concatenating their representations [40] 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" [39] 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.

6 A calculus of closures with input/output ($\lambda\widehat{\rho}_{\mathrm{i/o}}$)

In this and the two subsequent sections we show calculi enriched with features whose implementations via abstract machines usually introduce a state, i.e., a global component of a machine configuration that can be accessed and updated at any time by a currently evaluated subclosure. We show the corresponding calculi of closures extended with a suitable syntactic entity to model state.

First, we present a simple calculus with primitives for modelling finite input and output, where closures are equipped with two additional components: an input channel (a finite list I), and an output channel (a finite list O). Since we provide explicit syntactic characterization of input and output, it is possible to give the calculus a standard reduction semantics instead of a labeled transition system with read and write actions expressed as annotations on transitions. Such a specification allows us to apply the refocusing technique and mechanically derive an abstract machine for this calculus.

6.1 The language of $\lambda\widehat{\rho}_{\mathrm{i/o}}$

The abstract syntax of the language is specified as follows:

(terms)	t	::=	$\ell \mid i \mid \lambda t \mid t\,t \mid \mathtt{in}\,t \mid \mathtt{out}\,t \mid t;t$
(closures)	c	::=	$t[s] \mid c\,c \mid \mathtt{in}\,c \mid \mathtt{out}\,c \mid c;c$
(values)	v	::=	$(\lambda t)[s]$
(substitutions)	s	::=	$\bullet \mid c \cdot s$
(reduction contexts)	C	::=	$[\,] \mid C[[\,]\,c] \mid C[v_0\,[\,]] \mid$ $C[[\,];c] \mid C[\mathtt{in}\,[\,]] \mid C[\mathtt{out}\,[\,]]$
(input)	I	::=	$\bullet \mid \ell :: I$
(output)	O	::=	$\bullet \mid \ell :: O$
(i/o closures)	$\widetilde{c}$	::=	$c[I, O]$
(i/o values)	$\widetilde{v}$	::=	$v[I, O]$
(i/o contexts)	$\widetilde{C}$	::=	$C[I, O]$

The set of terms is extended with three constructs: `in` for reading in a literal, `out` for writing out a literal, and a binary operator $\cdot;\cdot$ to sequentialize computation.

The new i/o closures are built on top of the usual closures (terms with explicit substitutions, and their compositions) equipped with two lists of literals: an input channel denoted I, and an output channel denoted O.

6.2 Notion of context-sensitive reduction

Input/output channels are global for the entire computation (contrary to substitutions, propagated locally to each subterm), which is obtained by ensuring that at each step only one subclosure can access and modify these channels (hence the restricted forms of closures). The reduction rules are performed only on i/o closures, making the calculus deterministic.

$$
\begin{array}{lrcl}
\text{(Beta)} & \langle((\lambda t)[s])\ v,\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle t[v \cdot s],\ \widetilde{C}\rangle \\
\text{(Access)} & \langle i[c_1 \cdots c_j],\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle s(i),\ \widetilde{C}\rangle \\
\text{(Prop)} & \langle(t_0\ t_1)[s],\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle t_0[s]\ t_1[s],\ \widetilde{C}\rangle \\
(\text{Prop}_{seq}) & \langle(t_0; t_1)[s],\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle t_0[s]; t_1[s],\ \widetilde{C}\rangle \\
(\text{Prop}_{in}) & \langle(\mathtt{in}\ t)[s],\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle(\mathtt{in}\ t[s]),\ \widetilde{C}\rangle \\
(\text{Prop}_{out}) & \langle(\mathtt{out}\ t)[s],\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle(\mathtt{out}\ t[s]),\ \widetilde{C}\rangle \\
\text{(Seq)} & \langle v; c,\ \widetilde{C}\rangle & \to_{\mathrm{i/o}} & \langle c,\ \widetilde{C}\rangle \\
\text{(In)} & \langle(\mathtt{in}\ (\lambda t)[s]),\ C[\ell :: I,\ O]\rangle & \to_{\mathrm{i/o}} & \langle t[\ell[\bullet] \cdot s],\ C[I,\ O]\rangle \\
\text{(Out)} & \langle(\mathtt{out}\ \ell[s]),\ C[I,\ O]\rangle & \to_{\mathrm{i/o}} & \langle \ell[s],\ C[I,\ \ell :: O]\rangle
\end{array}
$$

The need for context-sensitive reductions arises in the last two reductions that manipulate global input and output channels, respectively. The rule (In) reads a literal from an input channel and stores it in the substitution, and the (Out) rule prints a literal to the output channel. Note that the reduction rules are compatible with contexts C, and therefore they facilitate local reasoning about programs in the following sense:

Proposition 5. *For all closures c_1, c_2 such that $c_1[I,\ O] =_{\mathrm{i/o}} c_2[I,\ O]$, and for every context C,*

$$(C[c_1])[I,\ O] =_{\mathrm{i/o}} (C[c_2])[I,\ O].$$

It is possible to reformulate the calculus in the usual context-insensitive form, by propagating the input/output channels down to each closure—similarly to the way a substitution is propagated. The final abstract machine obtained by refocusing is the same as the one presented below.

6.3 An eval/apply abstract machine

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

$$
\begin{array}{rcl}
\langle \ell,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\, (\ell,\, s),\, I,\, O\rangle \\
\langle \lambda t,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\, (\lambda t,\, s),\, I,\, O\rangle \\
\langle i,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, s',\, C,\, I,\, O\rangle \quad \text{if } s(i) = (t,\, s') \\
\langle t_0\, t_1,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)],\, I,\, O\rangle \\
\langle t_0; t_1,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t_0,\, s,\, C[[\,]; (t_1,\, s)],\, I,\, O\rangle \\
\langle \mathtt{in}\, t,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, s,\, C[\mathtt{in}\,[\,]],\, I,\, O\rangle \\
\langle \mathtt{out}\, t,\, s,\, C,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, s,\, C[\mathtt{out}\,[\,]],\, I,\, O\rangle \\
\\
\langle [\,],\, v,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & (v,\, I,\, O) \\
\langle C[[\,]\,(t,\, s)],\, v,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, s,\, C[v\,[\,]],\, I,\, O\rangle \\
\langle C[(\lambda t,\, s)\,[\,]],\, v,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, v \cdot s,\, C,\, I,\, O\rangle \\
\langle C[[\,]; (t,\, s)],\, v,\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\, s,\, C,\, I,\, O\rangle \\
\langle C[\mathtt{in}\,[\,]],\, (\lambda t,\, s),\, \ell :: I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\, t,\, \ell[\bullet] \cdot s,\, I,\, O\rangle \\
\langle C[\mathtt{out}\,[\,]],\, (\ell,\, s),\, I,\, O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\, (\ell,\, s),\, I,\, \ell :: O\rangle
\end{array}
$$

6.4 Formal correspondence

Proposition 6. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{i/o}}$-calculus,*

$$t[\bullet][I,\, \bullet] \rightarrow^*_{i/o} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,],\, I,\, \bullet\rangle \Rightarrow^*_{i/o} v.$$

7 Stack inspection

This section addresses Fournet and Gordon's λ_{sec}-calculus, which formalizes security enforcement by stack inspection [42]. 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 that the tail-optimization of the cm machine is reflected by a non-standard plug function. 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 $\lambda\widehat{\rho}_{\text{sec}}$

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

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:

$$
\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C\rangle & \to_{\text{sec}} & \langle c_i,\ C\rangle \quad \text{if } i \le j \\
\text{(Beta)} & \langle((\lambda t)[s])\ v,\ C\rangle & \to_{\text{sec}} & \langle t[v \cdot s],\ C\rangle \\
\text{(Prop)} & \langle(t_0\ t_1)[s],\ C\rangle & \to_{\text{sec}} & \langle(t_0[s])\ (t_1[s]),\ C\rangle \\
(\text{Prop}_G) & \langle(\texttt{grant}\ R\ \texttt{in}\ t)[s],\ C\rangle & \to_{\text{sec}} & \langle\texttt{grant}\ R\ \texttt{in}\ t[s],\ C\rangle \\
(\text{Prop}_F) & \langle(R[t])[s],\ C\rangle & \to_{\text{sec}} & \langle R[t[s]],\ C\rangle \\
(\text{Prop}_T) & \langle(\texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1)[s],\ C\rangle & \to_{\text{sec}} & \langle\texttt{test}\ R\ \texttt{then}\ t_0[s]\ \texttt{else}\ t_1[s],\ C\rangle \\
\text{(Frame)} & \langle R[v],\ C\rangle & \to_{\text{sec}} & \langle v,\ C\rangle \\
\text{(Grant)} & \langle\texttt{grant}\ R\ \texttt{in}\ v,\ C\rangle & \to_{\text{sec}} & \langle v,\ C\rangle \\
(\text{Test}_1) & \langle\texttt{test}\ R\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ C\rangle & \to_{\text{sec}} & \langle c_1,\ C\rangle \quad \text{if } \mathcal{OK}_{\text{sec}}(R, C) \\
(\text{Test}_2) & \langle\texttt{test}\ R\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ C\rangle & \to_{\text{sec}} & \langle c_2,\ C\rangle \quad \text{otherwise} \\
\text{(Fail)} & \langle\texttt{fail}[s],\ C\rangle & \to_{\text{sec}} & \langle\texttt{fail},\ [\,]\rangle
\end{array}
$$

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:

$$
\begin{array}{rcl}
\langle i,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle C,\ s(i)\rangle \\
\langle\lambda t,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle C,\ (\lambda t,\ s)\rangle \\
\langle t_0\ t_1,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle t_0,\ s,\ C[[\,]\ (t_1,\ s)]\rangle \\
\langle\texttt{grant}\ R\ \texttt{in}\ t,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle t,\ s,\ C[\texttt{grant}\ R\ \texttt{in}\ [\,]]\rangle \\
\langle\texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle t_0,\ s,\ C\rangle \quad \text{if } \mathcal{OK}_{\text{sec}}(R, C) \\
\langle\texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle t_1,\ s,\ C\rangle \quad \text{otherwise} \\
\langle\texttt{fail},\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \texttt{fail} \\
\langle R[t],\ s,\ C\rangle & \Rightarrow_{\text{sec}} & \langle t,\ s,\ C[R[[\,]]]\rangle \\
\\
\langle[\,],\ v\rangle & \Rightarrow_{\text{sec}} & v \\
\langle C[[\,]\ (t,\ s)],\ v\rangle & \Rightarrow_{\text{sec}} & \langle t,\ s,\ C[v\ [\,]]\rangle \\
\langle C[(\lambda t,\ s)\ [\,]],\ v\rangle & \Rightarrow_{\text{sec}} & \langle t,\ v \cdot s,\ C\rangle \\
\langle C[\texttt{grant}\ R\ \texttt{in}\ [\,]],\ v\rangle & \Rightarrow_{\text{sec}} & \langle C,\ v\rangle \\
\langle C[R[[\,]]],\ v\rangle & \Rightarrow_{\text{sec}} & \langle C,\ v\rangle
\end{array}
$$

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

7.1.4 Formal correspondence

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

$$t[\bullet] \to^*_{\mathrm{sec}} v \quad \textit{if and only if} \quad \langle t, \bullet, [\,]\rangle \Rightarrow^*_{\mathrm{sec}} v.$$

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 (noted m below):

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

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

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

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

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

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

the transitions of the storeless cm machine read as follows:

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

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

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

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

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

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

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 grammar of terms, closures and substitutions is the same, but the reduction contexts (which correspond to the stack frames in the machine) contain permission tables. Consequently, the functions plug and decompose are defined in a non-standard way:

$$\begin{array}{lcl}
\mathsf{plug}\,(c, m[\,]) & = & \mathsf{build}\,(m, c) \\
\mathsf{plug}\,(c_0, C[[\,](c_1, m)]) & = & \mathsf{plug}\,(\mathsf{build}\,(m, c_0\, c_1), C) \\
\mathsf{plug}\,(c, C[(v, m)[\,]]) & = & \mathsf{plug}\,(\mathsf{build}\,(m, v\, c), C)
\end{array}$$

where the auxiliary function build conservatively constructs a closure based on the

permission table of the reduction context:

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

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

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

The notion of reduction includes most rules of the $\lambda\widehat{\rho}_{\mathrm{sec}}$-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 $\mathcal{OK}_{\mathrm{sec}}$ to $\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}$) 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 $\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, ms)$,

$$\frac{}{\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(\emptyset, m, ms)} \qquad \frac{R \cap m^{-1}(\bot) = \emptyset}{\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, \bullet)}$$

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

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

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

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

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

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

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

7.3.2 The language of $\lambda\widehat{\rho}_{\mathrm{sec}}^{\mathrm{ucm}}$

$$
\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\, t \mid \mathtt{grant}\, R\, \mathtt{in}\, t \mid \mathtt{test}\, R\, \mathtt{then}\, t\, \mathtt{else}\, t \mid \\
 & & & \mathtt{test}\, R\, \mathtt{then}\, t\, \mathtt{else}\, t \mid R[t] \mid \mathtt{fail} \\
\text{(closures)} & c & ::= & t[s] \\
\text{(values)} & v & ::= & (\lambda t)[s] \mid \mathtt{fail} \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \\
\text{(annotated closures)} & \widetilde{c} & ::= & c[m, ms] \mid c\,\widetilde{c} \mid \widetilde{c}\,c \mid \mathtt{fail} \\
\text{(annotated values)} & \widetilde{v} & ::= & v[m, ms] \mid \mathtt{fail}
\end{array}
$$

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.

$$
\begin{array}{lrcl}
\text{(Prop)} & \langle (t_0\ t_1)[s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle (t_0[s][\varepsilon, m \cdot ms])\ (t_1[s]),\ C\rangle \\
\text{(Var)} & \langle i[c_1 \cdots c_j][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle c_i[m, ms],\ C\rangle \ \text{ if } i \le j \\
\text{(Test}_1\text{)} & \langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1[s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle t_0[s][m, ms],\ C\rangle \\
 & & & \text{if } \mathcal{OK}_{\text{sec}}^{\text{ucm}}(R, m, ms) \\
\text{(Test}_2\text{)} & \langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1[s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle t_1[s][m, ms],\ C\rangle \\
 & & & \text{otherwise} \\
\text{(Fail)} & \langle \texttt{fail}[s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle \texttt{fail},\ [\,]\rangle \\
\text{(Switch)} & \langle (v[m, ms])\ c,\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle v\ (c[\varepsilon, ms]),\ C\rangle \\
\text{(Beta)} & \langle ((\lambda t)[s])\ (v[m, m' \cdot ms]),\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle t[v \cdot s][m', ms],\ C\rangle \\
\text{(Frame)} & \langle R[t][s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle t[s][m[\overline{R} \mapsto \bot], ms],\ C\rangle \\
\text{(Grant)} & \langle \texttt{grant}\ R\ \texttt{in}\ t[s][m, ms],\ C\rangle & \to_{\text{sec}}^{\text{ucm}} & \langle t[s][m[R \mapsto \top], ms],\ C\rangle
\end{array}
$$

A new reduction rule (Switch) is now necessary to go from one evaluated subclosure to a subclosure to evaluate. The (Beta) rule doubles up with discarding the permission table of the actual parameter. The (Frame) and (Grant) rules embody the state counterpart of Clements and Felleisen's design to enable proper tail recursion.

7.3.4 Formal correspondence

Proposition 10. *For any term t in the λ_{sec}-calculus,*

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

7.4 Conclusion

We have presented three corresponding calculi of closures and machines for stack inspection, showing first how the storeless fg machine reflects the $\lambda\widehat{\rho}_{\text{sec}}$-calculus, second, how the $\lambda\widehat{\rho}_{\text{sec}}^{\text{cm}}$-calculus reflects the storeless cm machine, and third, how the $\lambda\widehat{\rho}_{\text{sec}}^{\text{ucm}}$-calculus reflects the unzipped storeless cm machine. In doing so, we have provided a calculus account of machine design and optimization.

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 is an eval/apply machine with an environment and a store:

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

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.

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:

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

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.

$$
\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C[\sigma]\rangle & \to_{\mathrm{ptr}} & \langle \sigma(c_i),\ C[\sigma]\rangle \quad \text{if } i \le j \\
\text{(Beta)} & \langle ((\lambda x.t)[s])\ v,\ C[\sigma]\rangle & \to_{\mathrm{ptr}} & \langle t[(x,\ \ell)\cdot s],\ C[\sigma[\ell \mapsto v]]\rangle \\
 & & & \text{if } \ell \text{ does not occur within} s, v, C, \sigma \\
\text{(Prop)} & \langle (t_0\ t_1)[s],\ C[\sigma]\rangle & \to_{\mathrm{ptr}} & \langle (t_0[s])\ (t_1[s]),\ C[\sigma]\rangle \\
\text{(Upd)} & \langle \mathtt{upd}(\ell, v),\ C[\sigma]\rangle & \to_{\mathrm{ptr}} & \langle \sigma(\ell),\ C[\sigma[\ell \mapsto v]]\rangle
\end{array}
$$

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

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

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. We do not include them here for lack of space.

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:

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

9.2 Notion of context-sensitive reduction

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

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

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]

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

This lazy abstract machine coincides with the one derived by Ager et al. out of a call-by-need interpreter for the λ-calculus [3], thereby connecting the present syntactic correspondence between calculi and abstract machines with the functional correspondence between evaluators and abstract machines [2, 4, 7].

[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 (denoted $C[\mathtt{upd}(\ell, [\,])]$ here).

9.4 Formal correspondence

Proposition 12. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{l}}$-calculus,*

$$t[\bullet][\bullet] \to_{\mathrm{l}}^{*} v[\sigma] \quad \textit{if and only if} \quad \langle t, \bullet, [\,], \bullet\rangle \Rightarrow_{\mathrm{l}}^{*} (v, \sigma).$$

The $\lambda\widehat{\rho}_{\mathrm{l}}$-calculus therefore directly corresponds to call-by-need evaluation [76].

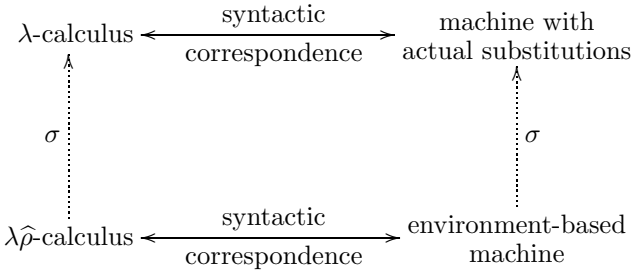
In $\lambda\widehat{\rho}_{\mathrm{l}}$, sharing is made possible through a global heap where actual parameters are stored. On the other hand, a number of other calculi modeling call by need extend the set of terms with a *local* let-like construct, either by statically translating the source language into an intermediate language with explicit indications of sharing (as in Launchbury's approach [57]), 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 operators to the top level.

Another specificity is that allocation occurs early in $\lambda\widehat{\rho}_{\mathrm{l}}$, 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 our calculus with either of the store propagated inside closures or of late allocation, and from there, mechanically derive the corresponding abstract machine.

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):



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, $\lambda\rho$ directly corresponds to the notions of evaluation (i.e., weak-head normalization) accounted for by environment-based machines, even in the presence of computational effects (state and control).

Refocusing: Despite its pragmatic origin—fusing a plug function and a decomposition function in a reduction-based evaluation function to improve its efficiency [32], and in combination with compressing intermediate transitions and unfolding closures, refocusing proves consistently useful to construct reduction-free evaluation functions in the form of abstract machines, even in the presence of computational effects.

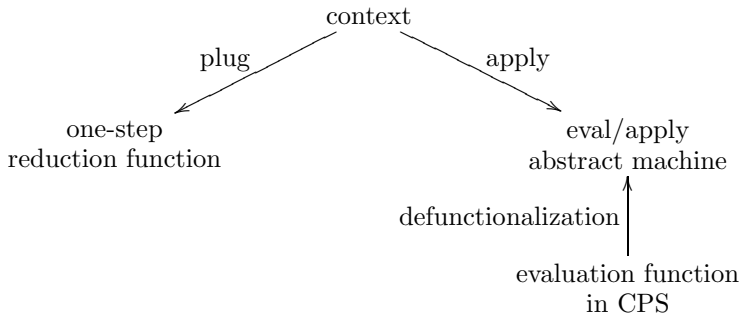
Defunctionalization: Despite its practical origin—representing a higher-order function with first-order means [65], defunctionalization proves consistently useful, witness the next item and also the fact that except for the abstract machines for $\lambda\widehat{\rho}\mathcal{F}$ and the cm machine, *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 can be tricky in practice). But more significantly [26], reduction contexts and evaluation contexts *coincide*, which means that they mediate between one-step reduction and evaluation, particularly since, as initiated by Reynolds [65], defunctionalizing a continuation-passing evaluator yields an abstract machine [2, 3, 4, 7], and since

as already pointed out above, a vast number of abstract machines are in defunctionalized form [9, 11, 27]:



Together, the syntactic correspondence and the functional correspondence connect apparently distinct approaches to the same computational situations. We already illustrated this connection in Section 5 with delimited continuations; let us briefly illustrate it with the simpler example of call/cc:

> Call/cc was introduced in the Scheme programming language [17] as a Church encoding of Reynolds's escape operator [65]. A typed version of it is available in Standard ML of New Jersey [46] and Griffin has identified its logical content [43]. It is endowed with a variety of specifications: a CPS interpreter [47, 65], a denotational semantics [51], a big-step operational semantics [46], the CEK machine [38], calculi in the form of reduction semantics [37], and a number of implementation techniques [18, 49]—not to mention its call-by-name version in the archival version of Krivine's machine [54].

Question: How do we know that all the specifications 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 interpreters. So by construction, all the specifications that are inter-derivable are consistent.

Normalization by evaluation: Finally, refocusing provides a guideline for constructing reduction-free normalization functions out of reduction-based ones [25]. The reduction-free normalization functions take the form of eval/apply abstract machines, which usually are in defunctionalized form, which paves the way to writing normalization functions as usually encountered in the area of normalization by evaluation. We have illustrated the method with weak reduction and weak-head normalization (i.e., evaluation), but it also works for strong reduction and normalization, thus linking one-step reduction, abstract machines for strong reduction, and normalization functions.

Acknowledgments

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

References

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

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

[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*, 2005. To appear. 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 (revised version). Research Report BRICS RS-05-11, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2005. Accepted for publication in Logical Methods in Computer Science. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[8] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. Research Report BRICS RS-05-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[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. Research Report BRICS RS-05-10, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2005. Accepted for publication in the Journal of Functional Programming as a theoretical pearl.

[11] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Research Report BRICS RS-05-16, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[12] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. Research Report BRICS RS-05-13, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2005. Extended version of an article to appear in Information Processing Letters.

[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 [77], pages 333–340.

[21] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. In Danvy [28]. 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. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, number 124 in Electronic Notes in Theoretical Computer Science, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

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

[27] 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.

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

[29] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [77], pages 151–160.

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

[31] 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.

[32] 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.

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

[34] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. Technical Report 615, Computer Science Department, Indiana University, Bloomington, Indiana, June 2005.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In Danvy [28]. To appear. Available online at `http://www.pps.jussieu.fr/~krivine/`.

[55] Peter Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998.

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

[57] 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.

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

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

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

[61] Michel Parigot. $\lambda\mu$-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.

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

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

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

[72] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.

[73] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[74] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

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

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

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

[78] 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.

Recent BRICS Report Series Publications

RS-05-22 **Małgorzata Biernacka and Olivier Danvy.** ***A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*****. July 2005. iv+39 pp.**

RS-05-21 **Philipp Gerhardy and Ulrich Kohlenbach.** ***General Logical Metatheorems for Functional Analysis*****. July 2005. 65 pp.**

RS-05-20 **Ivan B. Damgård, Serge Fehr, Louis Salvail, and Christian Schaffner.** ***Cryptography in the Bounded Quantum Storage Model*****. July 2005.**

RS-05-19 **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas qLuttik.** ***Finite Equational Bases in Process Algebra: Results and Open Questions*****. June 2005. 28 pp. To appear in the LNCS series in Jan Willem Klop's 60th birthday volume.**

RS-05-18 **Peter Bogetoft, Ivan B. Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft.** ***Secure Computing, Economy, and Trust: A Generic Solution for Secure Auctions with Real-World Applications*****. June 2005. 37 pp.**

RS-05-17 **Ivan B. Damgård, Thomas B. Pedersen, and Louis Salvail.** ***A Quantum Cipher with Near Optimal Key-Recycling*****. May 2005.**

RS-05-16 **Dariusz Biernacki, Olivier Danvy, and Kevin Millikin.** ***A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*****. May 2005. ii+24 pp.**

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

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

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

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