BRICS

Basic Research in Computer Science

BRICS RS-03-33 O. Danvy: A Rational Deconstruction of Landin's SECD Machine

A Rational Deconstruction of Landin's SECD Machine

Olivier Danvy

BRICS Report Series **RS-03-33**

ISSN 0909-0878 **October 2003**

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**

A Rational Deconstruction of Landin's SECD Machine

Olivier Danvy

BRICS *

Department of Computer Science

University of Aarhus †

October 2003

Abstract

Landin's SECD machine was the first abstract machine for the λ-calculus viewed as a programming language. Both theoretically as a model of computation and practically as an idealized implementation, it has set the tone for the subsequent development of abstract machines for functional programming languages. However, and even though variants of the SECD machine have been presented, derived, and invented, the precise rationale for its architecture and modus operandi has remained elusive. In this article, we deconstruct the SECD machine into a λ-interpreter, i.e., an evaluation function, and we reconstruct λ-interpreters into a variety of SECD-like machines. The deconstruction and reconstructions are transformational: they are based on equational reasoning and on a combination of simple program transformations—mainly closure conversion, transformation into continuation-passing style, and defunctionalization.

The evaluation function underlying the SECD machine provides a precise rationale for its architecture: it is an environment-based eval-apply evaluator with a callee-save strategy for the environment, a data stack of intermediate results, and a control delimiter. Each of the components of the SECD machine (stack, environment, control, and dump) is therefore rationalized and so are its transitions.

The deconstruction and reconstruction method also applies to other abstract machines and other evaluation functions. It makes it possible to systematically extract the denotational content of an abstract machine in the form of a compositional evaluation function, and the (small-step) operational content of an evaluation function in the form of an abstract machine.

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

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

Contents

1 Introduction

Forty years ago, Peter Landin wrote a profoundly influencial article, "The Mechanical Evaluation of Expressions" [38], where, in retrospect, he outlined a substantial part of the functional-programming research programme for the following decades. This visionary article stands out for advocating the use of the λ-calculus as a meta-language and for introducing the first abstract machine for the λ-calculus (i.e., in Landin's terms, applicative expressions), the SECD machine. However, and in addition, it also introduces the notions of 'syntactic sugar' over a core programming language; of 'closure' to represent functional values; of circularity to implement recursion; of thunks to delay computations; of delayed evaluation; of partial evaluation; of disentangling nested applications into where-expressions at preprocessing time; of what has since been called de Bruijn indices; of sharing; of what has since been called graph reduction; of call by need; of what has since been called strictness analysis; and of domain-specific languages—all concepts that are ubiquitous in programming languages today. The topic of this article is the SECD machine.

Since "The Mechanical Evaluation of Expressions," many other abstract machines for the λ-calculus have been invented, discovered, or derived [20]. In fact, the literature simply abounds with derivations of abstract machines—though with one remarkable exception: there is no derivation of Landin's original SECD machine, even though it was the first such abstract machine. The SECD machine is the starting point of many university courses and textbooks and it has been the topic of many variations and optimizations, be it for its source language (call by name, call by need, other syntactic constructs, including control operators), for its environment (de Bruijn indices, de Bruijn levels, explicit substitutions, higher-order abstract syntax), or for its control (proper tail recursion, one stack instead of two). Yet in forty years of existence, it has not been derived or reconstructed. The common agreement is that there is something special, something original and still unexplained about the SECD machine.

The goal of this article is to pinpoint and explain the originality of the SECD machine. To this end, we show how to mechanically deconstruct the SECD machine into an evaluator for applicative expressions and then how to rationally reconstruct a variety of SECD-like machines. This deconstruction–reconstruction is actually interesting in itself because it provides a bridge between small-step operational semantics (in the form of an abstract machine) and denotational semantics (in the form of a compositional evaluation function). It is also general because it applies to other evaluators and other abstract machines [2]. The derivation is based on a combination of simple, correct, and well-known program-transformation tools, each of which is reviewed in appendix: CPS transformation [17, 52], delimited continuations [16], defunctionalization [18, 48], and closure conversion [38]. In fact, these transformations are so classical that one could almost say that the present work could have been carried out years ago, would it be only for Piet Hein's gentle reminder that Things Take Time [31].

1.1 Deconstruction of the SECD machine

The SECD machine is defined as one transition function over a quadruple—a stack of intermediate values (of type `S`), an environment (of type `E`), a control stack (of type `C`), and a dump (of type `D`):

```
run : S * E * C * D -> value
```

This transition function is complicated because it has several induction variables. Our single creative step is to first disentangle it into four transition functions, each of which has one induction variable, i.e., operates on one element of the quadruple:

```
run_c : S * E * C * D -> value
run_d : S * D -> value
run_t : term * S * E * C * D -> value
run_a : S * E * C * D -> value
```

Depending on the control stack, `run_c` dispatches towards `run_d` if the control stack is empty, `run_t` if the top of the control stack contains a term, and `run_a` if the top of the control stack contains an apply directive.

- We observe that these four functions are in defunctionalized form (the control stack and the dump are defunctionalized data types and two of the four functions are the corresponding apply functions), and we refunctionalize them, eliminating the two apply functions:

  ```
  run_t : term * S * E * C * D -> value
  run_a : S * E * C * D -> value
  where C = S * E * D -> value
        D = S -> value
  ```

- We observe that the result is in continuation-passing style, and we transform it back to direct style, eliminating the dump continuation:

  ```
  run_t : term * S * E * C -> stack
  run_a : S * E * C -> S
  where C = S * E -> S
  ```

- We observe that the result is almost in continuation-passing style, modulo the reinitialization of a continuation when evaluating the body of a λ-abstraction, and we transform it back to direct style with a control delimiter, eliminating the control continuation:

  ```
  run_t : S * E -> S * E
  run_a : S * E -> S * E
  ```

- We observe that the result threads a data stack of intermediate results, and we rewrite it to do without, eliminating the stack:

  ```
  run_t : term * E -> value * E
  run_a : value * value * E -> value * E
  ```

- We observe that the result is in closure-converted form, and we unconvert it, eliminating the closures.
- We observe that the result is a compositional evaluator in direct style.

Given a disentangled—though altogether not unexpectable—transition function for the SECD machine, all the observations above are in some sense unavoidable as well as economical—though the author is well aware that to a man with a hammer, the world looks like a nail. The order of these transformations, however, is not fixed. Both closure unconversion and data-stack elimination could occur earlier in the deconstruction.

1.2 Denotational content of the SECD machine

The end result of the deconstruction outlined in Section 1.1 shows that the denotational content of the SECD machine is a (curried) evaluation function of type

```
term -> E -> value * E
```

where `term` is the type of a term, `value` is the type of a value, and `E` is the type of an environment mapping variables to values. This evaluator maps a term `t` into an ML function. This denotation maps an environment `e` in which to evaluate `t` into a pair `(v, e')`, where `v` is the value corresponding to `t` and `e'` is the same environment as `e`.

This evaluator is traditional in that it is composed of one 'eval' function (`run_t` above) to evaluate terms, and one 'apply' function (`run_a` above) to apply functions. (An alternative to this traditional eval–apply model is the push-enter model of Krivine's machine [36] and of the spineless tagless G-machine [44].) This evaluator, however, is also unconventional in that:

1. its environment is managed in a callee-save fashion (witness the environment paired with the resulting value), and
2. it uses a control delimiter to evaluate the body of λ-abstractions.

It seems to us that these two properties account both for the specificity and for the intriguing originality of Landin's SECD machine:

Specificity: The two properties show that the evaluation mechanism of the SECD machine is environment-based, that the environment is threaded and saved in a callee-save fashion, and that the body of each λ-abstraction is evaluated afresh. The rest—closures, stack, control, and dump—are inessential programming artefacts.

Originality: Environments are usually managed in a caller-save fashion in interpreters, and relatively rare are programs that use delimited continuations. (In fact, control delimiters were invented a quarter of a century after the SECD machine [15, 16, 23, 25].)

1.3 Overview

We first detail the deconstruction of the SECD machine into a compositional evaluator in direct style (Section 2). We then illustrate how to reconstruct a variety of SECD-like machines (Section 3), including one with an instruction set, and we conclude.

1.4 Prerequisites and domain of discourse

We use ML as a meta-language. We assume a basic familiarity with Standard ML and with reasoning about ML programs. In particular, given two ML expressions `e` and `e'` we write `e` $\cong$ `e'` to express that `e` and `e'` are observationally equivalent.

The source language. The source language is the λ-calculus, extended with literals (as observables). A program is a closed term.

```
structure Source
= struct
    type ide = string
    datatype term = LIT of int
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
    type program = term
  end
```

The (polymorphic) environment. We make use of a structure `Env` satisfying the following signature:

```
signature ENV
= sig
    type 'a env
    val empty : 'a env
    val extend : Source.ide * 'a * 'a env -> 'a env
    val lookup : Source.ide * 'a env -> 'a
  end
```

The empty environment is denoted by `Env.empty`. The function extending an environment with a new binding is denoted by `Env.extend`. The function fetching the value of an identifier from an environment is denoted by `Env.lookup`.

Expressible and denotable values. There are three kinds of values: integers, the successor function, and function closures:

```
datatype value = INT of int
               | SUCC
               | CLOSURE of value Env.env * Source.ide * Source.term
```

Following Landin [38], function closures pair a λ-abstraction (i.e., its formal parameter and its body) and the environment of its declaration.

The initial environment. We define the successor function in the initial environment:

```
val e_init = Env.extend ("succ", SUCC, Env.empty)
```

2 Deconstruction of the SECD machine

We now substantiate the deconstruction outlined in Section 1.1.

Section 2.1 presents the SECD machine as originally specified and classically presented in the literature, i.e., as one tail-recursive transition function `run`. Section 2.2 presents an alternative specification where `run` is disentangled into four mutually (tail) recursive transition functions `run_c`, `run_d`, `run_t`, and `run_a`, each of which has one induction variable. This disentangled definition is in defunctionalized form, and Section 2.3 presents its higher-order counterpart. This counterpart is in continuation-passing style, and Section 2.4 presents its direct-style equivalent. This equivalent is almost in continuation-passing style, which is characteristic of delimited control. Section 2.5 presents the corresponding direct-style evaluator, which uses a control delimiter. This evaluator uses a data stack of intermediate results. Section 2.6 presents the corresponding stackless evaluator. This evaluator is in closure-converted form. Section 2.7 present the corresponding higher-order evaluator. This evaluator is compositional and assessed in Section 2.8.

In addition, Section 2.9 reviews the J operator.

2.1 The original specification of the SECD machine

The SECD machine is a transition function over a state with four components:

- A *stack* register holding a list of intermediate results. This component has type `value list`.
- An *environment* register holding the current environment. This component has type `value Env.env`.
- A *control* register holding a list of control directives. This component has type `directive`, where `directive` is defined as follows:

```
datatype directive = TERM of Source.term
                   | APPLY
```

- A *dump* register holding a list of triples. Each triple contains snapshots of the stack, environment, and control registers. This component has type `(value list * value Env.env * directive list) list`.

The SECD machine is defined with a set of transitions between its four components. Here is its transitive closure:

```
(*  run : S * E * C * D -> value  *)
(*  where S = value list          *)
(*        E = value Env.env       *)
(*        C = directive list      *)
(*        D = (S * E * C) list    *)
fun run (v :: nil, e', nil, nil)                                    (* 1 *)
    = v
  | run (v :: nil, e', nil, (s, e, c) :: d)                         (* 2 *)
    = run (v :: s, e, c, d)
  | run (s, e, (TERM (LIT n)) :: c, d)                              (* 3 *)
    = run ((INT n) :: s, e, c, d)
  | run (s, e, (TERM (VAR x)) :: c, d)                              (* 4 *)
    = run ((Env.lookup (x, e)) :: s, e, c, d)
  | run (s, e, (TERM (LAM (x, t))) :: c, d)                         (* 5 *)
    = run ((CLOSURE (e, x, t)) :: s, e, c, d)
  | run (s, e, (TERM (APP (t0, t1))) :: c, d)                       (* 6 *)
    = run (s, e, (TERM t1) :: (TERM t0) :: APPLY :: c, d)
  | run (SUCC :: (INT n) :: s, e, APPLY :: c, d)                    (* 7 *)
    = run ((INT (n+1)) :: s, e, c, d)
  | run ((CLOSURE (e', x, t)) :: v' :: s, e, APPLY :: c, d)         (* 8 *)
    = run (nil, Env.extend (x, v', e'), (TERM t) :: nil, (s, e, c) :: d)
```

```
(*  evaluate0 : Source.program -> value  *)
fun evaluate0 t                                                     (* 9 *)
    = run (nil, e_init, (TERM t) :: nil, nil)
```

Essentially:

1. The first clause specifies what to do if both the current list of control directives and the current dump are empty, which corresponds to terminating the computation: the value on top of the stack is returned.

2. The second clause specifies what to do if the current list of control directives is empty but the current dump is not empty, which corresponds to a function return: the computation should continue with the stack, environment, and control stored in the top-most component of the dump, transferring the top-most value of the current stack onto the new stack.

3. The third clause specifies what to do if the top current control directive is a literal, which corresponds to evaluating this literal: the corresponding value should be pushed on the current stack.

4. The fourth clause specifies what to do if the top current control directive is an identifier, which corresponds to evaluating this identifier: the corresponding value should be fetched from the current environment and pushed on the current stack.

5. The fifth clause specifies what to do if the top current control directive is a λ-abstraction, which corresponds to evaluating this λ-abstraction: the corresponding function closure should be pushed on the current stack. This closure groups the current environment and the two components of the λ-abstraction, i.e., its formal parameter and its body.

6. The sixth clause specifies what to do if the top current control directive is an application, which corresponds to evaluating an application: an apply directive, the operator, and the operand should be pushed on the list of control directives.

7. The seventh clause specifies what to do if the top current control directive is an apply directive, the top of the current stack is the successor function, and the next element in the current stack is an integer, which corresponds to the application of the successor function: the current stack should be popped twice and the integer should be incremented and pushed on the stack.

8. The eighth clause specifies what to do if the top current control directive is an apply directive, the top of the current stack is a closure, and there is a next element in the current stack, which corresponds to a function call: the stack should be popped twice and, together with the current environment and the rest of the list of control directives, pushed on the dump (thereby saving the current state of the machine). The current stack should be initialized with the empty list, the current environment should be initialized with the closure environment, suitably extended, and the current list of directives should be initialized with the body of the closure.

9. Evaluation is initialized with an empty current stack, the initial environment, the expression to evaluate as a single control directive, and an empty dump.

The SECD machine does not terminate for divergent source terms. If it becomes stuck, an ML pattern-matching error is raised (alternatively, the co-domain of `run` could be made `value option` and an else clause could be added). Otherwise, the result of the evaluation is `v` for some ML value `v : value`.

2.2 A more structured specification

In the definition of Section 2.1, all the possible transitions are meshed together in one recursive function, `run`. Let us factor `run` into several mutually recursive functions, each of them with one induction variable.

In this disentangled definition,

- `run_c` interprets the list of control directives, i.e., it specifies which transition to take if the list is empty, starts with a term, or starts with an apply directive. If the list is empty, it calls `run_d`. If the list starts with a term, it calls `run_t`, caching the term in an extra component (the first parameter of `run_t`). If the list starts with an apply directive, it calls `run_a`.

- `run_d` interprets the dump, i.e., it specifies which transition to take if the dump is empty or non-empty, given a valid stack.
- `run_t` interprets the top term in the list of control directives.
- `run_a` interprets the top value in the current stack.

```
(*  run_c : S * E * C * D -> value                 *)
(*  run_d : S * D -> value                         *)
(*  run_t : Source.term * S * E * C * D -> value   *)
(*  run_a : S * E * C * D -> value                 *)
(*    where S = value list                         *)
(*          E = value Env.env                      *)
(*          C = directive list                     *)
(*          D = (S * E * C) list                   *)
fun run_c (s, e, nil, d)
    = run_d (s, d)
  | run_c (s, e, (TERM t) :: c, d)
    = run_t (t, s, e, c, d)
  | run_c (s, e, APPLY :: c, d)
    = run_a (s, e, c, d)
and run_d (v :: nil, nil)
    = v
  | run_d (v :: nil, (s, e, c) :: d)
    = run_c (v :: s, e, c, d)
and run_t (LIT n, s, e, c, d)
    = run_c ((INT n) :: s, e, c, d)
  | run_t (VAR x, s, e, c, d)
    = run_c ((Env.lookup (x, e)) :: s, e, c, d)
  | run_t (LAM (x, t), s, e, c, d)
    = run_c ((CLOSURE (e, x, t)) :: s, e, c, d)
  | run_t (APP (t0, t1), s, e, c, d)
    = run_t (t1, s, e, (TERM t0) :: APPLY :: c, d)
and run_a (SUCC :: (INT n) :: s, e, c, d)
    = run_c ((INT (n+1)) :: s, e, c, d)
  | run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, d)
    = run_t (t, nil, Env.extend (x, v', e'), nil, (s, e, c) :: d)

(*  evaluate1 : Source.program -> value  *)
fun evaluate1 t
    = run_t (t, nil, e_init, nil, nil)
```

Proposition 1 (full correctness) *For any ML value* `t : Source.program`,

$$\texttt{evaluate1 t} \;\cong\; \texttt{evaluate0 t}$$

Proof: By equational reasoning and fixed-point induction [58]. The invariants are as follows. For any ML values `s : S`, `e : E`, `c : C`, `d : D`, and `t : Source.term`,

$$\begin{cases} \texttt{run\_c (s, e, c, d)} \cong \texttt{run (s, e, c, d)} \\ \texttt{run\_d (s, d)} \cong \texttt{run (s, e, nil, d)} \\ \texttt{run\_t (t, s, e, c, d)} \cong \texttt{run (s, e, (TERM t) :: c, d)} \\ \texttt{run\_a (s, e, c, d)} \cong \texttt{run (s, e, APPLY :: c, d)} \end{cases}$$

□

2.3 A higher-order counterpart

In the disentangled definition of Section 2.2, there are two possible ways to construct a dump (nil and cons) and three possible ways to construct a list of control directives (nil, cons'ing a term, and cons'ing an apply directive). (We could phrase these constructions as two data types rather than as two lists.)

These data types, together with `run_d` and `run_c`, are in the image of defunctionalization (`run_d` and `run_c` are the apply functions of these two data types). The corresponding higher-order evaluator reads as follows.

```
(*  run_t : Source.term * S * E * C * D -> value  *)
(*  run_a : S * E * C * D -> value                *)
(*    where S = value list                        *)
(*          E = value Env.env                     *)
(*          C = (S * E * D) -> value              *)
(*          D = S -> value                        *)
fun run_t (LIT n, s, e, c, d)
    = c ((INT n) :: s, e, d)
  | run_t (VAR x, s, e, c, d)
    = c ((Env.lookup (x, e)) :: s, e, d)
  | run_t (LAM (x, t), s, e, c, d)
    = c ((CLOSURE (e, x, t)) :: s, e, d)
  | run_t (APP (t0, t1), s, e, c, d)
    = run_t (t1, s, e,
             fn (s, e, d) => run_t (t0, s, e,
                                    fn (s, e, d) => run_a (s, e, c, d),
                                    d),
             d)
and run_a (SUCC :: (INT n) :: s, e, c, d)
    = c ((INT (n+1)) :: s, e, d)
  | run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, d)
    = run_t (t, nil, Env.extend (x, v', e'),
             fn (s, _, d) => d s,
             fn (v :: nil) => c (v :: s, e, d))

(*  evaluate2 : Source.program -> value  *)
fun evaluate2 t
    = run_t (t, nil, e_init,
             fn (s, _, d) => d s,
             fn (v :: nil) => v)
```

The resulting evaluator is in continuation-passing style, with two nested continuations. It inherits the characteristics of the SECD machine, i.e., it threads a stack of intermediate results, an environment, a control continuation, and a dump continuation. As an evaluator, it is a bit unusual in that:

1. it has two continuations (`C` and `D`),
2. it threads a stack of intermediate results (`S`), and
3. the environment is saved by the recursive callees, not by the callers. (Usually, the environment is not threaded but saved across recursive calls.)

Otherwise the interpreter follows the traditional eval–apply schema identified by McCarthy in his definition of Lisp in Lisp [41], by Reynolds in his definitional interpreters [48], and by Steele and Sussman in their lambda-papers [51–54]: `run_t` is eval and `run_a` is apply.

Proposition 2 (full correctness) *For any ML value* `p : Source.program`*,*

`evaluate2 p` $\cong$ `evaluate1 p`*.*

Proof: Defunctionalizing `evaluate2` yields `evaluate1`, and defunctionalization has been proved correct [5, 43]. □

2.4 A dump-less direct-style counterpart

The evaluator of Section 2.3 is in continuation-passing style and therefore it is in the image of the CPS transformation [11]. Its direct-style counterpart reads as follows, renaming `run_t` as `eval` and `run_a` as `apply`.

```
(*   eval : Source.term * S * E * C -> stack  *)
(*  apply : S * E * C -> S                    *)
(*    where S = value list                    *)
(*          E = value Env.env                 *)
(*          C = S * E -> S                    *)
fun eval (LIT n, s, e, c)
    = c ((INT n) :: s, e)
  | eval (VAR x, s, e, c)
    = c ((Env.lookup (x, e)) :: s, e)
  | eval (LAM (x, t), s, e, c)
    = c ((CLOSURE (e, x, t)) :: s, e)
  | eval (APP (t0, t1), s, e, c)
    = eval (t1, s, e, fn (s, e) =>
      eval (t0, s, e, fn (s, e) =>
      apply (s, e, c)))
and apply (SUCC :: (INT n) :: s, e, c)
    = c ((INT (n+1)) :: s, e)
  | apply ((CLOSURE (e', x, t)) :: v' :: s, e, c)
    = let val (v :: nil) = eval (t, nil, Env.extend (x, v', e'),
                                 fn (s, _) => s)
      in c (v :: s, e)
      end
```

```
(*  evaluate3 : Source.program -> value  *)
fun evaluate3 t
    = let val (v :: nil) = eval (t, nil, e_init, fn (s, _) => s)
      in v
      end
```

Proposition 3 (full correctness) *For any ML value* `p : Source.program`*,*

`evaluate3 p` $\cong$ `evaluate2 p`.

Proof: CPS-transforming `evaluate3` yields `evaluate2`, and the CPS transformation is meaning-preserving. □

2.5 A control-less direct-style counterpart

All but two of the calls to `eval` are tail calls in the evaluator of Section 2.4. Thus, except for these two calls, the evaluator is in CPS. These two calls are characteristic of delimited continuations [16,23]. To account for them, we use the control delimiter `reset`. (Operationally, this control delimiter is moot because no continuations are captured [16, 34]. It can therefore simply be defined as taking a thunk and forcing it, as we do below; in general of course, the definition is not as simple [26]. Section 3.6 analyzes the consequences of omitting reset altogether.) With such a definition of `reset`, the direct-style counterpart of the evaluator reads as follows:

```
(*  (* mock-up *) reset : (unit -> 'a) -> 'a  *)
fun reset thunk
    = thunk ()

(*   eval : Source.term * S * E -> S * E  *)
(*  apply : S * E -> S * E                *)
(*    where S = value list                *)
(*          E = value Env.env             *)
fun eval (LIT n, s, e)
    = ((INT n) :: s, e)
  | eval (VAR x, s, e)
    = ((Env.lookup (x, e)) :: s, e)
  | eval (LAM (x, t), s, e)
    = ((CLOSURE (e, x, t)) :: s, e)
  | eval (APP (t0, t1), s, e)
    = let val (s, e) = eval (t1, s, e)
          val (s, e) = eval (t0, s, e)
      in apply (s, e)
      end
and apply (SUCC :: (INT n) :: s, e)
    = ((INT (n+1)) :: s, e)
  | apply ((CLOSURE (e', x, t)) :: v' :: s, e)
    = let val (v :: nil, _)
               = reset (fn () => eval (t, nil, Env.extend (x, v', e')))
      in (v :: s, e)
      end
```

```
(*  evaluate4 : Source.program -> value  *)
fun evaluate4 t
    = let val (v :: nil, _)
              = reset (fn () => eval (t, nil, e_init))
      in v
      end
```

Proposition 4 (full correctness) *For any ML value* `p : Source.program`*,*

$$\texttt{evaluate4 p} \cong \texttt{evaluate3 p}.$$

Proof: CPS-transforming `evaluate4` yields `evaluate3`, and the CPS transformation is meaning-preserving. □

2.6 A stack-less counterpart

In the evaluator of Section 2.5, `eval` and `apply` thread a data stack of intermediate results. The stackless counterpart of this evaluator reads as follows.

```
(*   eval : Source.term * E -> value * E    *)
(*  apply : value * value * E -> value * E  *)
(*    where E = value Env.env               *)
fun eval (LIT n, e)
    = (INT n, e)
  | eval (VAR x, e)
    = (Env.lookup (x, e), e)
  | eval (LAM (x, t), e)
    = (CLOSURE (e, x, t), 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
and apply (SUCC, INT n, e)
    = (INT (n+1), e)
  | apply (CLOSURE (e', x, t), v', e)
    = let val (v, _)
              = reset (fn () => eval (t, Env.extend (x, v', e')))
      in (v, e)
      end
```

```
(*  evaluate5 : Source.program -> value  *)
fun evaluate5 t
    = let val (v', _)
              = reset (fn () => eval (t, e_init))
      in v'
      end
```

Proposition 5 (full correctness) *For any ML value* `p : Source.program`*,*

$$\texttt{evaluate5 p} \cong \texttt{evaluate4 p}.$$

Proof: By equational reasoning and fixed-point induction. The invariants are as follows, postscripting `eval` and `apply` with a `5` for `evaluate5` and `eval` and `apply` with a `4` for `evaluate4`.

For any ML values `t : Source.term`, `e : E`, and `v`, `v0`, and `v1 : value`,

$$\begin{cases} \texttt{eval5 (t, e)} \cong \texttt{(v, e)} & \textit{iff} \text{ for all } \texttt{s : value list}, \\ & \quad \texttt{eval4 (t, s, e)} \cong \texttt{(v :: s, e)} \\ \texttt{apply5 (v0, v1, e)} \cong \texttt{(v, e)} & \textit{iff} \text{ for all } \texttt{s : value list}, \\ & \quad \texttt{apply4 (v0 :: v1 :: s, e)} \cong \texttt{(v :: s, e)} \end{cases}$$

□

2.7 A compositional counterpart

The evaluators of Sections 2.3, 2.4, 2.5, and 2.6 represent functional values with closures. In Section 1.4, this representation was epitomized by the definition of values:

```
datatype value = INT of int
               | SUCC
               | CLOSURE of value Env.env * Source.ide * Source.term
```

A function closure pairs a source λ-abstraction and the environment of its declaration.

Because of this representation, none of the evaluators above are compositional in the sense of denotational semantics [49, 55, 58].[1] On the other hand, because they use closures, these evaluators are in closure-converted form. We closure-unconvert the latest one as follows.

```
datatype value = INT of int
               | SUCC
               | FUN of value -> value

(*   eval : Source.term -> E -> value * E   *)
(*  apply : value * value * E -> value * E  *)
(*    where E = value Env.env               *)
fun eval (LIT n, e)
    = (INT n, e)
  | eval (VAR x, e)
    = (Env.lookup (x, e), e)
  | eval (LAM (x, t), e)
    = (FUN (fn v
            => reset (fn ()
                      => let val (v', _)
                                 = eval (t, Env.extend (x, v, e))
                         in v'
                         end)),
       e)
```

[1] To be compositional, they should solely define the meaning of each term as a composition of the meaning of its parts.

```
  | eval (APP (t0, t1), e)
    = let val (v1, e) = eval (t1, e)
          val (v0, e) = eval (t0, e)
      in apply (v0, v1, e)
      end
and apply (SUCC, INT n, e)
    = (INT (n+1), e)
  | apply (FUN f, v, e)
    = (f v, e)

(*  evaluate6 : Source.program -> value  *)
fun evaluate6 t
    = reset (fn () => let val (v', _) = eval (t, e_init)
                      in v'
                      end)
```

Proposition 6 (full correctness) *For any ML value* `p : Source.program`*,*

$$\texttt{evaluate6 p} \cong \texttt{evaluate5 p}.$$

Proof: Closure-converting `evaluate6` yields `evaluate5`, and closure conversion is meaning-preserving. □

The evaluator above is not unique, though. We can also choose a callee-save representation of functions:

```
datatype value = INT of int
               | SUCC
               | FUN of value * value Env.env
                        -> value * value Env.env

fun ...
  | eval (LAM (x, t), e)
    = (FUN (fn (v, e')
            => reset (fn ()
                      => let val (v', _)
                                 = eval (t, Env.extend (x, v, e))
                         in (v', e')
                         end)),
       e)
and ...
  | apply (FUN f, v, e)
    = f (v, e)
```

In this evaluator, functions are passed the environment of their caller together with their actual parameter and they return it with their result. With such an interpreter, it would be very simple to obtain dynamic scope—in the clause for λ-abstractions, one would just replace `e` by `e'` in the recursive call to `eval`.

2.8 Assessment

Through a series of meaning-preserving steps, we have transformed the SECD machine (i.e., a transition function) into an evaluator (i.e., a compositional evaluation function). For each of these language processors—the original one, the intermediate ones, and the final one—evaluating an ill-typed source term is undefined (i.e., in ML, evaluation gets stuck and a pattern-matching error is raised); evaluating a divergent source term diverges; and evaluating a well-typed and convergent source term converges to a value.

It seems to us that this deconstruction of the SECD machine into an evaluation function sheds a new light on it. Its stack, environment, control, and dump registers are explained as artefacts of a particular evaluation algorithm: environment-based with a callee-save strategy, right-to-left call by value, and with one data stack for intermediate results and two continuations, the inner one for the current λ-abstraction. In Section 3, we show how different evaluation algorithms give rise to different SECD machines.

On a structural note, we also observe that defunctionalizing the function space of an evaluator leads one to deep closures (i.e., closures pointing to the current branch of the environment tree), whereas defunctionalizing the function space of a normal program leads one to flat closures (i.e., closures pointing to a minimal copy of the values of the variables occurring free in a λ-abstraction). Flat closures in an interpreter therefore yield deep closures in interpreted programs.

2.9 Landin's J operator

Shortly after "The Mechanical Evaluation of Expressions," Landin wrote "A Generalization of Jumps and Labels" [37], in which he introduced first-class control in programming languages, with the control operator J. J is a precursor of call/cc in Scheme [35], and it has been described in the literature every ten years henceforth [9, 22, 56]. The present deconstruction sheds a new light on it (in a nutshell, J gives access to the meta-continuation of the interpreter) but this new light distracts from the main point of this article—how to deconstruct and then reconstruct the SECD machine, and we will report on it elsewhere.

3 Reconstructions of SECD-like machines

Each of the deconstruction steps of Section 2 is reversible. In this section, we review briefly how to rationally reconstruct a variety of SECD-like machines.

3.1 The original SECD machine

Closure-converting the evaluator of Section 2.7, and then introducing a data stack, CPS-transforming the result twice, defunctionalizing the result into four mutually recursive transition functions, and merging them into one yields the original SECD machine.

3.2 A left-to-right SECD machine

Changing the evaluation algorithm so that sub-terms in an application are evaluated from left to right, and proceeding as outlined in Section 3.1 yields an SECD machine where sub-terms in an application are evaluated from left to right. Conversely, it is also simple to modify the SECD machine and to deconstruct the result into an evaluation function where sub-terms are evaluated from left to right. The relevant clause, in the evaluators of Section 2.7, reads as follows:

```
| eval (APP (t0, t1), e)
  = let val (v0, e) = eval (t0, e)
        val (v1, e) = eval (t1, e)
    in apply (v0, v1, e)
    end
```

3.3 A properly tail-recursive SECD machine

It is a simple programming exercise to make any of the evaluators above properly tail-recursive, by singling out the treatment of tail calls. The corresponding SECD machine is properly tail recursive as well. Conversely, it is also simple to modify the SECD machine to make it properly tail recursive and to deconstruct the result into a properly tail-recursive evaluation function.

- In the original version of the SECD machine (Section 2.1), one adds the following clause before the eighth:

  ```
  | run ((CLOSURE (e', x, t)) :: v' :: nil, e, APPLY :: nil, d)
    = run (nil, Env.extend (x, v', e'), (TERM t) :: nil, d)
  | run ((CLOSURE (e', x, t)) :: v' :: s, e, APPLY :: c, d)    (* 8 *)
    = ... (* as before *)
  ```

  If the control register contains only one directive and this directive is `APPLY`, the call is a tail call. The tail-call optimization consists in pushing nothing on the dump.

- In the disantangled version (Section 2.2), one adds the following clause to the definition of `run_a`:

  ```
  | run_a ((CLOSURE (e', x, t)) :: v' :: nil, e, nil, d)
    = run_t (t, nil, Env.extend (x, v', e'), nil, d)
  | run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, d)
    = ... (* as before *)
  ```

  If the control stack is empty, the call is a tail call. The tail-call optimization consists in pushing nothing on the dump.

  Merging the clauses of this version yields the one just above.

- In the higher-order version (Section 2.3), one can tag the control continuation with an inherited boolean flag indicating whether the current expression is in tail position, and extend run_a with a new clause:

```
| run_a ((CLOSURE (e', x, t)) :: v' :: nil, e, c, true, d)
  = run_t (t, nil, Env.extend (x, v', e'), c, true, d)
| run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, false, d)
  = ... (* as before *)
```

  If the flag is true, the call is a tail call. The tail-call optimization consists in composing no function with the dump continuation.

  Defunctionalizing this version yields the one just above.

- In the dump-less version (Section 2.4), c is also tagged with an inherited boolean flag and apply is extended with a new clause:

```
  | apply ((CLOSURE (e', x, t)) :: v' :: nil, e, c, true)
    = eval (t, nil, Env.extend (x, v', e'), c, true)
  | apply ((CLOSURE (e', x, t)) :: v' :: s, e, c, false)
  = ... (* as before *)
```

  If the flag is true, the source call is a tail call. The tail-call optimization consists in calling eval tail-recursively.

  CPS-transforming this version yields the one just above.

- In the control-less version (Section 2.5), the boolean flag is still inherited and apply is extended with a new clause:

```
| apply ((CLOSURE (e', x, t)) :: v' :: nil, e, true)
  = eval (t, nil, Env.extend (x, v', e'), true)
| apply ((CLOSURE (e', x, t)) :: v' :: s, e, false)
  = ... (* as before *)
```

  If the flag is true, the source call is a tail call. The tail-call optimization consists in calling eval tail-recursively.

  CPS-transforming this version yields the one just above.

- In the stack-less version (Section 2.6), the boolean flag is still inherited and apply is extended with a new clause:

```
| apply (CLOSURE (e', x, t), v', e, true)
  = eval (t, Env.extend (x, v', e'), true)
| apply (CLOSURE (e', x, t), v', e, false)
  = ... (* as before *)
```

  If the flag is true, the source call is a tail call. The tail-call optimization consists in calling eval tail-recursively.

  Introducing a stack in this version yields the one just above.

- The first compositional version (Section 2.7) is unsuited for tail-call optimization because functions are called non-tail recursively in the second clause of `apply`. The second version is better suited: The denotation of functions is passed a boolean flag indicating whether the call is a tail call:

```
datatype value = INT of int
               | SUCC
               | FUN of value * value Env.env * boolean
                        -> value * value Env.env

fun ...
  | eval (LAM (x, t), e, b)
    = (FUN (fn (v, e', true)
               => eval (t, Env.extend (x, v, e), true)
             | (v, e', false)
               => ... (* as before *)),
       e)
and ...
  | apply (FUN f, v, e, b)
    = f (v, e, b)
```

  If the flag is true, the source function is called tail-recursively. The tail-call optimization consists in calling `eval` tail-recursively.

  Closure-converting this version yields the one just above.

3.4 A call-by-name SECD machine

It is a simple programming exercise to make any of the evaluators above call by name, by delaying the evaluation of actual parameters with thunks [33]. More directly, one can also bypass the thunks and use a call-by-name CPS transformation [30]. The corresponding SECD machine follows call by name as well. Conversely, one can also modify the SECD machine to make it use thunks and to deconstruct the result into a call-by-name evaluation function.

3.5 A call-by-need SECD machine

Threading a heap of memo-thunks is the canonical way to implement call by need. The corresponding SECD machine follows call by need as well. In contrast, directly modifying the SECD machine to make it implement call by need requires considerably more insight. The idea is developed elsewhere [4].

3.6 An SEC machine

In Section 2.5, the control delimiter serves no operational purpose since no continuations are captured. Eliding it leads one to an abstract machine without dump. Deconstructing this abstract machine into an evaluation function yields an evaluator without control delimiters:

- Version 1:

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

fun ...
  | eval (LAM (x, t), e)
    = (FUN (fn v => let val (v', _) = eval (t, Env.extend (x, v, e))
                    in v'
                    end),
       e)
```

- Version 2:

```
datatype value = ... | FUN of value * value Env.env
                              -> value * value Env.env

fun ...
  | eval (LAM (x, t), e)
    = (FUN (fn (v, e')
            => let val (v', _) = eval (t, Env.extend (x, v, e))
               in (v', e')
               end),
       e)
```

As has been discussed in the literature [56], the dual existence of the control and dump components in the SECD machine led Landin to a slightly complicated control operator, J. Unifying these two components leads one to the traditional escape and call/cc control operators [35, 48].

3.7 An EC machine

An evaluator without a data stack still has to save intermediate results. The corresponding abstract machine saves them on the control stack.

The evaluator reads as follows:

```
fun eval (LIT n, e)
    = (INT n, e)
  | eval (VAR x, e)
    = (Env.lookup (x, e), e)
  | eval (LAM (x, t), e)
    = (FUN (fn v => let val (v', _) = eval (t, Env.extend (x, v, e))
                    in v'
                    end),
       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
```

```
and apply (SUCC, INT n, e)
    = (INT (n+1), e)
  | apply (FUN f, v, e)
    = (f v, e)
```

The abstract machine reads as follows:

```
datatype stackable = ENV of value Env.env
                   | TERM of Source.term
                   | VALUE of value

(*  run_c :         value * E * C -> value  *)
(*  run_t :   Source.term * E * C -> value  *)
(*  run_a : value * value * E * C -> value  *)
(*    where E = value Env.env               *)
(*          C = stackable list              *)
fun run_c (v, e, nil)
    = v
  | run_c (v, e, (ENV e') :: c)
    = run_c (v, e', c)
  | run_c (v1, e, (TERM t0) :: c)
    = run_t (t0, e, (VALUE v1) :: c)
  | run_c (v0, e, (VALUE v1) :: c)
    = run_a (v0, v1, e, c)
and run_t (LIT n, e, c)
    = run_c (INT n, e, c)
  | run_t (VAR x, e, c)
    = run_c (Env.lookup (x, e), e, c)
  | run_t (LAM (x, t), e, c)
    = run_c (CLOSURE (x, t, e), e, c)
  | run_t (APP (t0, t1), e, c)
    = run_t (t1, e, (TERM t0) :: c)
and run_a (SUCC, INT n, e, c)
    = run_c (INT (n+1), e, c)
  | run_a (CLOSURE (x, t, e'), v, e, c)
    = run_t (t, Env.extend (x, v, e'), (ENV e) :: c)
```

`run_c` interprets the control stack, `run_t` is 'eval', and `run_a` is 'apply'.

3.8 An SC machine

An alternative to environments is to use substitutions. In such an interpreter, there is no environment and therefore nothing for the callee to save. We leave it as an exercise to the reader.

3.9 A C machine

An evaluator that is substitution based and uses no data stack yields an abstract machine with a control stack. Again, we leave it as an exercise to the reader.

Suffice it to say that it is the same abstract machine as in Curien's lecture notes on abstract machines, control, and sequents [10].

3.10 Higher-order abstract syntax

Another alternative to environments is to use higher-order abstract syntax [45]. The corresponding machine is a higher-order version of the SECD machine without an E register [13].

3.11 de Bruijn indices

Names, in source terms, can be replaced by their lexical offset. The corresponding interpreters and abstract machines do not look up variables but fetch their values directly from the environment.

3.12 An instruction set for the SECD machine

Using Wand's technique of combinator-based compilers [57], the author and his students have factored the evaluation function corresponding to the SECD abstract machine into a byte-code compiler and a byte-code interpreter, i.e., a virtual machine [1]. The resulting instruction set coincides with Henderson's in his textbook on the application and implementation of functional programming [32]. Elsewhere, we also present a decompilation function for the virtual SECD machine [3].

3.13 Assessment

We have outlined how the series of meaning-preserving steps used to deconstruct the SECD machine into an evaluation function can be reversed to construct a variety of SECD machines. In fact, the author and his students have shown that this deconstruction–reconstruction methodology applies to other abstract machines than the SECD machine, e.g., Krivine's abstract machine, Felleisen et al.'s CEK machine and its many variants, Hannan and Miller's CLS machine, Schmidt's VEC machine, Curien et al.'s Categorical Abstract Machine, and Leroy's ZINC machine [1, 2], as well as for call-by-need evaluators and lazy abstract machines [4]. In fact, the method applies as well to other language paradigms than functional programming, e.g., logic programming [7] and also imperative programming and object-oriented programming. It also applies to constructing abstract machines for normalization from normalization functions [1, Section 3].

3.14 Related work

In his famous 700 follow-up article [39,42], Morris presents a "shorter equivalent" of the SECD machine as an interpreter written in an applicative language. We note, though, that while Morris's interpreter is definitely shorter, it is not strictly equivalent to the SECD machine. (For example, its environment is saved by the callers, not by the callees.) Indeed, defunctionalizing the CPS counterpart of Morris's interpreter yields a different abstract machine that has one control stack and no dump. (In fact, this abstract machine coincides with Felleisen et al.'s CEK abstract machine [21, 24].)

In a similar way, in "Call-by-name, call-by-value, and the λ-calculus" [46], Plotkin formalized the SECD machine with respect to a canonical, caller-save, evaluation function that is similar to Morris's. In the light of the reconstruction presented here, the correctness proof of the SECD machine reduces to proving the equivalence between a caller-save and a callee-save evaluation function, which is simpler.

4 Conclusion

We have characterized the denotational content of the SECD machine as an evaluator with a callee-save strategy for the environment and a control delimiter.[2] In doing so, we have outlined a methodology for extracting the denotational content of abstract machines in the form of a compositional evaluation function. This methodology is reversible and enables one to extract the (small-step) operational content of evaluation functions in the form of an abstract machine in a fairly mechanical way: one closure-converts its expressible and denotable values to make them first-order; one CPS-transforms the closure-converted evaluation function to make it tail-recursive, i.e., iterative, and to materialize its control flow into continuations; and one defunctionalizes these continuations to make the evaluation function first order, thereby obtaining a transition function, i.e., a finite-state, iterative abstract machine. Optionally, one introduces a data stack to hold intermediate results. The methodology also scales to other evaluation functions and other abstract machines; in particular, it applies directly to λ-calculi extended with computational effects à la Moggi, e.g., control and state, and to other language paradigms than functional programming [1, 2, 4, 7].

In passing, we have also presented a new application of defunctionalization and a new example of control delimiters in programming practice.

Acknowledgments

The rational deconstruction presented here arose because of a discussion with Mayer Goldberg in July 2002, at the occasion of our joint work on compilation and decompilation [3]. The author is also grateful to Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard for our subsequent joint study of the functional correspondence between evaluation functions and abstract machines [1, 2, 4, 7].

[2] Landin was aware that abstract machines are interpreters, witness his introduction of the SECD machine as a way of "interpreting" applicative expressions. (The quotes are his. The other quotes in the abstract of his article occur when he wrote that his article contributes to the "theory" of computing.)

A first version of this article was written in the early fall of 2002 [14]. It gave rise to presentations at the University of Tokyo in September 2002, at INRIA-Rocquencourt in December 2002, at the University of Rennes in December 2002, and at the 2.8 Working Group on functional programming in January 2003. At the time, there was no data-stack elimination.

The present version contains data-stack elimination and was written during the summer of 2003. It has benefited from the comments of Mads Sig Ager, Małgorzata Biernacka, Dariusz Biernacki, Julia Lawall, Jan Midtgaard, and Henning Korsholm Rohde. Thanks are also due to Harry Mairson, John Reynolds, and Mitchell Wand for their input about the title as well as for their encouraging words.

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

A Toolbox

In this appendix, we review the elements of the toolbox mentioned in Section 1.

A.1 CPS transformation

A λ-term is transformed into continuation-passing style (CPS) by naming each of its intermediate results, by sequentializing the computation of these results, and by introducing continuations. Equivalently, such a term can be first transformed into monadic normal form and then translated into the term model of the continuation monad [29]. The CPS transformation is abundantly described in the literature [19, 27, 47, 52].

For example, a term such as $\lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$ is named and sequentialized into

$$\begin{array}{l}\lambda f.\lambda g.\lambda x.\mathsf{let}\ v_1 = f\ x\\ \qquad\qquad \mathsf{in}\ \mathsf{let}\ v_2 = g\ x\\ \qquad\qquad\quad \mathsf{in}\ v_1\ v_2\end{array}$$

and its call-by-value CPS counterpart reads as

$$\lambda k.k\ (\lambda f.\lambda k.k\ (\lambda g.\lambda k.k\ (\lambda x.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.v_1\ v_2\ k))))).$$

In both the sequentialized version and the CPS version, v_1 names the result of $f\ x$ and v_2 names the result of $g\ x$.

A.2 Delimited continuations

A λ-term uses delimited continuations when some of its intermediate continuations are reinitialized to the identity function or when not all calls to a continuation are evaluation-order independent [16]. For example, in contrast to the CPS abstraction

$$\lambda f.\lambda k.f\ 42\ k$$

which is strictly in continuation-passing style (all calls are tail calls and all sub-terms are trivial), the non-CPS abstraction

$$\lambda f.\lambda k.k\ (f\ 42\ (\lambda a.a))$$

uses delimited continuations; the function denoted by f is passed an initial continuation, and the result of its application is sent to k. This term is therefore evaluation-order sensitive [46, 48]. The direct-style counterpart of the first abstraction,

$$\lambda f.f\ 42$$

is an ordinary λ-term, whereas the direct-style counterpart of the second,

$$\lambda f.reset(f\ 42)$$

uses the control delimiter *reset* [12, 16, 17, 26, 28, 34, 40]. Should the function denoted by f capture its continuation, it would capture all of it in the first case (and applying this captured continuation would be like a jump); in the second case, however, it would capture only a delimited part of the continuation (and applying this captured continuation would be like a call). In this article, we make no other use of reset than to reinitialize the continuation.

A.3 Defunctionalization

In a higher-order program, first-class functions occur as instances of function abstractions. Often, these function abstractions can be enumerated, either exhaustively or more discriminately using a control-flow analysis [50]. Defunctionalization is a program transformation where function types are replaced by an enumeration of the function abstractions in the source program.

Defunctionalization consumes the results of a control-flow analysis. A defunctionalizer replaces:

- function spaces by an enumeration, in the form of a data type, of the possible lambda-abstractions that can float there;
- function introduction by an injection into the corresponding data type; and
- function elimination by an apply function dispatching over elements of the corresponding data type.

For example, let us defunctionalize the following ML program:

```
fun aux f
    = (f 1) + (f 10)

fun main (x, y)
    = (aux (fn z => z)) * (aux (fn z => x + y + z))
```

The `aux` function is passed a first-class function, applies it to 1 and 10, and sums the results. The `main` function calls `aux` twice and multiplies the results. All in all, two function abstractions occur in this program, in `main`, as arguments of `aux`.

Defunctionalizing this program amounts to defining a data type with two constructors, one for each function abstraction, and its associated apply function. The first function abstraction contains no free variables and therefore the first data-type constructor is constant. The second function abstraction contains two free variables (`x` and `y`, of type integer), and therefore the second data-type constructor requires two integers.

In `main_def`, the first functional argument is thus introduced with the first constructor, and the second functional argument with the second constructor and the values of `x` and `y`. In `aux_def`, the functional argument is passed to a (second-class) function `apply` that eliminates it with a case expression dispatching over the two constructors.

```
datatype lam = LAM1
             | LAM2 of int * int

fun apply (LAM1, z)
    = z
  | apply (LAM2 (x, y), z)
     = x + y + z

fun aux_def f
    = (apply (f, 1)) + (apply (f, 10))

fun main_def (x, y)
    = (aux_def LAM1) * (aux_def (LAM2 (x, y)))
```

Defunctionalization was discovered by Reynolds thirty-two years ago [48]. Compared to closure conversion, it has been little used in practice since then, and has only been formalized over the last few years [5, 6, 43]. More detail can be found in Danvy and Nielsen's study [18]. The key observation here is that defunctionalizing a CPS program yields a transition function [2].

A.4 Closure conversion

In retrospect, closure conversion is a particular case of defunctionalization, where the function space has only one constructor and the apply function is inlined.

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 Mayer Goldberg. A symmetric approach to compilation and decompilation. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 296–331. Springer-Verlag, 2002.

[4] 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.

[5] 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.

[6] 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.

[7] 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).

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

[9] William H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[10] Pierre-Louis Curien. Abstract machines, control, and sequents. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 123–136, Caminha, Portugal, September 2000. Springer-Verlag.

[11] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[12] 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.

[13] Olivier Danvy. The mechanical evaluation of higher-order expressions. In *Preliminary proceedings of the 14th Conference on Mathematical Foundations of Programming Semantics*, London, UK, May 1998.

[14] Olivier Danvy. A lambda-revelation of the SECD machine. Technical Report BRICS RS-02-53, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2002.

[15] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[16] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

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

[18] 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.

[19] Olivier Danvy and Lasse R. Nielsen. On one-pass CPS transformations. Technical Report BRICS RS-02-03, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. Accepted in the Journal of Functional Programming.

[20] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[21] 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.

[22] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

[23] 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.

[24] 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.

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

[26] Andrzej Filinski. Representing monads. In Boehm [8], pages 446–457.

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

[28] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 271–282, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[29] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [8], pages 458–471.

[30] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

[31] Piet Hein. *Grooks*. The MIT Press, 1966.

[32] Peter Henderson. *Functional Programming – Application and Implementation*. Prentice-Hall International, 1980.

[33] Peter Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[34] 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.

[35] 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.

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

[37] Peter Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(1):125–143, 1998.

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

[39] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[40] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[41] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, 1962.

[42] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

[43] 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.

[44] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[45] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

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

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

[48] 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.

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

[50] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[51] Guy L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976.

[52] 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.

[53] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.

[54] Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[55] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

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

[57] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[58] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-03-33 Olivier Danvy. ***A Rational Deconstruction of Landin's SECD Machine*****. October 2003. 32 pp. This report supersedes the earlier BRICS report RS-02-53.**

RS-03-32 Philipp Gerhardy and Ulrich Kohlenbach. ***Extracting Herbrand Disjunctions by Functional Interpretation*****. October 2003. 17 pp.**

RS-03-31 Stephen Lack and Paweł Sobociński. ***Adhesive Categories*****. October 2003. 25 pp. Appears in Walukiewicz, editor,** ***Foundations of Software Science and Computation Structures*****, FoSSaCS '04 Proceedings, LNCS 2987, 2004, pages 273–288.**

RS-03-30 Jesper Makholm Byskov, Bolette Ammitzbøll Madsen, and Bjarke Skjernaa. ***New Algorithms for Exact Satisfiability*****. October 2003. 31 pp.**

RS-03-29 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. ***A Runtime System for XML Transformations in Java*****. October 2003. 15 pp.**

RS-03-28 Zoltán Ésik and Kim G. Larsen. ***Regular Languages Definable by Lindström Quantifiers*****. August 2003. 82 pp. This report supersedes the earlier BRICS report RS-02-20. Appears in** ***Theoretical Informatics and Applications*****, 37(3):179–241, 2003.**

RS-03-27 Luca Aceto, Willem Jan Fokkink, Rob J. van Glabbeek, and Anna Ingólfsdóttir. ***Nested Semantics over Finite Trees are Equationally Hard*****. August 2003. 31 pp. To appear in** ***Information and Computation*****.**

RS-03-26 Olivier Danvy and Ulrik P. Schultz. ***Lambda-Lifting in Quadratic Time*****. August 2003. 23 pp. Extended version of a paper appearing in Hu and Rodríguez-Artalejo, editors,** ***Sixth International Symposium on Functional and Logic Programming*****, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151. This report supersedes the earlier BRICS report RS-02-30.**