On Evaluation Contexts, Continuations, and the Rest of the Computation

Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

Abstract

Continuations are variously understood as representations of the current evaluation context and as representations of the rest of the computation, but these understandings contradict each other: plugging an expression in a context yields a new expression whereas sending an intermediate result to a continuation yields the final answer. We show that continuations-as-evaluation-contexts are the defunctionalized representation of the continuation of a single-step reduction function and that continuations-as-the-rest-of-the-computation are the continuation of an evaluation function. Furthermore, we show that defunctionalizing the continuation of an evaluator gives rise to the same evaluation contexts as in the single-step reducer. The only difference is how these evaluation contexts are interpreted: a 'plug' interpretation yields one-step reduction, whereas a 'refocus' interpretation yields evaluation.

We then present a constructive corollary of Reynolds's historical warning about depending on the evaluation order of a meta-language for an interpreter: The two best-known abstract machines for the λ-calculus, Krivine's machine and Felleisen et al.'s CEK machine, are in fact the call-by-name and call-by-value counterparts of the *same* (evaluation-order dependent) interpreter for the λ-calculus.

1 Introduction

The notion of continuation is ubiquitous in many different areas of computer science, including logic, constructive mathematics, programming languages, and programming. Nevertheless, continuations are a remarkably elusive, even mystifying, notion. They pop up virtually everywhere as a uniform solution to control-related problems, and it seems that no two alternative solutions to these problems are alike. Worse, no particular effort seems to have been devoted to connecting these alternative solutions to the solutions based on continuations and from there, to transpose these alternative solutions to other domains.

* Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

† Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. Email: `danvy@brics.dk`

CW'04 January 17, 2004, Venice, Italy.

1.1 Continuations, informally

What is a continuation? To some, it is the representation of an evaluation context, i.e., an expression with a hole; plugging an expression into this hole yields a new expression. To others, a continuation is a representation of the rest of the computation; sending it an intermediate result yields the final result of the overall computation. These two notions are plausible and even widespread (the latter one is actually the original definition [63]), but they are incompatible with each other. In the former case, a continuation expects an expression and returns another expression. In the latter case, a continuation expects a value and returns a final result.

The primary goal of this article is to reconcile these two common, but contradictory, understandings of continuations as representations of the current context and as representations of the rest of the computation.

1.2 Continuations, authoritatively

When they are mentioned at all, continuations are presented with considerable variations in textbook and lecture notes. In *Concepts in Programming Languages* [47], Mitchell briefly defines a continuation as a function representing the remaining program to evaluate; he mentions continuation-passing style as a way to obtain tail recursion. In *Programming Languages: Theory and Practice* [40], Harper summarily defines a continuation as a control stack and argues that a formal semantics is much clearer; he mentions continuation-passing style as a way to "roll one's own" continuation. In *Compiling with Continuations* [5], Appel defines a continuation as a function that expresses what to do next; he then makes a substantial use of continuation-passing style. In *Essentials of Programming Languages* [34], Friedman, Wand, and Haynes define a continuation as an abstraction of the control context; they dedicate two chapters to continuation-passing interpreters and transforming programs into continuation-passing style. In *Programming Languages and Lambda Calculi* [28], Felleisen and Flatt define a continuation as an inside-out evaluation context in an abstract machine; they do not consider continuation-passing style. In *Lisp in Small Pieces* [53], Queinnec defines a continuation as a representation of all that remains to compute; he mentions contexts as an alternative representation of continuations.

A secondary goal of this article is to unify these common, but distinct, representations of continuations. Our thesis is that Reynolds's defunctionalization provides the key to this unification, in the sense that control stacks and evaluation contexts are defunctionalized continuations.

1.3 Prerequisites

We expect a passing familiarity with functional programming (ML), and we build on the notions of evaluators, abstract machines, CPS transformation, defunctionalization, and syntactic theories:

Evaluation functions: An evaluator is a compositional function mapping an abstract-syntax tree to an expressible value, if there is one; it implements a denotational semantics [58].

Abstract machines: An abstract machine is a transition function over computational states; it implements an operational semantics [52].

CPS transformation: A program is transformed into continuation-passing style (CPS) by naming all of its intermediate results, sequentializing their computation, and introducing continuations. Each CPS transformation encodes an evaluation order [20, 41, 51, 57, 62].

Defunctionalization: A program is defunctionalized by replacing each of its function spaces by a first-order data type and a first-order apply function [56]. Each data type enumerates all the function abstractions that may give rise to inhabitants of the corresponding function space [7, 8, 13, 21, 49, 56, 65].

A particular case of defunctionalization is closure conversion: in an evaluator, closure conversion amounts to replacing each of the function spaces in expressible and denotable values by a tuple, and inlining the corresponding apply function [46, 56]. (Other styles of closure conversion exist, though [6].)

Syntactic theories: A syntactic theory provides a reduction relation on expressions by defining syntax, values, evaluation contexts, and redexes [26, 28, 70]. For example, a syntactic theory for arithmetic expressions is specified as follows.

Syntax: $e ::= n \mid e+e$

Values: n

Redexes: $n+n'$

Evaluation contexts: $E ::= [\,] \mid E[n+[\,]] \mid E[[\,]+e]$

Plugging an expression e into a context E:

$$\begin{array}{rcl} plug([\,],e) & = & e \\ plug(E[n+[\,]],e) & = & plug(E,n+e) \\ plug(E[[\,]+e'],e) & = & plug(E,e+e') \end{array}$$

Reduction relation: $E[n+n'] \rightarrow E[n'']$, where n'' is the sum of n and n'.

These definitions satisfy a "unique decomposition" lemma [70]: any expression e that is not a value can be uniquely decomposed into an evaluation context E and a redex $n+n'$ such that $e = plug(E,n+n')$.

From syntactic theory to abstract machine: Nielsen and the author have established the conditions under which one can deforest an evaluation function when it is defined as the transitive closure of one-step reduction in a syntactic theory [22]. At each step, a term is decomposed into an evaluation context and a redex, the redex is contracted, and the contractum is plugged into the evaluation context. Deforesting such an evaluation function makes it possible to avoid the construction of intermediate expressions. Our key point is to construct a "refocus" function that makes it possible to replace the decompose-contract-plug-decompose-contract-plug-... loop by an initial decomposition followed by a contract-refocus-contract-refocus-... loop. The result is an abstract machine.

For example, here is the refocus function corresponding to the syntactic theory just above:

$$\begin{array}{rcl} refocus([\,],n) & = & n \\ refocus(E[n'+[\,]],n) & = & refocus(E,n'+n) \\ refocus(E[[\,]+e],n) & = & decompose(e,E[n+[\,]]) \end{array}$$

where *decompose* decomposes a computation into an evaluation context and a redex.

1.4 Overview

The rest of this article is organized as follows. We first investigate continuations as evaluation contexts and continuations as the rest of the computation; to this end, we revisit the simple example of arithmetic expressions above (Section 2). We then consider the λ-calculus (Section 3) and analyze further consequences (Section 4).

2 A simple example: arithmetic expressions

To investigate continuations as evaluation contexts and continuations as the rest of the computation, we go through the simple exercise of writing a one-step reduction function and then an evaluator for arithmetic expressions. We write each of them in direct style, and we successively CPS-transform them and then defunctionalize their continuations.

Our arithmetic expressions are minimal: they consist of literals and additions.

```
datatype exp = VALUE of value
             | COMP of comp
and    value = LIT of int
and     comp = ADD of exp * exp
```

Literals are the only values and additions are the only computations.

2.1 A one-step reduction function

We write the one-step reduction function by recursive descent, using the recursive calls to reach the left-most-innermost redex, and constructing the reduced expression at return time:

```
(*  reduce1 : comp -> exp  *)
fun reduce1 (ADD (VALUE (LIT n1), VALUE (LIT n2)))
    = VALUE (LIT (n1 + n2))
  | reduce1 (ADD (VALUE v1, COMP c2))
    = COMP (ADD (VALUE v1, reduce1 c2))
  | reduce1 (ADD (COMP c1, e2))
    = COMP (ADD (reduce1 c1, e2))
```

We then CPS-transform `reduce1`:

```
(*  reduce1c : comp * (exp -> 'a) -> 'a  *)
fun reduce1c (ADD (VALUE (LIT n1), VALUE (LIT n2)), k)
    = k (VALUE (LIT (n1 + n2)))
  | reduce1c (ADD (VALUE v1, COMP c2), k)
    = reduce1c (c2, fn e2 => k (COMP (ADD (VALUE v1, e2))))
  | reduce1c (ADD (COMP c1, e2), k)
    = reduce1c (c1, fn e1 => k (COMP (ADD (e1, e2))))
```

Finally, we defunctionalize the continuations in `reduce1c`. We assume an initial continuation that is the identity function, and therefore the polymorphic type variable in the type of `reduce1c` is specialized to `exp`. Three functional abstractions can build inhabitants in the function space `exp -> exp`. The first is the initial continuation and it has no free variables. The second is the continuation in the second clause, and it has `v1` and `k` as free variables. The third is the continuation in the third clause, and it has `e2` and `k` as free variables. The data type of defunctionalized continuations has thus three constructors.

```
datatype cont = CONT0
              | CONT1 of value * cont
              | CONT2 of exp * cont

(*  apply : cont * exp -> exp  *)
fun apply (CONT0, e)
    = e
  | apply (CONT1 (v1, k), e2)
    = apply (k, COMP (ADD (VALUE v1, e2)))
  | apply (CONT2 (e2, k), e1)
    = apply (k, COMP (ADD (e1, e2)))

(*  reduce1cd : comp * cont -> exp  *)
fun reduce1cd (ADD (VALUE (LIT n1), VALUE (LIT n2)), k)
    = apply (k, VALUE (LIT (n1 + n2)))
  | reduce1cd (ADD (VALUE v1, COMP c2), k)
    = reduce1cd (c2, CONT1 (v1, k))
  | reduce1cd (ADD (COMP c1, e2), k)
    = reduce1cd (c1, CONT2 (e2, k))
```

We observe that the data type `cont` is isomorphic to the data type of evaluation contexts for arithmetic expressions, and that its apply function coincides with the corresponding plug function. *Evaluation contexts, together with their plug function, are therefore a representation of the continuation of a one-step reduction function.*

2.2 An evaluation function

We write an evaluation function by recursive descent:

```
(*  eval : exp -> int  *)
fun eval (VALUE (LIT n))
    = n
  | eval (COMP (ADD (e1, e2)))
    = (eval e1) + (eval e2)

(*  main : exp -> int  *)
fun main e
    = eval e
```

We then CPS-transform `eval`:

```
(*  evalc : exp * (int -> 'a) -> 'a  *)
fun evalc (VALUE (LIT n), k)
    = k n
  | evalc (COMP (ADD (e1, e2)), k)
    = evalc (e1,
             fn n1 => evalc (e2,
                             fn n2 => k (n1 + n2)))

(*  main : exp -> int  *)
fun main e
    = eval (e, fn n => n)
```

Finally, we defunctionalize the continuations in `evalc`. The initial continuation is the identity function and therefore the polymorphic type variable in the type of `evalc` is specialized to `int`. Three functional abstractions can build inhabitants in the function space `int -> int`. The first is the initial continuation and it has no free variables. The second is the inner continuation in the `ADD` clause, and it has `n1` and `k` as free variables. The third is the outer continuation in the `ADD` clause, and it has `e2` and `k` as free variables. The data type of defunctionalized continuations thus has three constructors. Due to the recursive call to `evalc` in the outer continuation, the apply function of defunctionalized continuations and the defunctionalized version of `evalc` are mutually recursive:

```
datatype cont = CONT0
              | CONT1 of int * cont
              | CONT2 of exp * cont

(*  apply : cont * int -> int  *)
fun apply (CONT0, n)
    = n
  | apply (CONT1 (n1, k), n2)
    = apply (k, n1 + n2)
  | apply (CONT2 (e2, k), n1)
    = evalcd (e2, CONT1 (n1, k))

(*  evalcd : exp * cont -> int  *)
and evalcd (VALUE (LIT n), k)
    = apply (k, n)
  | evalcd (COMP (ADD (e1, e2)), k)
    = evalcd (e1, CONT2 (e2, k))

(*  main : exp -> int  *)
fun main e
    = eval (e, CONT0)
```

We observe that the data type `cont` is isomorphic to the data type of evaluation contexts for arithmetic expressions, and that its apply function coincides with the corresponding refocus function. *Evaluation contexts, together with their refocus function, are therefore a representation of the continuation of an evaluation function.*

2.3 Conclusion

Continuations have two sides: they can represent the context for one-step reduction and they can represent the rest of the computation for evaluation. Common to both sides is the notion of evaluation context:

$$\begin{tikzcd} & \text{evaluation contexts} \arrow[dl, "\text{plug}"'] \arrow[dr, "\text{refocus}"] & \\ \text{one-step reduction} & & \text{evaluation} \end{tikzcd}$$

- Evaluation contexts, together with the plug interpretation, are the defunctionalized representation of the continuation of a one-step reducer.
- Evaluation contexts, together with the refocus interpretation, are the defunctionalized representation of the continuation of an evaluator.

Identifying these two representations of evaluation contexts makes it possible to reconcile the two common—but contradictory—understandings of continuations as representations of the current context and as representations of the rest of the computation.

Evaluation contexts were first proposed in Felleisen's PhD thesis [26] and since then they have had a clear impact in the formal study of programming languages. Yet they have never before been formally connected with the continuation of a one-step reduction function or with the continuation of an evaluation function.

It takes some skill to define evaluation contexts. Until the unique-decomposition lemma is proven, one is never sure whether the enumeration is complete and whether it is not somehow redundant. In contrast, the characterization of evaluation contexts as a defunctionalized continuation in a recursive descent to locate the next redex provides both a guideline and a security. Also, the unique-decomposition lemma holds as a corollary when one starts from a compositional recursive descent.

```
structure Eval0
= struct
    datatype expval = FUNCT of denval -> expval
    withtype denval = expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (fn v => eval (t, v :: e))
      | eval (APP (t0, t1), e)
        = let val (FUNCT f) = eval (t0, e)
          in f (eval (t1, e))
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

Figure 1. Canonical evaluation-order dependent evaluator

```
structure Eval1
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (t, e)
      | eval (APP (t0, t1), e)
        = let val (FUNCT (t', e')) = eval (t0, e)
          in eval (t', (eval (t1, e)) :: e')
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

Figure 2. Evaluator of Figure 1, closure-converted

3 A constructive corollary of Reynolds's evaluation-order dependence

In earlier work, the author and his students have observed that a defunctionalized CPS program implements an abstract machine [2, 3, 4, 10, 11, 18]. In particular, we have found that Krivine's abstract machine [15, 39, 45] is the defunctionalized and continuation-passing counterpart of a closure-converted call-by-value evaluator for the λ-calculus and that Felleisen et al.'s CEK machine [28, 29, 33] is the defunctionalized and continuation-passing counterpart of a closure-converted call-by-name evaluator for the λ-calculus [2].

The goal of this section is to show that Krivine's abstract machine and the CEK machine can in fact be derived from the same evaluator for the λ-calculus. This evaluator is the most canonical one for the λ-calculus: it is in direct style, higher-order, compositional, and with an environment. As pointed by Reynolds [56], it is also evaluation-order dependent: if the evaluation order of the defining language is call by name (resp. call by value), the evaluation order of the defined language is also call by name (resp. call by value). We specify this evaluation order with the corresponding CPS transformation:

- Our implementation of the abstract syntax of the λ-calculus is as follows:

  ```
  datatype term = IND of int    (* de Bruijn index *)
                | ABS of term
                | APP of term * term
  ```

  Variables are represented by their lexical offset (i.e., their de Bruijn index).

- Figure 1 displays an evaluation-order dependent evaluator in the concrete syntax of Standard ML. This evaluator is compositional (all recursive calls on the right side of the equal sign are made over proper sub-parts of the terms on the left side) and higher order (the domain of expressible values is a function space), with an environment (a list of denotable values).

- Figure 2 displays a first-order counterpart of the evaluator of Figure 1, again in the syntax of Standard ML. This evaluator was obtained by in-place defunctionalization of the expressible values, i.e., closure conversion [46, 56].

```
structure Eval1n
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = (expval -> expval) -> expval

    (*  eval : term * denval list * (expval -> expval)  *)
    (*          -> expval                                *)
    fun eval (IND n, e, k)
        = List.nth (e, n) k
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t', e')) =>
            eval (t', (fn k' => eval (t1, e, k')) :: e', k))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

Figure 3. Call-by-name CPS counterpart of Figure 2

```
structure Eval1v
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    (*  eval : term * denval list * (expval -> expval)  *)
    (*          -> expval                                *)
    fun eval (IND n, e, k)
        = k (List.nth (e, n))
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t', e')) =>
            eval (t1, e, fn v1 =>
            eval (t', v1 :: e', k)))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

Figure 4. Call-by-value CPS counterpart of Figure 2

```
structure Eval1nd
= struct
    datatype expval = FUNCT of term * denval list
         and denval = THUNK of term * denval list

    datatype cont = CONT0
                  | CONT1 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = let val (THUNK (t', e')) = List.nth (e, n)
          in eval (t', e', k)
          end
      | eval (ABS t', e', CONT1 (t1, e, k))
        = eval (t', (THUNK (t1, e)) :: e', k)
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT1 (t1, e, k))
      | eval (ABS t, e, CONT0)
        = FUNCT (t, e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

Figure 5. Defunctionalized counterpart of Figure 3

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t, e]$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t, nil, nil\rangle$
$\langle n, e, s\rangle$	$\Rightarrow$	$\langle t', e', s\rangle$ where $nth(e, n) = [t', e']$
$\langle \lambda t', e', [t_1, e] :: s\rangle$	$\Rightarrow$	$\langle t', [t_1, e] :: e', s\rangle$
$\langle t_0\, t_1, e, s\rangle$	$\Rightarrow$	$\langle t_0, e, [t_1, e] :: s\rangle$
$\langle \lambda t, e, nil\rangle$	$\Rightarrow$	$[t, e]$

The abstract machine operates on triples consisting of a term, an environment, and a stack of expressible values.

Each line in the table matches a clause in Figure 5.

Figure 6. Krivine's abstract machine

```
structure Eval1vd
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    datatype cont = CONT0
                  | CONT1 of denval * cont
                  | CONT2 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = apply (k, List.nth (e, n))
      | eval (ABS t, e, k)
        = apply (k, FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT2 (t1, e, k))
    and apply (CONT2 (t1, e, k), v0)
        = eval (t1, e, CONT1 (v0, k))
      | apply (CONT1 (FUNCT (t', e'), k), v1)
        = eval (t', v1 :: e', k)
      | apply (CONT0, v)
        = v

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

Figure 7. Defunctionalized counterpart of Figure 4

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t, e]$
- Evaluation contexts:

$$k ::= \texttt{CONT0} \mid \texttt{CONT1}(v, k) \mid \texttt{CONT2}(t, e, k)$$

- Initial transition, transition rules, and final transition:

t	$\Rightarrow_{init}$	$\langle t, nil, \texttt{CONT0}\rangle$
$\langle n, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, v\rangle$ where $nth(e, n) = v$
$\langle \lambda t, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, [t, e]\rangle$
$\langle t_0\, t_1, e, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0, e, \texttt{CONT2}(t_1, e, k)\rangle$
$\langle \texttt{CONT2}(t_1, e, k), v_0\rangle$	$\Rightarrow_{apply}$	$\langle t_1, e, \texttt{CONT1}(v_0, k)\rangle$
$\langle \texttt{CONT1}([t', e'], k), v_1\rangle$	$\Rightarrow_{apply}$	$\langle t', v_1 :: e', k\rangle$
$\langle \texttt{CONT0}, v\rangle$	$\Rightarrow_{final}$	v

The abstract machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a term, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and an expressible value.

Each line in the table matches a clause in Figure 7.

Figure 8. The CEK machine

- Figure 3 displays the call-by-name CPS counterpart of the evaluator of Figure 2.
- Figure 4 displays the call-by-value CPS counterpart of the evaluator of Figure 2.
- Figure 5 displays the defunctionalized version of the evaluator of Figure 3, with the corresponding apply function inlined. Merging the domains of expressible values and of denotable values into one (recursive) domain of thunks pairing terms and environments, and representing the data type `cont` as a list yields the transition function of Krivine's machine (Figure 6).
- Figure 7 displays the defunctionalized version of the evaluator of Figure 4. It corresponds to the transition function of the CEK machine (Figure 8).

Therefore, if the CPS transformation is call-by-name, the resulting transition function is that of Krivine's abstract machine, and if the CPS transformation is call-by-value, the resulting transition function is that of the CEK machine:

$$\begin{tikzcd}
 & \text{canonical evaluator (Figure 1)} \arrow[d, "\text{closure conv.}"] & \\
 & \text{Figure 2} \arrow[dl, "\text{call-by-name CPS transf.}"'] \arrow[dr, "\text{call-by-value CPS transf.}"] & \\
\text{Figure 3} \arrow[d, "\text{defunct.}"'] & & \text{Figure 4} \arrow[d, "\text{defunct.}"] \\
\text{Krivine's machine (Figures 5 and 6)} & & \text{CEK machine (Figures 7 and 8)}
\end{tikzcd}$$

Reynolds's point was that in general the evaluation order of the defining language, in a definitional interpreter, determines the evaluation order of the defined language if the definitional interpreter is in direct style (and does not use thunks). The author and his students have recently shown that a call-by-name interpreter leads one to Krivine's abstract machine and that a call-by-value interpreter leads one to the CEK machine [2]. It is a further (and new) consequence of the embodiment of evaluation order in a CPS transformation [41] that Krivine's abstract machine and the CEK machine can in fact be derived from the *same* canonical evaluator. In particular, other CPS transformations would lead to other abstract machines.

Krivine's abstract machine has been discovered, the CEK machine has been invented, and each of them has been celebrated independently and on its own right. Yet, as shown here, they are two sides of the same coin.

4 Consequences

We review further consequences of the connection between evaluation contexts, continuations, and the rest of the computation.

4.1 Designing syntactic theories and abstract machines

Beside making it simple to connect one-step reducers and evaluators, the interpretation of evaluation contexts with a plug function or with a refocus function has direct consequences for designing syntactic theories and abstract machines:

- For programming languages where one can write a one-step reducer using recursive descent, one can mechanically construct the grammar of evaluation contexts and the corresponding plug function, and rest assured that the unique-decomposition lemma holds [70].

  Furthermore, given such a one-step reduction machinery, one can mechanically construct the corresponding abstract machine [22].
- For programming languages where one can write an evaluator using recursive descent, one can mechanically construct the grammar of evaluation contexts and the corresponding refocus function. The result is the transition function of an abstract machine.

  Conversely, one can see abstract machines such as Krivine's machine and the CEK machine as defunctionalized continuation-passing interpreters.

The two points above are not just an academic observation—they have concrete consequences in that they have made it possible for the author and his students to uniformly transform a given evaluator into an abstract machine that was independently invented or discovered, to uniformly exhibit the evaluator underlying a given abstract machine, and to design new evaluators, new abstract machines, and new virtual machines [1, 2]. Beside Krivine's machine and the CEK machine, examples include Landin's SECD machine, Hannan and Miller's CLS machine, Curien et al.'s Categorical Abstract Machine, Schmidt's VEC machine, and Leroy's Zinc machine as well as abstract machines for non-strict functional languages [3], logic-programming languages [11], functional languages with computational effects, including the security technique of stack inspection [4], imperative languages, and object-oriented languages. In clear contrast, such evaluators and machines are usually considered independently and on a case-by-case basis. And when abstract machines are derived, it is the medium (i.e., the derivation) rather than the result that tends to be the message [66, 67].

In particular, starting from a monad-based evaluator for the lambda-calculus, we can pick an arbitrary monad and mechanically construct an evaluator, an abstract machine, and a syntactic theory for the corresponding computational effect. In striking contrast, abstract machines and syntactic theories for computational effects have been designed in isolation and reported as such in the literature.

On the other hand, syntactic theories have also been successfully used in situations where the unique-decomposition lemma does not hold, e.g., Concurrent ML [55]. Such situations would require first-class continuations, which are out of scope here.

4.2 Normalization

Another application of the the insight presented in Section 3 and of the derivation reported at PPDP 2003 [2] concerns (not necessarily type-directed) normalization functions as encountered in the area of normalization by evaluation [9, 16, 25]. The author and his students have derived abstract machines as well as virtual machines for normalization [1]. Specifically, we have shown that a call-by-name normalization function yields a machine that generalizes Krivine's machine, and that a call-by-value normalization function yields a machine that generalizes the CEK machine. In the light of Section 3, though, the author now realizes that these two machines are in fact derived from the *same* normalization function.

In noticeable contrast, existing machines for normalization have been designed in isolation rather than by derivation [15, 36].

4.3 Delimited continuations

In CPS, all calls are tail calls. Yet in some situations, it is very convenient to re-initialize a continuation and to mix CPS with non-tail calls. In a program that re-initializes continuations and where not all calls are tail calls, a continuation no longer represents the rest of the computation. Instead, it is delimited by the re-initialization. Capturing such a continuation yields a first-class continuation that returns to its point of activation. Such first-class continuations can be composed. (In contrast, first-class continuations obtained by `call/cc`-like control operators do not return to their point of activation and therefore they cannot be composed.)

Fifteen years ago, Felleisen introduced an operator to delimit control (a "prompt") together with other operators to abstract delimited control [27]. These control operators were specified using a representation of control as a list of activation records. Delimiting control amounted to putting a mark on this list, abstracting delimited control amounted to making a copy of the list up to the closest mark, and activating a delimited continuation amounted to concatenating the copied list to the current list of activation records [30]. Felleisen's work triggered a series of alternative control operators, all based on representing control as a list of activation records interspersed with control marks [38, 42, 43, 48, 54, 59, 60].

To the author, Felleisen's operator for delimiting control fitted precisely a pervasive pattern of functional programming with layered continuations, together with another control operator, `shift` [20]. Consequently, the two control operators to delimit and to abstract control enjoy a number of applications—in fact, they correspond to computational monads [31]—and they are still the topic of study today [35, 44]. Furthermore, they generalize directly to the CPS hierarchy [10, 19, 23], which also corresponds to layered monads [32].

These two lines of work have been opposed because one represents control as a list of activation records, as in an initial algebra, and the other as a continuation function, as in a final algebra [30]. This opposition continues today when control is only considered as a list of activation records, fit for arbitrary surgery.[1] The two representations, however, could be synergized, e.g., by seeing the former as a defunctionalized version of the latter and by identifying when the latter is not a functional version of the former. For example, Felleisen's $\mathcal{F}^+$ control operator appears to have no CPS counterpart [20, Section 5.3].

Another advantage of characterizing delimited continuations using repeated CPS transformations is that, through the derivation outlined in Section 3 (closure conversion, CPS transformations (note the plural), and defunctionalization), one obtains abstract machines for delimited control [17]. In these machines, delimited control is represented through a series of control stacks, one for each layered continuation.[2]

[1]The danger of this surgery is that it is so plausible. For example, in the first implementation of Lisp, it was sweepingly plausible to push the bindings of the formals and the actuals on the stack at call time, and to pop them off at return time. The result was dynamic scope.

4.4 Landin's SECD machine

Imagine an environment-based, call-by-value evaluator for the λ-calculus with a callee-save strategy, that furthermore delimits control when evaluating the body of a λ-abstraction. This evaluator operates on the same representation of λ-terms as in Section 3.

```
datatype value = FUN of value -> value

(*  eval : term * value list -> value * value list  *)
fun eval (IND n, e)
    = (List.nth (e, n), e)
  | eval (ABS t, e)
    = (FUN (fn v => reset (fn () => #1 (eval (t, v :: e)))),
       e)
  | eval (APP (t0, t1), e)
    = let val (v1, e) = eval (t1, e)
          val (v0, e) = eval (t0, e)
      in apply (v0, v1, e)
      end
(*  apply : value * value * value list ->
            value * value list  *)
and apply (FUN f, v, e)
    = (f v, e)

(*  evaluate : term -> value  *)
fun evaluate t
    = reset (fn () => #1 (eval (t, nil)))
```

From this evaluator, one can reconstruct Landin's SECD machine as follows:

1. closure conversion of the function space in the domain of values;
   ```
   datatype value = FUN of term * E
   withtype E = value list
   ```
2. introduction of a data stack to hold the intermediate results of `eval`;
   ```
   eval : term * S * E -> S * E
   withtype S = value list
        and E = value list
   ```

[2]The author wishes to emphasize this point with an anecdote about Gasbichler and Sperber's implementation of delimited continuations in Scheme 48 [35]. In the course of their work, Gasbichler and Sperber consulted each of the authors of control operators for delimited control, to make sure that their implementation of each delimited-control operator was accurate. This consultation apparently took some time to stabilize. In sharp contrast, it reduced to one e-mail reply from the author, with the guideline of checking that the CPS counterpart of the implementation matches the CPS specification of shift and reset. The next time the author heard of Gasbichler and Sperber's work, it was in the list of accepted papers at ICFP 2002.

3. CPS transformation;

```
eval : term * S * E * C -> value
withtype S = value list
     and E = value list
     and C = S * E -> value
```

4. second CPS transformation, to get rid of the non-tail call due to the presence of `reset`;

```
eval : term * S * E * C * D -> 'a
withtype S = value list
     and E = value list
     and C = S * E * D -> 'a
     and D = value -> 'a
```

5. defunctionalization of the two layered continuations;

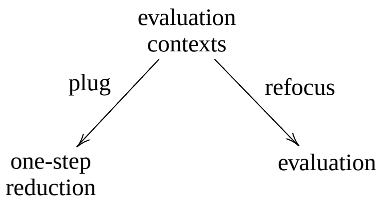
6. fusion of the resulting mutually recursive functions into one.

The SECD machine is a transition function operating on a four-component state: a stack register, an environment register, a control register, and a dump register [46]. The stack register holds the data stack introduced above; the environment register holds the environment threaded in the evaluator above; the control register holds the first continuation in defunctionalized form; and the dump register holds the second continuation in defunctionalized form. We therefore claim that the denotational essence of the SECD machine is this evaluator, with its callee-save strategy for the environment and its control delimiter. The rest—stack register, control register, and dump register—are mere operational artifacts.

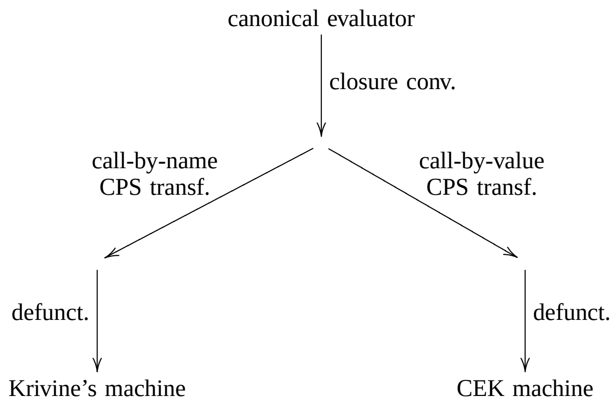
This derivation is documented in a BRICS technical report [18]. It is based on the insight of Section 2 and at the origin of the derivation of Section 3. It solves a long-standing open problem about the particular architecture of the SECD machine, which had never been fully explained—though many variations and simplifications exist. These variations and simplifications (as well as arbitrary new ones) can be obtained by tuning this evaluator and then transforming it into an abstract machine. For example, omitting the control delimiter (which operationally is unused) yields an SEC machine.

5 Conclusion and current work

We have reconciled the notion of continuations as evaluation contexts with the notion of continuations as representations of the rest of the computation. To this end, we have factored the continuation of a single-step reducer and the continuation of an evaluator as the same evaluation contexts with two different interpretations:



As a consequence of this factorization, we have shown that the two best-known abstract machines for the λ-calculus can be derived from the same canonical evaluator for the λ-calculus:



This derivation provides a constructive corollary of Reynolds's historical warning about the evaluation order of defining languages [56] and it scales to normalization functions and abstract machines for normalization. It is an instance of a functional correspondence that lets one reconstruct known abstract machines, construct new ones, e.g., with monadic computational effects, systematically equip them with stack inspection [14], and mechanically construct the corresponding syntactic theories.

In this article, we have considered continuations in the operational setting of reduction, evaluation, and normalization. They are, however, ubiquitous in many other areas, such as semantics [64] and logic [37] as well as in operating-systems services [24, 69].

Acknowledgments:

The author is grateful to Lasse Reichstein Nielsen for our joint study of defunctionalization, which led us to refocusing and the mechanical construction of abstract machines from one-step reduction functions, a study now jointly continued with Małgorzata Biernacka. Thanks are also due to Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard for our joint study of the correspondence between evaluation functions, abstract machines, and virtual machines, and to Hayo Thielecke for the opportunity to present this work at CW'04.

This article has benefited from Julia Lawall's comprehensive as well as timely comments.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

6 References

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

[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. Technical Report BRICS RS-03-24, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Accepted for publication in Information Processing Letters.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[6] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[7] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, Sendai, Japan, October 2001. Springer-Verlag.

[8] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[10] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004. To appear.

[11] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Presented at the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

[12] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[13] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Smolka [61], pages 56–71.

[14] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspections. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, number 2618 in Lecture Notes in Computer Science, pages 22–37, Warsaw, Poland, April 2003. Springer-Verlag.

[15] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [68], pages 333–340.

[16] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[17] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Smolka [61], pages 88–103.

[18] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.

[19] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [68], pages 151–160.

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

[21] 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.

[22] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. Technical Report BRICS RS-02-04, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. 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.

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

[24] Richard Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 122–136, Pacific Grove, California, October 1991.

[25] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

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

[27] 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.

[28] 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.

[29] 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.

[30] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[31] Andrzej Filinski. Representing monads. In Boehm [12], pages 446–457.

[32] 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.

[33] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[34] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[35] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Peyton Jones [50], pages 271–282.

[36] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Peyton Jones [50], pages 235–246.

[37] 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.

[38] 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.

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

[40] Robert Harper. Programming languages: Theory and practice. Working Draft. http://www.cs.cmu.edu/~rwh/plbook/, Spring Semester, 2002.

[41] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [12], pages 458–471.

[42] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.

[43] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[44] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

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

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

[47] John Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[48] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

[49] Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.

[50] Simon Peyton Jones, editor. *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, Pittsburgh, Pennsylvania, September 2002. ACM Press.

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

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

[53] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, 1996.

[54] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.

[55] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[56] 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.

[57] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[58] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

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

99, January 1990.

[60] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [68], pages 161–175.

[61] Gert Smolka, editor. *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany, March 2000. Springer-Verlag.

[62] 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.

[63] 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.

[64] Hayo Thielecke. From control effects to typed continuation passing. In Greg Morrisett, editor, *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 139–149, New Orleans, Louisiana, January 2003. ACM Press.

[65] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[66] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[67] Mitchell Wand. A semantic prototyping system. In Susan L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 213–221, Montréal, Canada, June 1984. ACM Press.

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

[69] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, 1999. Reprinted from the proceedings of the 1980 Lisp Conference, with a foreword.

[70] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.