

A Rational Deconstruction of Landin’s SECD Machine

Olivier Danvy

BRICS*, Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
danvy@brics.dk

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 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.

1 Introduction

Forty years ago, Peter Landin wrote a profoundly influential article, “The Mechanical Evaluation of Expressions” [27], 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

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

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 [16]. 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. Since its inception, the SECD machine has been the starting point of many university courses and textbooks and 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 [1]. The derivation is based on a combination of simple, correct, and well-known program-transformation tools: CPS transformation [13, 41], delimited continuations [12], defunctionalization [14, 37], and closure conversion [27]. 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 [24].

1.1 Deconstruction of the SECD Machine

Let us outline our deconstruction of the SECD machine, before substantiating it in Section 2. 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 -> S
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 transition function for the SECD machine, all the observations above are in some sense unavoidable (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

$$\text{term} \rightarrow E \rightarrow \text{value} * E$$

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 [26] and of the spineless tagless G-machine [33].) 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 [12, 18].)

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 pure 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 [27], 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 stack-less 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.

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)

```

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`.

In the full version of this article [11], we analyze each of the transitions above; this analysis is however inessential for what follows.

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). *Given a source program, evaluate0 and evaluate1 either both diverge or yield expressible values that are structurally equal.*

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 [30], by Reynolds in his definitional interpreters [37], and by Steele and Sussman in their lambda-papers [40, 41, 42, 43]: `run_t` is eval and `run_a` is apply.

Proposition 2 (full correctness). *Given a source program, `evaluate1` and `evaluate2` either both diverge or yield expressible values that are structurally equal.*

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 [9]. 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). *Given a source program, evaluate2 and evaluate3 either both diverge or yield expressible values that are structurally equal.*

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 [12, 18]. To account for them, we use the control delimiter `reset`. Operationally, this control delimiter is moot here because no continuations are captured [12, 25]. 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 [20]. (Omitting `reset` leads to a dump-less variant of the SECD machine [11, Section 3.6].) 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). *Given a source program, evaluate3 and evaluate4 either both diverge or yield expressible values that are structurally equal.*

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). *Given a source program, evaluate4 and evaluate5 either both diverge or yield expressible values that are structurally equal.*

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 [38, 44, 45].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)

```

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

```

| 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)
| 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). *Given a source program, evaluate5 and evaluate6 either both diverge or yield expressible values that are related by closure conversion.*

The evaluator above is not unique, though. We can also choose a callee-save representation of functions, as developed in Section 2.7 of the full version of this article [11].

2.8 Assessment

Through a series of meaning-preserving steps, we have transformed the SECD machine (i.e., the transitive closure of a state-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, left-to-right call by value, and with one data stack for intermediate results and two continuations, the inner one for the current λ -abstraction.

3 Reconstructions of SECD-Like Machines

Each of the deconstruction steps of Section 2 is reversible. In the full version of this article [11], we review briefly how to rationally reconstruct a variety of SECD-like machines: the original SECD machine, a left-to-right SECD machine, a properly tail-recursive SECD machine, a call-by-name SECD machine, a call-by-need SECD machine, an SEC machine, an EC machine, an SC machine, a C machine, an SCD machine, and an SECD machine with an instruction set.

4 Related Work

In his famous 700 follow-up article [28,31], 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 [17, 19].)

In a similar way, in “Call-by-name, call-by-value, and the λ -calculus” [34], 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.

In his formalization of (a tail-recursive version of) the SECD machine [35], Ramsdell also observes that this machine uses callee-save convention.

5 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 a continuation; and one defunctionalizes this continuation to make the

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.)

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, 3, 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 [2]. 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, 3, 4, 7].

A first version of this article was written in the early fall of 2002. 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 article contains data-stack elimination and is a shorter version of a BRICS technical report [11]. This technical report 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 Kørsholm Rohde. The present shortened version has benefited from the sagacity of four anonymous IFL reviewers. 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. Last but not least, thanks are due to Peter Landin himself for his feedback and for the historical explanations he kindly provided me at Queen Mary in May 2004.

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

References

1. M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, Aug. 2003.
2. M. S. Ager, O. Danvy, and M. Goldberg. A symmetric approach to compilation and decompilation. In T. Æ. Mogensen, D. A. Schmidt, and I. H. 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.

3. M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-3.
4. M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 2005. Accepted for publication. Extended version available as the technical report BRICS RS-04-28.
5. A. Banerjee, N. Heintze, and J. G. Riecke. Design and correctness of program transformations based on control-flow analysis. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 420–447, Sendai, Japan, Oct. 2001. Springer-Verlag.
6. J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In M. Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press.
7. D. Biernacki and O. Danvy. From interpreter to logic engine by defunctionalization. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, Aug. 2003. Springer-Verlag.
8. H.-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994. ACM Press.
9. O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
10. O. Danvy. Type-directed partial evaluation. In G. L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, Jan. 1996. ACM Press.
11. O. 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, Oct. 2003.
12. O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
13. O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
14. O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.
15. O. Danvy and L. R. Nielsen. On one-pass CPS transformations. Technical Report BRICS RS-02-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Jan. 2002. Accepted for publication in the *Journal of Functional Programming*.
16. S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.
17. M. 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, Aug. 1987.

18. M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, Jan. 1988. ACM Press.
19. M. Felleisen and M. Flatt. Programming languages and lambda calculi. Unpublished lecture notes. <http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>, 1989-2003.
20. A. Filinski. Representing monads. In Boehm [8], pages 446–457.
21. D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.
22. M. Gasbichler and M. Sperber. Final shift for call/cc: direct implementation of shift and reset. In S. 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, Sept. 2002. ACM Press.
23. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In Boehm [8], pages 458–471.
24. P. Hein. *Grooks*. The MIT Press, 1966.
25. Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In O. Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, Aug. 2003. ACM Press.
26. J.-L. Krivine. Un interprète du λ -calcul. Brouillon. Available online at <http://www.pps.jussieu.fr/~krivine>, 1985.
27. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
28. P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.
29. J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In C. L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 227–238, Orlando, Florida, June 1994. ACM Press.
30. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, 1962.
31. L. Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.
32. L. R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Dec. 2000.
33. S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
34. G. D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
35. J. D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.
36. J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.
37. J. 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.

38. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
39. O. 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.
40. G. L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Nov. 1976.
41. G. 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.
42. G. L. Steele Jr. and G. J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Mar. 1976.
43. G. L. Steele Jr. and G. 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.
44. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
45. G. Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

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 [23]. The CPS transformation is abundantly described in the literature [15, 21, 36, 41].

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

$$\lambda f.\lambda g.\lambda x.\text{let } v_1 = f\ x\ \text{in let } v_2 = g\ x\ \text{in } v_1\ v_2$$

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 [12]. 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 [34, 37]. 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* [10, 12, 13, 20, 22, 25, 29]. 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 [39]. Defunctionalization is a transformation where function types are replaced by an enumeration of the function abstractions in this 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 [37]. 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, 32]. More detail can be found in Danvy and Nielsen's study [14]. The key observation here is that defunctionalizing a CPS program yields a transition function [1].

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.