
A RATIONAL DECONSTRUCTION OF LANDIN'S SECD MACHINE WITH THE J OPERATOR

OLIVIER DANVY^a AND KEVIN MILLIKIN^b

^a Department of Computer Science, Aarhus University
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
e-mail address: danvy@brics.dk

^b Google Inc.
Aabogade 15, DK-8200 Aarhus N, Denmark
e-mail address: kmillikin@google.com

ABSTRACT. Landin's SECD machine was the first abstract machine for applicative expressions, i.e., functional programs. Landin's J operator was the first control operator for functional languages, and was specified by an extension of the SECD machine. We present a family of evaluation functions corresponding to this extension of the SECD machine, using a series of elementary transformations (transformation into continuation-passing style (CPS) and defunctionalization, chiefly) and their left inverses (transformation into direct style and refunctionalization). To this end, we modernize the SECD machine into a bisimilar one that operates in lockstep with the original one but that (1) does not use a data stack and (2) uses the caller-save rather than the callee-save convention for environments. We also identify that the dump component of the SECD machine is managed in a callee-save way. The caller-save counterpart of the modernized SECD machine precisely corresponds to Thielecke's double-barrelled continuations and to Felleisen's encoding of J in terms of call/cc. We then variously characterize the J operator in terms of CPS and in terms of delimited-control operators in the CPS hierarchy.

As a byproduct, we also present several reduction semantics for applicative expressions with the J operator, based on Curien's original calculus of explicit substitutions. These reduction semantics mechanically correspond to the modernized versions of the SECD machine and to the best of our knowledge, they provide the first syntactic theories of applicative expressions with the J operator.

The present work is concluded by a motivated wish to see Landin's name added to the list of co-discoverers of continuations. Methodologically, however, it mainly illustrates the value of Reynolds's defunctionalization and of refunctionalization as well as the expressive power of the CPS hierarchy (1) to account for the first control operator and the first abstract machine for functional languages and (2) to connect them to their successors. Our work also illustrates the value of Danvy and Nielsen's refocusing technique to connect environment-based abstract machines and syntactic theories in the form of reduction semantics for calculi of explicit substitutions.

1998 ACM Subject Classification: D.1.1, F.3.2.

Key words and phrases: Abstract machines, continuations, control operators, reduction semantics.

1. INTRODUCTION

Forty years ago, Peter Landin unveiled the first control operator, J, to a heretofore unsuspecting world [81, 82, 84]. He did so to generalize the notion of jumps and labels for translating Algol 60 programs into applicative expressions, using the J operator to account for the meaning of an Algol label. For a simple example, consider the block

begin s_1 ; **goto** L ; L : s_2 **end**

where the sequencing between the statements (‘basic blocks,’ in compiler parlance [8]) s_1 and s_2 has been made explicit with a label and a jump to this label. This block is translated into the applicative expression

$$\lambda().\mathbf{let} L = \mathbf{J} s'_2 \mathbf{in let} () = s'_1 () \mathbf{in} L ()$$

where s'_1 and s'_2 respectively denote the translation of s_1 and s_2 . The occurrence of J captures the continuation of the outer let expression and yields a ‘program closure’ that is bound to L . Then, s'_1 is applied to $()$. If this application completes, the program closure bound to L is applied: (1) s'_2 is applied to $()$ and then, if this application completes, (2) the captured continuation is resumed, thereby completing the execution of the block.

Landin also showed that the notion of program closure makes sense not just in an imperative setting, but also in a functional one. He specified the J operator by extending the SECD machine [80, 83].

1.1. The SECD machine. Over the years, the SECD machine has been the topic of considerable study: it provides an unwavering support for operational semantics [1, 9, 11, 19, 25, 26, 53, 68, 69, 74, 95, 100, 117], compilation [5, 10, 21, 23, 62, 64, 65, 96, 98, 99, 106], and parallelism [2, 20], and it lends itself readily to variations [47, 48, 51, 67, 101, 105, 116] and generalizations [87, 88, 120]. In short, it is standard textbook material [55, 63, 70, 71, 79, 109], even though its architecture is generally agreed to be on the ‘baroque’ side, since most subsequent abstract machines have no data stack and only one control stack instead of two. Nobody, however, seems to question its existence as a distinct artifact (i.e., man-made construct) mediating between applicative expressions (i.e., functional programs) and traditional sequential imperative implementations.

Indeed abstract machines provide a natural meeting ground for theoretically-minded and experimentally-minded computer scientists: they are as close to an actual implementation as most theoreticians will ever get, and to an actual formalization as most experimentalists will ever go. For example, Plotkin [100] proved the correctness of the SECD machine in reference to a definitional interpreter due to Morris [91] and a variety of implementations take the SECD machine as their starting point [10, 20, 23, 87].

1.2. The authors’ thesis. Is there, however, such a gap between applicative expressions and abstract machines? The thrust of Steele’s MSc thesis [110] was that after CPS transformation,¹ a λ -abstraction can be seen as a label and a tail call as a machine jump with the

¹CPS’ stands for ‘Continuation-Passing Style;’ this term is due to Steele [110]. In a CPS program, all calls are tail calls and functions thread a functional accumulator, the continuation, that represents ‘the rest of the computation’ [114]. CPS programs are either written directly or the result of a CPS transformation [39, 100]. (See Appendix A.2.) The left inverse of the CPS transformation is the direct-style transformation [33, 41].

machine registers holding the actual parameters. Furthermore the point of Reynolds's defunctionalization [102] is that higher-order programs can be given an equivalent first-order representation.²

It nevertheless took 40 years for the SECD machine to be 'rationally deconstructed' into a compositional evaluation function [35], the point being that

- (1) the SECD machine is essentially in defunctionalized form, and
- (2) its refunctionalized counterpart is an evaluation function in CPS, which turns out to be compositional.

This deconstruction laid the ground for a functional correspondence between evaluators and abstract machines [3, 4, 6, 7, 13, 16, 36, 37, 73, 89, 90, 93].

It is therefore the authors' thesis [36, 90] that the gap between abstract machines and applicative expressions is bridged by Reynolds's defunctionalization.

Our goal here is to show that the functional correspondence between evaluators and abstract machines also applies to the SECD machine with the J operator, which, as we show, also can be deconstructed into a compositional evaluation function. As a corollary, we present several new simulations of the J operator, and the first syntactic theories for applicative expressions with the J operator.

1.3. Deconstruction of the SECD machine with the J operator. Let us outline our deconstruction of the SECD machine before substantiating it in the next sections. We follow the order of the first deconstruction [35], though with a twist: for simplicity and without loss of generality, in the middle of the derivation, we first abandon the stack-threading, callee-save features of the SECD machine, which are non-standard, for the more familiar—and therefore less 'baroque'—stackless, caller-save features of traditional definitional interpreters [59, 91, 102, 111]. (These concepts are reviewed in the appendices. The point here is that the SECD machine manages the environment in a callee-save fashion.) We then identify that the dump too is managed in a callee-save fashion and we present the corresponding caller-save counterpart.

The SECD machine is defined as the iteration of a state-transition function operating over a quadruple—a data stack (of type *S*) containing intermediate values, an environment (of type *E*), a control stack (of type *C*), and a dump (of type *D*) and yielding a value (of type *value*):

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

The first deconstruction [35] showed that together the *C* and *D* components represent the current continuation and that the *D* component represents the continuation of the current caller, if there is one. As already pointed out in Section 1.1, since Landin's work, the *C* and *D* components of his abstract machine have been unified into one component; reflecting this unification, control operators capture both what used to be *C* and *D* instead of only what used to be *D*.

²In the early 1970's [102], John Reynolds introduced defunctionalization as a variation of Landin's 'function closures' [80], where a term is paired together with its environment. In a defunctionalized program, what is paired with an environment is not a term, but a tag that determines this term uniquely. In ML, the tagged environments are grouped into data types, and auxiliary apply functions dispatch over the tags. (See Appendix A.3.) The left inverse of defunctionalization is 'refunctionalization' [43, 44].

1.3.1. *Disentangling and refunctionalization (Section 2)*. The above definition of `run` looks complicated because it has several induction variables, i.e., it dispatches over several components of the quadruple. Our deconstruction proceeds as follows:

- We disentangle `run` into four mutually recursive transition functions, each of which has one induction variable, i.e., dispatches over one component of the quadruple (boxed in the signature below):

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

The first function, `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. This disentangled specification, as it were, is in defunctionalized form [43, 44, 102]: the control stack and the dump are defunctionalized data types, and `run_c` and `run_d` are the corresponding apply functions.

- Refunctionalization eliminates the two apply functions:

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

`C` and `D` are now function types. As identified in the first rational deconstruction [35], the resulting program is a continuation-passing interpreter. This interpreter threads a data stack to hold intermediate results and uses a callee-save convention for environments to process subterms. (For information and comparison, Appendix B illustrates an interpreter with no data stack for intermediate results and a caller-save convention for environments, Appendix C illustrates an interpreter with no data stack for intermediate results and a callee-save convention for environments, and Appendix D illustrates an interpreter with a data stack for intermediate results and a caller-save convention for environments.)

At this point, we could continue as in the first deconstruction [35] and exhibit the direct-style counterpart of this interpreter. The result, however, would be less simple and less telling than first making do without the data stack (Section 1.3.2) and second adopting the more familiar caller-save convention for environments (Section 1.3.3) before continuing the deconstruction towards a compositional interpreter in direct style (Section 1.3.4).

1.3.2. *A first modernization: eliminating the data stack (Section 3)*. In order to focus on the nature of the `J` operator, we first eliminate the data stack:

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

(Two simpler interpreters are presented and contrasted in Appendices B and D. The first, in Appendix B, has no data stack for intermediate results, and the second, in Appendix D, has one.)

1.3.3. A second modernization: from callee-save to caller-save environments (Section 3).

Again, in order to focus on the nature of the J operator, we adopt the more familiar caller-save convention for environments. In passing, we also rename `run_t` as `eval` and `run_a` as `apply`:

```
eval :      term * E * C * D -> value
apply : value * value * C * D -> value
where C = value * D -> value and D = value -> value
```

(Two simpler interpreters are presented and contrasted in Appendices B and C. The first, in Appendix B, uses a caller-save convention for environments, and the second, in Appendix C, uses a callee-save convention.)

1.3.4. Continuing the deconstruction: towards a compositional interpreter in direct style.

- A direct-style transformation eliminates the dump continuation:

```
eval :      term * E * C -> value
apply : value * value * C -> value
where C = value -> value
```

The clause for the J operator and the main evaluation function are expressed using the delimited-control operators `shift` and `reset` [38].³ The resulting interpreter still threads an explicit continuation, even though it is not tail-recursive.

- Another direct-style transformation eliminates the control continuation:

```
eval :      term * E -> value
apply : value * value -> value
```

The clauses catering for the non-tail-recursive uses of the control continuation are expressed using the delimited-control operators `shift1`, `reset1`, `shift2`, and `reset2` [13, 38, 46, 75, 94]. The resulting evaluator is in direct style. It is also in closure-converted form: the applicable values are a defunctionalized data type and `apply` is the corresponding apply function.

- Refunctionalization eliminates the apply function:

```
eval : term * E -> value
```

The resulting evaluation function is compositional, and the corresponding syntax-directed encoding gives rise to new simulations of the J operator.

1.3.5. A variant: from callee-save to caller-save dumps (Section 4). In Section 1.3.3, we kept the dump component because it is part of the SECD machine semantics of the J operator. We observe, however, that the dump is managed in a callee-save way. We therefore change gear and consider the caller-save counterpart of the interpreter:

```
eval :      term * E * C * D -> value
apply : value * value * C      -> value
where C = value -> value and D = value -> value
```

³ Delimited continuations represent part of the rest of the computation: the control operator `reset` delimits control and the control operator `shift` captures the current delimited continuation [38]. These two control operators provide a direct-style handle for programs with two layers of continuations. This programming pattern is also used for ‘success’ and ‘failure’ continuations in the functional-programming approach to backtracking. Programs that have been CPS-transformed twice exhibit two such layers of continuations. Here, C is the first layer and D is the second. Iterating a CPS transformation gives rise to a CPS hierarchy [13, 38, 76, 94].

This caller-save interpreter is still in CPS. We can write its direct-style counterpart and refunctionalize its applicable values, which yields another compositional evaluation function in direct style. This compositional evaluation function gives rise to new simulations of the J operator, some of which had already been invented independently.

1.3.6. *Assessment.* As illustrated in Sections 1.3.2, 1.3.3, and 1.3.5, there is plenty of room for variation in the present deconstruction. Each step is reversible: one can CPS-transform and defunctionalize an evaluator and (re)construct an abstract machine [3, 4, 6, 7, 13, 16, 35–37].

1.4. **Syntactic theories of applicative expressions with the J operator.** Let us outline our syntactic theories of applicative expressions substantiating them in the next sections.

1.4.1. *Explicit, callee-save dumps (Section 7).* We present a reduction semantics for Curien’s calculus of closures extended with the J operator, and we derivationally link it to the caller-save, stackless SECD machine of Section 7.1.

1.4.2. *Implicit, caller-save dumps (Section 8).* We present another reduction semantics for Curien’s calculus of closures extended with the J operator, and we derivationally link it to a version of the SECD machine which is not in defunctionalized form.

1.4.3. *Explicit, caller-save dumps (Section 9).* We outline a third reduction semantics for Curien’s calculus of closures extended with the J operator, and we show how it leads towards Thielecke’s double-barrelled continuations.

1.4.4. *Inheriting the dump through the environment (Section 10).* We present a fourth reduction semantics for Curien’s calculus of closures extended with the J operator, and we derivationally link it to a version of the CEK machine that reflects Felleisen’s simulation of the J operator.

1.5. **Prerequisites and domain of discourse: the functional correspondence.** We mostly use pure ML as a meta-language. We assume a basic familiarity with Standard ML and with reasoning about pure ML programs as well as an elementary understanding of defunctionalization [43, 44, 102] and its left inverse, refunctionalization; of the CPS transformation [38, 41, 59, 91, 102, 110] and its left inverse, the direct-style transformation; and of delimited continuations [13, 38, 46, 56, 75]. From Section 3.2, we use pure ML with delimited-control operators as a meta-language.

The source language of the SECD machine. The source language is the λ -calculus, extended with literals (as observables) and the J operator. Except for the variables in the initial environment of the SECD machine, a program is a closed term:

```
datatype term = LIT of int
              | VAR of string
              | LAM of string * term
              | APP of term * term
              | J
type program = term
```

The control directives. The control component of the SECD machine is a list of control directives, where a directive is a term or the tag APPLY:

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

The environment. We use a structure `Env` with the following signature:

```
signature ENV = sig
  type 'a env
  val empty : 'a env
  val extend : string * 'a * 'a env -> 'a env
  val lookup : string * '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`. These functions are pure and total and therefore throughout, we call them without passing them any continuation, i.e., in direct style [40].

Values. There are five kinds of values: integers, the successor function, function closures, “state appenders” [21, page 84], and program closures:

```
datatype value = INT of int
              | SUCC
              | FUNCLO of E * string * term
              | STATE_APPENDER of D
              | PGMCLO of value * D

withtype S = value list (* data stack *)
and E = value Env.env (* environment *)
and C = directive list (* control *)
and D = (S * E * C) list (* dump *)
```

A function closure pairs a λ -abstraction (i.e., its formal parameter and its body) and its lexical environment. A state appender is an intermediate value; applying it yields a program closure. A program closure is a first-class continuation.⁴

The initial environment. The initial environment binds the successor function:

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

⁴The terms ‘function closures’ and ‘program closures’ are due to Landin [82]. The term ‘state appender’ is due to Burge [21]. The term ‘continuation’ is due to Wadsworth [118]. The term ‘first-class’ is due to Strachey [113]. The term ‘first-class continuation’ is due to Friedman and Haynes [58].

The starting specification: Several formulations of the SECD machine with the J operator have been published [21,51,82]. We take the most recent one, i.e., Felleisen’s [51], as our starting point, and we consider the others in Section 5:

```
(* run : S * E * C * D -> value *)
fun run (v :: s, e, nil, nil)
  = v
  | run (v :: s', e', nil, (s, e, c) :: d)
    = run (v :: s, e, c, d)
  | run (s, e, (TERM (LIT n)) :: c, d)
    = run ((INT n) :: s, e, c, d)
  | run (s, e, (TERM (VAR x)) :: c, d)
    = run ((Env.lookup (x, e)) :: s, e, c, d)
  | run (s, e, (TERM (LAM (x, t))) :: c, d)
    = run ((FUNCLO (e, x, t)) :: s, e, c, d)
  | run (s, e, (TERM (APP (t0, t1))) :: c, d)
    = run (s, e, (TERM t1) :: (TERM t0) :: APPLY :: c, d)
  | run (s, e, (TERM J) :: c, d) *)
    = run ((STATE_APPENDER d) :: s, e, c, d)
  | run (SUCC :: (INT n) :: s, e, APPLY :: c, d)
    = run ((INT (n+1)) :: s, e, c, d)
  | run ((FUNCLO (e', x, t)) :: v :: s, e, APPLY :: c, d)
    = run (nil, Env.extend (x, v, e'), (TERM t) :: nil, (s, e, c) :: d)
  | run ((STATE_APPENDER d') :: v :: s, e, APPLY :: c, d) *)
    = run ((PGMCLO (v, d')) :: s, e, c, d)
  | run ((PGMCLO (v, d')) :: v' :: s, e, APPLY :: c, d) *)
    = run (v :: v' :: nil, e_init, APPLY :: nil, d')

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

The function `run` implements the iteration of a transition function for the SECD machine: (s, e, c, d) is a state of the machine and each clause of the definition of `run` specifies a state transition.

The SECD machine is deterministic. It terminates if it reaches a state with an empty control stack and an empty dump; in that case, it produces a value on top of the data stack. It does not terminate for divergent source terms. It becomes stuck if it attempts to apply an integer or attempts to apply the successor function to a non-integer value, in that case an ML pattern-matching error is raised (alternatively, the codomain of `run` could be made `value option` and a fallthrough `else` clause could be added). The clause marked “1” specifies that the J operator, at any point, denotes the current dump; evaluating it captures this dump and yields a state appender that, when applied (in the clause marked “2”), yields a program closure. Applying a program closure (in the clause marked “3”) restores the captured dump.

1.6. Prerequisites and domain of discourse: the syntactic correspondence. We assume a basic familiarity with reduction semantics as can be gathered in Felleisen’s PhD thesis [50] and undergraduate lecture notes [52] and with Curien’s original calculus of closures [14,31], which is the ancestor of calculi of explicit substitutions. We also review the syntactic correspondence between reduction semantics and abstract machines in Section E by deriving the CEK machine from Curien’s calculus of closures for applicative order.

1.7. Overview. We first disentangle and refunctionalize Felleisen's version of the SECD machine (Section 2). We then modernize it, eliminating its data stack and making go from callee-save to caller-save environments, and deconstruct the resulting specification into a compositional evaluator in direct style; we then analyze the J operator (Section 3). Identifying that dumps are managed in a callee-save way in the modernized SECD machine, we also present a variant where they are managed in a caller-save way, and we deconstruct the resulting specification into another compositional evaluator in direct style; we then analyze the J operator (Section 4). Overall, the deconstruction takes the form of a series of elementary transformations. The correctness of each step is very simple: most of the time, it is a corollary of the correctness of the transformation itself.

We then review related work (Section 5) and outline the deconstruction of the original version of the SECD machine, which is due to Burge (Section 6).

We then present a reduction semantics for the J operator that corresponds to the specification of Section 3 (Section 7). We further present a syntactic theory of applicative expressions with the J operator using delimiters (Section 8), and we show how this syntactic theory specializes to a reduction semantics that yields the abstract machine of Section 4 (Section 9) and to another reduction semantics that embodies Felleisen's embedding of J into Scheme described in Section 4.5 (Section 10).

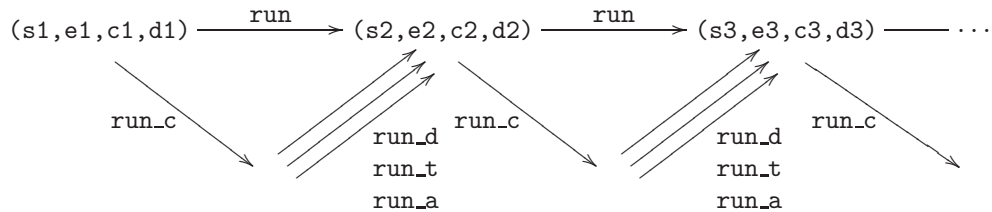
We then conclude (Sections 11 and 12).

2. DECONSTRUCTION OF THE SECD MACHINE WITH THE J OPERATOR:
 DISENTANGLING AND REFUNCTIONALIZATION

2.1. A disentangled specification. In the starting specification of Section 1.5, all the possible transitions are meshed together in one recursive function, `run`. As in the first rational deconstruction [35], we factor `run` into four mutually recursive functions, each with one induction variable. In this disentangled definition, `run_c` dispatches to the three other transition functions, which all dispatch back to `run_c`:

- `run_c` interprets the list of control directives, i.e., it specifies which transition to take according to whether 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 according to whether the dump is empty or non-empty, given a valid data stack; `run_t` interprets the top term in the list of control directives; and `run_a` interprets the top value in the data stack.

Graphically:



```

(* run_c :          S * E * C * D -> value          *)
(* run_d :          value * D -> value              *)
(* run_t :          term * S * E * C * D -> value   *)
(* run_a : value * value * S * E * C * D -> value   *)
fun run_c (v :: s, e, nil, d)
  = run_d (v, d)
  | run_c (s, e, (TERM t) :: c, d)
    = run_t (t, s, e, c, d)
  | run_c (v0 :: v1 :: s, e, APPLY :: c, d)
    = run_a (v0, v1, s, e, c, d)
and run_d (v, nil)
  = v
  | run_d (v, (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 ((FUNCLO (e, x, t)) :: s, e, c, d)
  | run_t (APP (t0, t1), s, e, c, d)
    = run_c (s, e, (TERM t1) :: (TERM t0) :: APPLY :: c, d)
  | run_t (J, s, e, c, d)
    = run_c ((STATE_APPENDER d) :: s, e, c, d)
and run_a (SUCC, INT n, s, e, c, d)
  = run_c ((INT (n+1)) :: s, e, c, d)
  | run_a (FUNCLO (e', x, t), v, s, e, c, d)
    = run_c (nil, Env.extend (x, v, e'), (TERM t) :: nil, (s, e, c) :: d)
  | run_a (STATE_APPENDER d', v, s, e, c, d)
    = run_c ((PGMCLO (v, d')) :: s, e, c, d)
  | run_a (PGMCLO (v, d'), v', s, e, c, d)
    = run_c (v :: v' :: nil, e_init, APPLY :: nil, d')
fun evaluate1 t (* evaluate1 : program -> value *)
  = run_c (nil, e_init, (TERM t) :: nil, nil)

```

By construction, the two machines operate in lockstep, with each transition of the original machine corresponding to two transitions of the disentangled machine. Since the two machines start in the same initial state, the correctness of the disentangled machine is a corollary of them operating in lockstep:

Proposition 2.1 (full correctness). *Given a program, evaluate0 and evaluate1 either both diverge or both yield values that are structurally equal.*

In the rest of this section, we only consider programs that yield an integer value, if any. Indeed we are going to modify the data type of the values as we go from abstract machine to evaluator, and we want a simple, comparable characterization of the results they yield.

Furthermore, again for simplicity, we short-circuit four state transitions in the abstract machine above:

```

...
  | run_t (APP (t0, t1), s, e, c, d)
    = run_t (t1, s, e, (TERM t0) :: APPLY :: c, d)

```

```

...
| run_a (FUNCLO (e', x, t), v, s, e, c, d)
  = run_t (t, nil, Env.extend (x, v, e'), nil, (s, e, c) :: d)
...
| run_a (PGMCLO (v, d'), v', s, e, c, d)
  = run_a (v, v', nil, e_init, nil, d')
...
fun evaluate1 t
  = run_t (t, nil, e_init, nil, nil)

```

2.2. A higher-order counterpart. In the disentangled definition of Section 2.1, there are two possible ways to construct a dump—nil and consing a triple—and three possible ways to construct a list of control directives—nil, consing a term, and consing an apply directive. One could phrase these constructions as two specialized data types rather than as two lists.

These data types, together with `run_d` and `run_c` as their apply functions, are in the image of defunctionalization. After refunctionalization, the higher-order evaluator reads as follows;⁵ it is higher-order because `c` and `d` now denote functions:

```

datatype value = INT of int
                | SUCC
                | FUNCLO of E * string * term
                | STATE_APPENDER of D
                | PGMCLO of value * D

withtype S = value list           (* data stack *)
       and E = value Env.env      (* environment *)
       and D = value -> value     (* dump continuation *)
       and C = S * E * D -> value (* control continuation *)

val e_init = Env.extend ("succ", SUCC, Env.empty)
(* run_t :      term * S * E * C * D -> value *)
(* run_a : value * value * S * E * C * D -> 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 ((FUNCLO (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 (v0 :: v1 :: s, e, d) =>
                    run_a (v0, v1, s, e, c, d), d), d)
  | run_t (J, s, e, c, d)
  = c ((STATE_APPENDER d) :: s, e, d)
and run_a (SUCC, INT n, s, e, c, d)
  = c ((INT (n+1)) :: s, e, d)
  | run_a (FUNCLO (e', x, t), v, s, e, c, d)
  = run_t (t, nil, Env.extend (x, v, e'), fn (v :: s, e, d) => d v,
          fn v => c (v :: s, e, d))

```

⁵Had we not short-circuited the four state transitions at the end of Section 2.1, the resulting higher-order evaluator would contain four β_v -redexes. Contracting these redexes corresponds to short-circuiting these transitions.

```

| run_a (STATE_APPENDER d', v, s, e, c, d)
  = c ((PGMCLO (v, d')) :: s, e, d)
| run_a (PGMCLO (v, d'), v', s, e, c, d)
  = run_a (v, v', nil, e_init, fn (v :: s, e, d) => d v, d')
fun evaluate2 t                                (* evaluate2 : program -> value *)
  = run_t (t, nil, e_init, fn (v :: s, e, d) => d v, fn v => v)

```

The resulting evaluator is in CPS, with two layered continuations `c` and `d`. It threads a stack of intermediate results (`s`), an environment (`e`), a control continuation (`c`), and a dump continuation (`d`). Except for the environment being callee-save, the evaluator follows a traditional eval-apply schema: `run_t` is eval and `run_a` is apply. Defunctionalizing it yields the definition of Section 2.1 and as illustrated in Appendix A, by construction, `run_t` and `run_a` in the defunctionalized version operate in lockstep with `run_t` and `run_a` in the refunctionalized version:

Proposition 2.2 (full correctness). *Given a program, evaluate1 and evaluate2 either both diverge or both yield values; and if these values have an integer type, they are the same integer.*

3. DECONSTRUCTION OF THE SECD MACHINE WITH THE J OPERATOR: NO DATA STACK AND CALLER-SAVE ENVIRONMENTS

We want to focus on `J`, and the non-standard aspects of the evaluator of Section 2.2 (the callee-save environment and the data stack) are a distraction. We therefore modernize this evaluator into a more familiar caller-save, stackless form [59,91,102,111]. Let us describe this modernization in two steps: first we transform the evaluator to use a caller-save convention for environments (as outlined in Section 1.3.2 and illustrated in Appendices B and C), and second we transform it to not use a data stack (as outlined in Section 1.3.3 and illustrated in Appendices B and D).

The environments of the evaluator of Section 2.2 are callee-save because the apply function `run_a` receives an environment `e` as an argument and “returns” one to its continuation `c` [8, pages 404–408]. Inspecting the evaluator shows that whenever `run_a` is passed a control directive `c` and an environment `e` and applies `c`, then the environment `e` is passed to `c`. Thus, the environment is passed to `run_a` only in order to thread it to the control continuation. The control continuations created in `run_a` and `evaluate2` ignore their environment argument, and the control continuations created in `run_t` are passed an environment that is already in their lexical scope. Therefore, neither the apply function `run_a` nor the control continuations need to be passed an environment at all.

Turning to the data stack, we first observe that the control continuations of the evaluator in Section 2.2 are always applied to a data stack with at least one element. Therefore, we can pass the top element of the data stack as a separate argument, changing the type of control continuations from $S * E * D \rightarrow \text{value}$ to $\text{value} * S * E * D \rightarrow \text{value}$. We can thus eliminate the data stack following an argument similar to the one for environments in the previous paragraph: the `run_a` function merely threads its data stack along to its control continuation; the control continuations created in `run_a` and `evaluate2` ignore their data-stack argument, and the control continuations created in `run_t` are passed a data stack that is already in their lexical scope. Therefore, neither the apply function `run_a`, the eval function `run_t`, nor the control continuations need to be passed a data stack at all.

3.1. A specification with no data stack and caller-save environments. The caller-save, stackless counterpart of the evaluator of Section 2.2 reads as follows, renaming `run_t` as `eval` and `run_a` as `apply` in passing:

```

datatype value = INT of int
                | SUCC
                | FUNCLO of E * string * term
                | STATE_APPENDER of D
                | PGMCLO of value * D

withtype E = value Env.env (* environment *)
         and D = value -> value (* dump continuation *)
         and C = value * D -> value (* control continuation *)

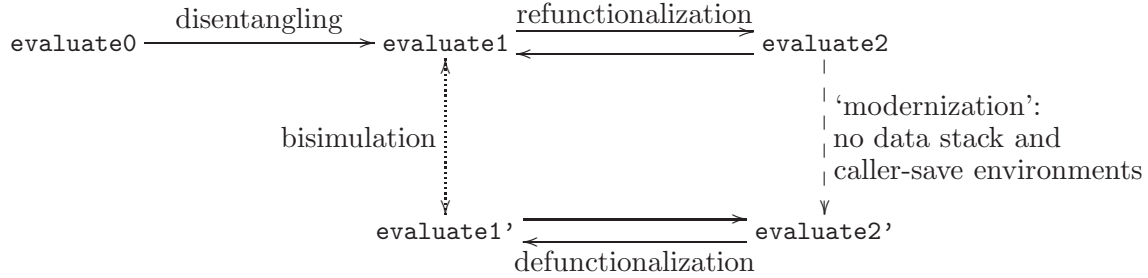
val e_init = Env.extend ("succ", SUCC, Env.empty)

(* eval : term * E * C * D -> value *)
(* apply : value * value * C * D -> value *)
fun eval (LIT n, e, c, d)
  = c (INT n, d)
  | eval (VAR x, e, c, d)
  = c (Env.lookup (x, e), d)
  | eval (LAM (x, t), e, c, d)
  = c (FUNCLO (e, x, t), d)
  | eval (APP (t0, t1), e, c, d)
  = eval (t1, e, fn (v1, d) =>
          eval (t0, e, fn (v0, d) =>
              apply (v0, v1, c, d), d), d)
  | eval (J, e, c, d)
  = c (STATE_APPENDER d, d)
and apply (SUCC, INT n, c, d)
  = c (INT (n+1), d)
  | apply (FUNCLO (e', x, t), v, c, d)
  = eval (t, Env.extend (x, v, e'), fn (v, d) => d v,
          fn v => c (v, d))
  | apply (STATE_APPENDER d', v, c, d)
  = c (PGMCLO (v, d'), d)
  | apply (PGMCLO (v, d'), v', c, d)
  = apply (v, v', fn (v, d) => d v, d')

fun evaluate2' t (* evaluate2' : program -> value *)
  = eval (t, e_init, fn (v, d) => d v, fn v => v)

```

The new evaluator is still in CPS, with two layered continuations. In order to justify it formally, we consider the corresponding abstract machine as obtained by defunctionalization (shown in Section 7; the ML code for `evaluate1'` is not shown here). This abstract machine and the disentangled abstract machine of Section 2.1 operate in lockstep and we establish a bisimulation between them. The full details of this formal justification are found in the second author's PhD dissertation [90, Section 4.4]. Graphically:



The following proposition follows as a corollary of the bisimulation and of the correctness of defunctionalization:

Proposition 3.1 (full correctness). *Given a program, evaluate2 and evaluate2' either both diverge or both yield values; and if these values have an integer type, they are the same integer.*

3.2. A dump-less direct-style counterpart. The evaluator of Section 3.1 is in continuation-passing style, and therefore it is in the image of the CPS transformation. In order to highlight the control effect of the J operator, we now present the direct-style counterpart of this evaluator.

The clause for J captures the current continuation (i.e., the dump) in a state appender, and therefore its direct-style counterpart naturally uses the undelimited control operator call/cc [41]. With an eye on our next step, we do not, however, use call/cc but its delimited cousins shift and reset [13, 38, 46] to write the direct-style counterpart.

Concretely, we use an ML functor to obtain an instance of shift and reset with value as the type of intermediate answers [46, 56]: reset delimits the (now implicit) dump continuation in eval, and corresponds to its initialization with the identity function; and shift captures it in the clauses where J is evaluated and where a program closure is applied. There is one non-tail call to eval, to evaluate the body of a λ -abstraction; this context is captured by shift when J is evaluated:

```

datatype value = INT of int
                | SUCC
                | FUNCLO of E * string * term
                | STATE_APPENDER of D
                | PGMCLO of value * D

withtype E = value Env.env (* environment *)
and C = value -> value (* control continuation *)
and D = value -> value (* first-class dump continuation *)

val e_init = Env.extend ("succ", SUCC, Env.empty)
structure SR = make_Shift_and_Reset (type intermediate_answer = value)
(* eval : term * E * C -> value *)
(* apply : value * value * C -> value *)
fun eval (LIT n, e, c)
  = c (INT n)
  | eval (VAR x, e, c)
  = c (Env.lookup (x, e))

```

```

| eval (LAM (x, t), e, c)
  = c (FUNCLO (e, x, t))
| eval (APP (t0, t1), e, c)
  = eval (t1, e, fn v1 => eval (t0, e, fn v0 => apply (v0, v1, c)))
| eval (J, e, c)
  = SR.shift (fn d => d (c (STATE_APPENDER d))) (* * *)
and apply (SUCC, INT n, c)
  = c (INT (n+1))
| apply (FUNCLO (e', x, t), v, c)
  = c (eval (t, Env.extend (x, v, e'), fn v => v)) (* * *)
| apply (STATE_APPENDER d, v, c)
  = c (PGMCLO (v, d))
| apply (PGMCLO (v, d), v', c)
  = SR.shift (fn d' => d (apply (v, v', fn v => v))) (* * *)
fun evaluate3' t (* evaluate3' : program -> value *)
  = SR.reset (fn () => eval (t, e_init, fn v => v))

```

The dump continuation is now implicit and is accessed using `shift`. The first occurrence of `shift` captures the current dump when `J` is evaluated. The second occurrence is used to discard the current dump when a program closure is applied. CPS-transforming this evaluator yields the evaluator of Section 3.1:

Proposition 3.2 (full correctness). *Given a program, `evaluate2'` and `evaluate3'` either both diverge or both yield values; and if these values have an integer type, they are the same integer.*

3.3. A control-less direct-style counterpart. The evaluator of Section 3.2 still threads an explicit continuation, the control continuation. It however is not in continuation-passing style because of the non-tail calls to `c`, `eval`, and `apply` (in the clauses marked “*” above) and the occurrences of `shift` and `reset`. This pattern of control is characteristic of the CPS hierarchy [13, 38, 46, 75] (see also Footnote 3, page 5). We therefore use the delimited-control operators `shift1`, `reset1`, `shift2`, and `reset2` to write the direct-style counterpart of this evaluator (`shift2` and `reset2` are the direct-style counterparts of `shift1` and `reset1`, and `shift1` and `reset1` are synonyms for `shift` and `reset`).

Concretely, we use two ML functors to obtain layered instances of `shift` and `reset` with `value` as the type of intermediate answers [46, 56]: `reset2` delimits the (now twice implicit) dump continuation in `eval`; `shift2` captures it in the clauses where `J` is evaluated and where a program closure is applied; `reset1` delimits the (now implicit) control continuation in `eval` and in `apply`, and corresponds to its initialization with the identity function; and `shift1` captures it in the clause where `J` is evaluated:

```

datatype value = INT of int
                | SUCC
                | FUNCLO of E * string * term
                | STATE_APPENDER of D
                | PGMCLO of value * D

withtype E = value Env.env (* environment *)
and D = value -> value (* first-class dump continuation *)
val e_init = Env.extend ("succ", SUCC, Env.empty)
structure SR1 = make_Shift_and_Reset (type intermediate_answer = value)

```



```

(* eval : term * E -> value *)
(* where E = value Env.env *)
fun eval (LIT n, e)
  = INT n
| eval (VAR x, e)
  = Env.lookup (x, e)
| eval (LAM (x, t), e)
  = FUN (fn v => SR1.reset (fn () => eval (t, Env.extend (x, v, e))))
| eval (APP (t0, t1), e)
  = let val v1 = eval (t1, e)
        val (FUN f) = eval (t0, e)
      in f v1 end
| eval (J, e)
  = SR1.shift (fn c => SR2.shift (fn d =>
    d (c (FUN (fn v =>
      FUN (fn v' => SR1.shift (fn c' =>
        SR2.shift (fn d' =>
          d (SR1.reset (fn () => let val (FUN f) = v
                                in f v' end))))))))))

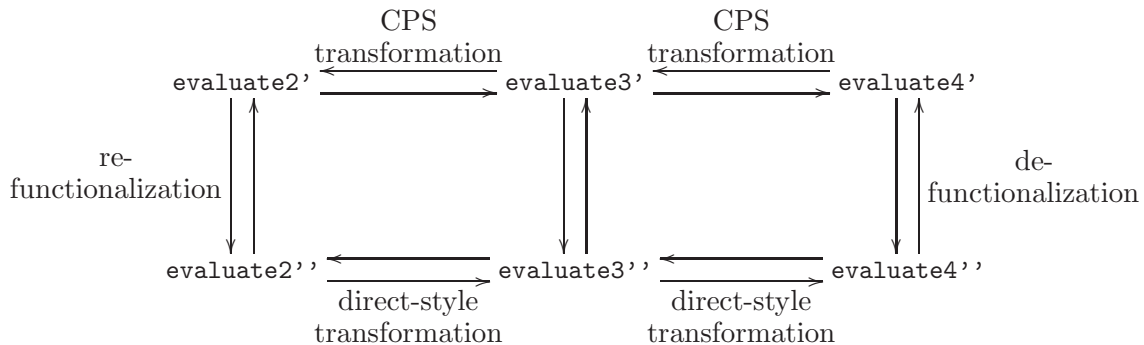
fun evaluate4'' t (* evaluate4'' : program -> value *)
  = SR2.reset (fn () => SR1.reset (fn () => eval (t, e_init)))

```

Unlike all the abstract machines and evaluators before, this evaluation function is compositional: all the recursive calls on the right-hand side are over proper sub-parts of the corresponding expression on the left-hand side. Defunctionalizing this evaluation function yields the evaluator of Section 3.3:

Proposition 3.4 (full correctness). *Given a program, evaluate4' and evaluate4'' either both diverge or both yield values; and if these values have an integer type, they are the same integer.*

3.5. Assessment. From Section 3.1 to Section 3.4, we have modernized the SECD machine into a stackless machine with a caller-save convention for environments, and then deconstructed the modernized version of this machine into a series of equivalent specifications, starting (essentially) from a relation between states and ending with an evaluation function. The diagram below graphically summarizes the deconstruction. The evaluators in the top row are the defunctionalized counterparts of the evaluators in the bottom row. (The ML code for evaluate2'' and evaluate3'' is not shown here.)



Using the tracing technique of Appendix A, we can show that `evaluate2'` and `evaluate2''` operate in lockstep. We have however not proved this lockstep property for `evaluate3'` and `evaluate3''` and for `evaluate4'` and `evaluate4''`, satisfying ourselves with Plotkin's Simulation theorem [100], suitably extended for shift and reset [76, 77].

3.6. On the J operator. We now reap the fruits of the modernization and the reconstruction, and present a series of simulations of the J operator (Sections 3.6.1, 3.6.2, and 3.6.3). We then put the J operator into perspective (Section 3.6.4).

3.6.1. Three simulations of the J operator. The evaluator of Section 3.4 (`evaluate4''`) and the refunctionalized counterparts of the evaluators of Sections 3.2 and 3.1 (`evaluate3''` and `evaluate2''`) are compositional. They can be viewed as syntax-directed encodings into their meta-language, as embodied in the first Futamura projection [60] and the original approach to denotational semantics [112]. Below, we state these encodings as three simulations of J: one in direct style, one in CPS with one layer of continuations, and one in CPS with two layers of continuations.

We assume a call-by-value meta-language with right-to-left evaluation.

- In direct style, using shift_2 (\mathcal{S}_2), reset_2 ($\langle \cdot \rangle_2$), shift_1 (\mathcal{S}_1), and reset_1 ($\langle \cdot \rangle_1$), based on the compositional evaluator `evaluate4''` in direct style:

$$\begin{aligned} \llbracket n \rrbracket &= n \\ \llbracket x \rrbracket &= x \\ \llbracket t_0 t_1 \rrbracket &= \llbracket t_0 \rrbracket \llbracket t_1 \rrbracket \\ \llbracket \lambda x.t \rrbracket &= \lambda x. \langle \llbracket t \rrbracket \rangle_1 \\ \llbracket \mathbf{J} \rrbracket &= \mathcal{S}_1 \lambda c. \mathcal{S}_2 \lambda d. d (c \lambda v. \boxed{\lambda v'. \mathcal{S}_1 \lambda c'. \mathcal{S}_2 \lambda d'. d \langle v v' \rangle_1}) \end{aligned}$$

A program p is translated as $\langle \langle \llbracket p \rrbracket \rangle_1 \rangle_2$.

- In CPS with one layer of continuations, using shift (\mathcal{S}) and reset ($\langle \cdot \rangle$), based on the compositional evaluator `evaluate3''` in CPS with one layer of continuations:

$$\begin{aligned} \llbracket n \rrbracket' &= \lambda c. c n \\ \llbracket x \rrbracket' &= \lambda c. c x \\ \llbracket t_0 t_1 \rrbracket' &= \lambda c. \llbracket t_1 \rrbracket' \lambda v_1. \llbracket t_0 \rrbracket' \lambda v_0. v_0 v_1 c \\ \llbracket \lambda x.t \rrbracket' &= \lambda c. c \lambda x. \lambda c. c (\llbracket t \rrbracket' \lambda v. v) \\ \llbracket \mathbf{J} \rrbracket' &= \lambda c. \mathcal{S} \lambda d. d (c \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c'. \mathcal{S} \lambda d'. d (v v' \lambda v''. v'')}) \end{aligned}$$

A program p is translated as $\langle \llbracket p \rrbracket' \lambda v. v \rangle$.

- In CPS with two layers of continuations (the outer continuation, i.e., the dump continuation, can be η -reduced in the first three clauses), based on the compositional evaluator `evaluate2''` in CPS with two layers of continuations:

$$\begin{aligned} \llbracket n \rrbracket'' &= \lambda c. \lambda d. c n d \\ \llbracket x \rrbracket'' &= \lambda c. \lambda d. c x d \\ \llbracket t_0 t_1 \rrbracket'' &= \lambda c. \lambda d. \llbracket t_1 \rrbracket'' (\lambda v_1. \lambda d. \llbracket t_0 \rrbracket'' (\lambda v_0. \lambda d. v_0 v_1 c d) d) d \\ \llbracket \lambda x.t \rrbracket'' &= \lambda c. \lambda d. c (\lambda x. \lambda c. \lambda d. \llbracket t \rrbracket'' (\lambda v. \lambda d. d v) \lambda v. c v d) d \\ \llbracket \mathbf{J} \rrbracket'' &= \lambda c. \lambda d. c (\lambda v. \lambda c. \lambda d''' . c \boxed{(\lambda v'. \lambda c'. \lambda d'. v v' (\lambda v''. \lambda d''. d'' v'') d)} d''') d \end{aligned}$$

A program p is translated as $\llbracket p \rrbracket'' (\lambda v. \lambda d. d v) \lambda v. v$.

Analysis: The simulation of literals, variables, and applications is standard. The control continuation of the body of each λ -abstraction is delimited, corresponding to it being evaluated with an empty control stack in the SECD machine. The J operator abstracts the control continuation and the dump continuation and immediately restores them, resuming the computation with a state appender which holds the abstracted dump continuation captive. Applying this state appender to a value v yields a program closure (boxed in the three simulations above). Applying this program closure to a value v' has the effect of discarding both the current control continuation and the current dump continuation, applying v to v' , and resuming the captured dump continuation with the result.

Assessment: The first rational deconstruction [35] already characterized the SECD machine in terms of the CPS hierarchy: the control stack is the first continuation, the dump is the second one (i.e., the meta-continuation), and abstraction bodies are evaluated within a control delimiter (i.e., an empty control stack). Our work further characterizes the J operator as capturing (a copy of) the meta-continuation.

3.6.2. *The \mathcal{C} operator and the CPS hierarchy.* In the terminology of reflective towers [42], continuations captured with shift are “pushy”—at their point of invocation, they compose with the current continuation by “pushing” it on the meta-continuation. In the second encoding of J in Section 3.6.1, the term $\mathcal{S}\lambda d'.d (v v' \lambda v''.v'')$ serves to discard the current continuation d' before applying the captured continuation d . Because of this use of shift to discard d' , the continuation d is composed with the identity continuation.

In contrast, still using the terminology of reflective towers, continuations captured with call/cc [29] or with Felleisen’s \mathcal{C} operator [50] are “jumpy”—at their point of invocation, they discard the current continuation. If the continuation d were captured with \mathcal{C} , then the term $d (v v' \lambda v''.v'')$ would suffice to discard the current continuation.

The first encoding of J in Section 3.6.1 uses the pushy control operators \mathcal{S}_1 (i.e., \mathcal{S}) and \mathcal{S}_2 . Murthy [94] and Kameyama [75] have investigated their jumpy counterparts in the CPS hierarchy, \mathcal{C}_1 (i.e., \mathcal{C}) and \mathcal{C}_2 . Jumpy continuations therefore suggest two new simulations of the J operator. We show only the clauses for J, which are the only ones that change compared to Section 3.6.1. As before, we assume a call-by-value meta-language with right-to-left evaluation.

- In direct style, using \mathcal{C}_2 , $\text{reset}_2 (\langle \cdot \rangle_2)$, \mathcal{C}_1 , and $\text{reset}_1 (\langle \cdot \rangle_1)$:

$$\llbracket \mathbf{J} \rrbracket = \mathcal{C}_1 \lambda c. \mathcal{C}_2 \lambda d. d (c \lambda v. \boxed{\lambda v'. d \langle v v' \rangle_1})$$

This simulation provides a new example of programming in the CPS hierarchy with jumpy delimited continuations.

- In CPS with one layer of continuations, using \mathcal{C} and $\text{reset} (\langle \cdot \rangle)$:

$$\llbracket \mathbf{J} \rrbracket' = \lambda c. \mathcal{C} \lambda d. d (c \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c'. d (v v' \lambda v''. v'')})$$

The corresponding CPS simulation of J with two layers of continuations coincides with the one in Section 3.6.1.

3.6.3. *The call/cc operator and the CPS hierarchy.* Like shift and \mathcal{C} , call/cc takes a snapshot of the current context. However, unlike shift and \mathcal{C} , in so doing call/cc leaves the current context in place. So for example, $1 + (\mathbf{call/cc} \lambda k.10)$ yields 11 because call/cc leaves the context $1 + []$ in place, whereas both $1 + (\mathcal{S}\lambda k.10)$ and $1 + (\mathcal{C}\lambda k.10)$ yield 10 because the context $1 + []$ is tossed away.

Therefore J can be simulated in CPS with one layer of continuations, using call/cc and exploiting its non-abortive behavior:

$$\llbracket \mathbf{J} \rrbracket' = \lambda c.\mathbf{call/cc} \lambda d.c \lambda v.\lambda c'.c \boxed{\lambda v'.\lambda c'.d (v v' \lambda v''.v'')}$$

The obvious generalization of call/cc to the CPS hierarchy does not work, however. One needs an abort operator as well in order for call/cc₂ to capture the initial continuation and the current meta-continuation. We leave the rest of this train of thought to the imagination of the reader.

3.6.4. *On the design of control operators.* We note that replacing \mathcal{C} with \mathcal{S} in Section 3.6.2 (resp. \mathcal{C}_1 with \mathcal{S}_1 and \mathcal{C}_2 with \mathcal{S}_2) yields a pushy counterpart for J, i.e., program closures returning to their point of activation. (Similarly, replacing \mathcal{C} with \mathcal{S} in the specification of call/cc in terms of \mathcal{C} yields a pushy version of call/cc, assuming a global control delimiter.) One can also envision an abortive version of J that tosses away the context it abstracts. In that sense, control operators are easy to invent, though not always easy to implement efficiently. Nowadays, however, the litmus test for a new control operator lies elsewhere, for example:

- (1) Which programming idiom does this control operator reflect [29, 38, 41, 102, 108]?
- (2) What is the logical content of this control operator [66, 97]?

Even though it was the first control operator ever, J passes this litmus test. As pointed out by Thielecke,

- (1) besides reflecting Algol jumps and labels [81], J provides a generalized return [115, Section 2.1], and
- (2) the type of $\mathbf{J} \lambda v.v$ is the law of the excluded middle [116, Section 5.2].

On the other hand, despite their remarkable fit to Algol labels and jumps (as illustrated in the beginning of Section 1), the state appenders denoted by J are unintuitive to use. For example, if a let expression is the syntactic sugar of a beta-redex (and x_1 is fresh), the observational equivalence

$$t_0 t_1 \cong \mathbf{let} x_1 = t_1 \mathbf{in} t_0 x_1$$

does *not* hold in the presence of J due to the non-standard translation of abstractions, even though it does hold in the presence of call/cc, \mathcal{C} , and shift for right-to-left evaluation. For example, given $C[] = (\lambda x_2.succ []) 10$, $t_0 = \mathbf{J} (\lambda k.k) 0$, and $t_1 = 100$, $C[t_0 t_1]$ yields 0 whereas $C[\mathbf{let} x_1 = t_1 \mathbf{in} t_0 x_1]$ yields 1.

4. DECONSTRUCTION OF THE SECD MACHINE WITH THE J OPERATOR: CALLER-SAVE DUMPS

In Section 3, we modernized the SECD machine by removing the intermediate data stack and by managing the environment in a caller-save rather than callee-save fashion. We left the ‘non-modern’ feature of the dump continuation alone because it was part of the SECD-machine semantics of the J operator. In this section, we turn our attention to this dump continuation, and we identify that like the environment in the original SECD machine, *the dump continuation is managed in a callee-save fashion*. Indeed the apply function receives a dump continuation from its caller and passes it in turn to the control continuation.

4.1. A specification with caller-save dump continuations. Let us modernize the SECD machine further by managing dump continuation in a caller-save fashion. Our reasoning is similar to that used in Section 3 for the environment. Inspecting the evaluator `evaluate2'` shows that when either `eval` or `apply` receives a control continuation `c` and a dump continuation `d` as arguments and applies `c`, the dump continuation `d` is passed to `c`. Therefore, when the control continuation passed to `eval` or `apply` is `fn (v, d) => d v` and the dump continuation is some `d'`, `d'` can be substituted for `d` in the body of the control continuation. After this change, inspecting the control continuations reveals that the ones created in `apply` and `evaluate2'` ignore their dump-continuation arguments, and the ones created in `eval` are passed a dump continuation that is already in their lexical scope. Therefore, the control continuations do not need to be passed a dump continuation. Since the dump continuation was passed to `apply` solely for the purpose of threading it to the control continuation, `apply` does not need to be passed a dump continuation either.

The evaluator of Section 3.1 with caller-save dump continuations reads as follows:

```
datatype value = INT of int
                | SUCC
                | FUNCLO of E * string * term
                | STATE_APPENDER of D
                | PGMCLO of value * D

withtype E = value Env.env (* environment *)
      and D = value -> value (* dump continuation *)
      and C = value -> value (* control continuation *)

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

(* eval : term * E * C * D -> value *)
(* apply : value * value * C -> value *)
fun eval (LIT n, e, c, d)
  = c (INT n)
  | eval (VAR x, e, c, d)
  = c (Env.lookup (x, e))
  | eval (LAM (x, t), e, c, d)
  = c (FUNCLO (e, x, t))
  | eval (APP (t0, t1), e, c, d)
  = eval (t1, e, fn v1 =>
           eval (t0, e, fn v0 =>
                 apply (v0, v1, c), d), d)
  | eval (J, e, c, d)
  = c (STATE_APPENDER d)
```

```

and apply (SUCC, INT n, c)
  = c (INT (n + 1))
| apply (FUNCLO (e', x, t), v, c)
  = eval (t, Env.extend (x, v, e'), c, c)
| apply (STATE_APPENDER d', v, c)
  = c (PGMCLO (v, d'))
| apply (PGMCLO (v, d'), v', c)
  = apply (v, v', d')

fun evaluate2'_alt t
  = eval (t, e_init, fn v => v, fn v => v)
  (* evaluate2'_alt : program -> value *)

```

This evaluator still passes two continuations to `eval`. However, the dump continuation is no longer passed as an argument to the control continuation. Thus, the two continuations have the same type. The dump continuation is a snapshot of the control continuation of the caller. It is reset to be the continuation of the caller when evaluating the body of a function closure and it is captured in a state appender by the J operator. Applying a program closure discards the current control continuation in favor of the captured dump continuation.

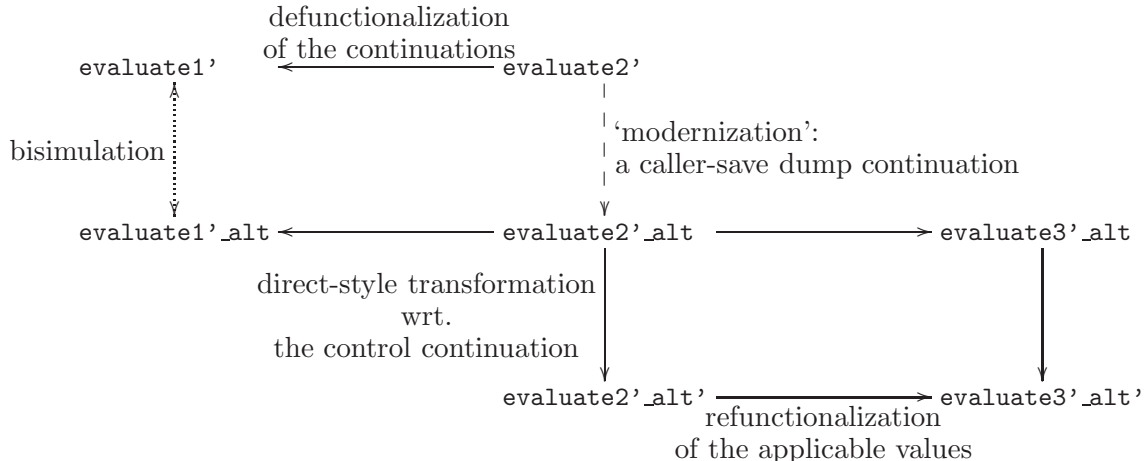
As in Section 3.1, the abstract machine corresponding to `evaluate2'_alt` (obtained by defunctionalization and displayed in Section 8) operates in lockstep with the abstract machine corresponding to `evaluate2'` (obtained by defunctionalization and displayed in Section 7). The following proposition is a corollary of this bisimulation and the correctness of defunctionalization:

Proposition 4.1 (full correctness). *Given a program, `evaluate2'` and `evaluate2'_alt` either both diverge or both yield values; and if these values have an integer type, they are the same integer.*

4.2. The rest of the rational deconstruction. The evaluator of Section 4.1 can be transformed exactly as the higher-order evaluators of Sections 2.2 and 3.1:

- (1) A direct-style transformation with respect to the control continuation yields an evaluator in direct style.
- (2) Refunctionalizing the applicable values yields a compositional, higher-order evaluator in direct style.

Graphically:



4.3. Two other simulations of the J operator. As in Section 3.6.1, the compositional evaluators of Section 4.2 can be viewed as syntax-directed translations into their meta-language. Below, we state these encodings as two further simulations of the J operator: one in CPS with an additional return continuation, and one in direct-style with a return continuation.

- In CPS with an additional return continuation, based on `evaluate3'_alt`:

$$\begin{aligned} \llbracket n \rrbracket' &= \lambda c. \lambda d. c \ n \\ \llbracket x \rrbracket' &= \lambda c. \lambda d. c \ x \\ \llbracket t_0 \ t_1 \rrbracket' &= \lambda c. \lambda d. \llbracket t_1 \rrbracket' (\lambda v_1. \llbracket t_0 \rrbracket' (\lambda v_0. v_0 \ v_1 \ c) \ d) \ d \\ \llbracket \lambda x. t \rrbracket' &= \lambda c. \lambda d. c \ \lambda v. \lambda c. \llbracket t \rrbracket' \ c \ c \\ \llbracket \mathbf{J} \rrbracket' &= \lambda c. \lambda d. c \ \lambda v_0. \lambda c. c \ \boxed{\lambda v_1. \lambda c. v_0 \ v_1 \ d} \end{aligned}$$

A program p is translated as $\llbracket p \rrbracket' (\lambda v. v) \ \lambda v. v$.

- In direct style with a return continuation, based on `evaluate3'_alt'`:

$$\begin{aligned} \llbracket n \rrbracket &= \lambda d. n \\ \llbracket x \rrbracket &= \lambda d. x \\ \llbracket t_0 \ t_1 \rrbracket &= \lambda d. \llbracket t_0 \rrbracket \ d \ (\llbracket t_1 \rrbracket \ d) \\ \llbracket \lambda x. t \rrbracket &= \lambda d. \lambda x. \mathcal{S} \lambda c. \langle c \ (\llbracket t \rrbracket \ c) \rangle \\ \llbracket \mathbf{J} \rrbracket &= \lambda d. \lambda v_0. \boxed{\lambda v_1. \mathcal{S} \lambda c. \langle d \ (v_0 \ v_1) \rangle} \end{aligned}$$

A program p is translated as $\langle \llbracket p \rrbracket \ \lambda v. v \rangle$.

NB. Operationally, the two occurrences of `reset` surrounding the body of the shift-expression are unnecessary. They could be omitted.

Assessment: Transformed terms are passed a pair of continuations, the usual continuation of the call-by-value CPS transform and a return continuation. Abstractions set the return continuation to be the continuation at their point of invocation, i.e., the continuation of their caller. The J operator captures the current return continuation in a program closure (boxed above).

4.4. Thielecke. In his work on comparing control constructs [116], Thielecke introduced a ‘double-barrelled’ CPS transformation, where terms are passed an additional ‘jump continuation’ in addition to the usual continuation of the call-by-value CPS transformation. By varying the transformation of abstractions, he was able to account for first-class continuations, exceptions, and jumping. His double-barrelled CPS transformation, including a clause for his JI operator (i.e., $\mathbf{J} \ \lambda x. x$) and modified for right-to-left evaluation, reads as follows:

$$\begin{aligned} \llbracket x \rrbracket &= \lambda c. \lambda d. c \ x \\ \llbracket t_0 \ t_1 \rrbracket &= \lambda c. \lambda d. \llbracket t_1 \rrbracket (\lambda v_1. \llbracket t_0 \rrbracket (\lambda v_0. v_0 \ v_1 \ c \ d) \ d) \ d \\ \llbracket \lambda x. t \rrbracket &= \lambda c. \lambda d. c \ \lambda x. \lambda c'. \lambda d'. \llbracket t \rrbracket \ c' \ c' \\ \llbracket \mathbf{JI} \rrbracket &= \lambda c. \lambda d. c \ \lambda x. \lambda c'. \lambda d'. d \ x \end{aligned}$$

The continuation c is the continuation of the usual call-by-value CPS transformation. The continuation d is a return continuation, i.e., a snapshot of the continuation of the caller of a function abstraction. It is set to be the continuation of the caller in the body of each function abstraction and it is captured as a first-class function by the JI operator. The extra continuation passed to each abstraction is not necessary (for the encoding of JI), and can be eliminated from the translation of abstractions and applications, as we did in Section 4.1.

As noted by Thielecke and earlier Landin, \mathbf{J} can be expressed in terms of \mathbf{JI} as:

$$\mathbf{J} \equiv (\lambda c. \lambda v. \lambda v'. c (v v')) (\mathbf{JI})$$

The β -expansion is necessary to move the occurrence of \mathbf{JI} outside of the outer abstraction, because λ -abstractions are CPS-transformed in a non-standard way. By CPS-transforming this definition and eliminating the extra continuation for function abstractions, we derive the same double-barrelled encoding of Landin's \mathbf{J} operator as in Section 4.3:

$$\begin{aligned} \llbracket x \rrbracket' &= \lambda c. \lambda d. c x \\ \llbracket t_0 t_1 \rrbracket' &= \lambda c. \lambda d. \llbracket t_1 \rrbracket' (\lambda v_1. \llbracket t_0 \rrbracket' (\lambda v_0. v_0 v_1 c) d) d \\ \llbracket \lambda x. t \rrbracket' &= \lambda c. \lambda d. c \lambda x. \lambda c'. \llbracket t \rrbracket' c' c' \\ \llbracket \mathbf{J} \rrbracket' &= \lambda c. \lambda d. c \lambda v. \lambda c'. c' \boxed{\lambda v'. \lambda c''. v v' d} \end{aligned}$$

Analysis: In essence, Thielecke's simulation corresponds to an abstract machine which is the caller-save counterpart of Landin's machine with respect to the dump.

4.5. Felleisen. Felleisen showed how to embed Landin's extension of applicative expressions with \mathbf{J} into the Scheme programming language [51]. The embedding is defined using Scheme syntactic extensions (i.e., macros). \mathbf{J} is treated as a dynamic identifier that is bound in the body of every abstraction, similarly to the dynamically bound identifier 'self' in an embedding of Smalltalk into Scheme [84]. The control aspect of \mathbf{J} is handled through Scheme's control operator `call/cc`.

Here are the corresponding simulations:

- In direct style, using either of `call/cc`, \mathcal{C} , or `shift` (\mathcal{S}), and a control delimiter ($\langle \cdot \rangle$):

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket t_0 t_1 \rrbracket &= \llbracket t_0 \rrbracket \llbracket t_1 \rrbracket \\ \llbracket \lambda x. t \rrbracket &= \lambda x. \mathbf{call/cc} \lambda d. \mathbf{let} \mathbf{J} = \lambda v. \boxed{\lambda v'. d (v v')} \mathbf{in} \llbracket t \rrbracket \\ &= \lambda x. \mathcal{C} \lambda d. \mathbf{let} \mathbf{J} = \lambda v. \boxed{\lambda v'. d (v v')} \mathbf{in} d \llbracket t \rrbracket \\ &= \lambda x. \mathcal{S} \lambda d. \mathbf{let} \mathbf{J} = \lambda v. \boxed{\lambda v'. \mathcal{S} \lambda c'. d (v v')} \mathbf{in} d \llbracket t \rrbracket \end{aligned}$$

A program p is translated as $\mathbf{let} \mathbf{J} = \lambda v. \lambda v'. \langle v v' \rangle \mathbf{in} \langle \llbracket p \rrbracket \rangle$.

- In CPS:

$$\begin{aligned} \llbracket x \rrbracket' &= \lambda c. c x \\ \llbracket t_0 t_1 \rrbracket' &= \lambda c. \llbracket t_1 \rrbracket' \lambda v_1. \llbracket t_0 \rrbracket' \lambda v_0. v_0 v_1 c \\ \llbracket \lambda x. t \rrbracket' &= \lambda c. c (\lambda x. \lambda d. \mathbf{let} \mathbf{J} = \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c''. v v' d} \mathbf{in} \llbracket t \rrbracket' d) \end{aligned}$$

A program p is translated as $\mathbf{let} \mathbf{J} = \lambda v. \lambda c. c (\lambda v'. \lambda c''. v v' \lambda v''. v'') \mathbf{in} \llbracket p \rrbracket' \lambda v. v$.

Analysis: The simulation of variables and applications is standard. The continuation of the body of each λ -abstraction is captured, and the identifier \mathbf{J} is dynamically bound to a function closure (the state appender) which holds the continuation captive. Applying this function closure to a value v yields a program closure (boxed in the simulations above). Applying this program closure to a value v' has the effect of applying v to v' and resuming the captured continuation with the result, abandoning the current continuation.

The evaluator corresponding to these simulations always has a binding of \mathbf{J} in the environment when evaluating the body of an abstraction (see Section 10). Under the assumption

that J is never shadowed in a program, passing this value as a separate argument to the evaluator leads one towards the definition of `evaluate2' _alt` in Section 4.1 (see Section 9).

5. RELATED WORK

5.1. Landin and Burge. Landin [82] introduced the J operator as a new language feature motivated by three questions about labels and jumps:

- Can a language have jumps without having assignments?
- Is there some component of jumping that is independent of labels?
- Is there some feature that corresponds to functions with arguments in the same sense that labels correspond to procedures without arguments?

Landin gave the semantics of the J operator by extending the SECD machine. In addition to using J to model jumps in Algol 60 [81], he gave examples of programming with the J operator, using it to represent failure actions as program closures where it is essential that they abandon the context of their application.

In his textbook [21, Section 2.10], Burge adjusted Landin's original specification of the J operator. Indeed, in Landin's extension of the SECD machine, J could only occur in the context of an application. Burge adjusted the original specification so that J could occur in arbitrary contexts. To this end, he introduced the notion of a "state appender" as the denotation of J .

Thielecke [115] gave a detailed introduction to the J operator as presented by Landin and Burge. Burstall [22] illustrated the use of the J operator by simulating threads for parallel search algorithms, which in retrospect is the first simulation of threads in terms of first-class continuations ever.

5.2. Reynolds. Reynolds [102] gave a comparison of J to `escape`, the binder form of Scheme's `call/cc` [29].⁶ He gave encodings of Landin's J (i.e., restricted to the context of an application) and `escape` in terms of each other.

His encoding of `escape` in terms of J reads as follows:

$$(\mathbf{escape} \ k \ \mathbf{in} \ t)^* = \mathbf{let} \ k = \mathbf{J} \ \lambda v.v \ \mathbf{in} \ t^*$$

As Thielecke notes [115], this encoding is only valid immediately inside an abstraction. Indeed, the dump continuation captured by J only coincides with the continuation captured by `escape` if the control continuation is the initial one (i.e., immediately inside a control delimiter). Thielecke therefore generalized the encoding by adding a dummy abstraction:

$$(\mathbf{escape} \ k \ \mathbf{in} \ t)^* = (\lambda(). \mathbf{let} \ k = \mathbf{J} \ \lambda x.x \ \mathbf{in} \ t^*) \ ()$$

From the point of view of the rational deconstruction of Section 3, this dummy abstraction implicitly inserts a control delimiter.

Reynolds's converse encoding of J in terms of `escape` reads as follows:

$$(\mathbf{let} \ d = \mathbf{J} \ \lambda x.t_1 \ \mathbf{in} \ t_0)^\circ = \mathbf{escape} \ k \ \mathbf{in} \ (\mathbf{let} \ d = \lambda x.k \ t_1^\circ \ \mathbf{in} \ t_0^\circ)$$

⁶`escape` k `in` $t \equiv \mathbf{call/cc} \ \lambda k.t$

where k does not occur free in t_0 and t_1 . For the same reason as above, this encoding is only valid immediately inside an abstraction and therefore it can be generalized by adding a dummy abstraction:

$$(\mathbf{let} \ d = \mathbf{J} \ \lambda x.t_1 \ \mathbf{in} \ t_0)^\circ = (\lambda().\mathbf{escape} \ k \ \mathbf{in} \ (\mathbf{let} \ d = \lambda x.k \ t_1^\circ \ \mathbf{in} \ t_0^\circ)) \ ()$$

5.3. Felleisen and Burge. Felleisen's version of the SECD machine with the J operator differs from Burge's. In the notation of Section 1.5, Burge's clause for applying program closures reads

```
| run ((PGMCLO (v, (s', e', c') :: d')) :: v' :: s, e, APPLY :: c, d)
  = run (v :: v' :: s', e', APPLY :: c', d')
```

instead of

```
| run ((PGMCLO (v, d')) :: v' :: s, e, APPLY :: c, d)
  = run (v :: v' :: nil, e_init, APPLY :: nil, d')
```

Felleisen's version delays the consumption of the dump until the function, in the program closure, completes, whereas Burge's version does not. The modification is unobservable because a program cannot capture the control continuation and because applying the argument of a state appender pushes the data stack, the environment, and the control stack on the dump. Felleisen's modification can be characterized as wrapping a control delimiter around the argument of a dump continuation, similarly to the simulation of static delimited continuations in terms of dynamic ones [18].

Burge's version, however, is not in defunctionalized form. In Section 6, we put it in defunctionalized form without resorting to a control delimiter and we outline the corresponding compositional evaluation functions and simulations.

6. DECONSTRUCTION OF THE ORIGINAL SECD MACHINE WITH THE J OPERATOR

We now outline the deconstruction of Burge's specification of the SECD machine with the J operator.

6.1. Our starting point: Burge's specification. As pointed out in Section 5.3, Felleisen's version of the SECD machine applies the value contained in a program closure before restoring the components of the captured dump. Burge's version differs by restoring the components of the captured dump before applying the value contained in the program closure. In other words,

- Felleisen's version applies the value contained in a program closure with an empty data stack, a dummy environment, an empty control stack, and the captured dump, whereas
- Burge's version applies the value contained in a program closure with the captured data stack, environment, control stack, and previous dump.

The versions induce a minor programming difference because the first makes it possible to use J in any context whereas the second restricts J to occur only inside a λ -abstraction.

Burge's specification of the SECD machine with J follows. Ellipses mark what does not change from the specification of Section 1.5:

```

(* run : S * E * C * D -> value *)
fun run (v :: nil, e, nil, d)
  = ...
  | run (s, e, (TERM t) :: c, d)
  = ...
  | run (SUCC :: (INT n) :: s, e, APPLY :: c, d)
  = ...
  | run ((FUNCLO (e', x, t)) :: v :: s, e, APPLY :: c, d)
  = ...
  | run ((STATE_APPENDER d') :: v :: s, e, APPLY :: c, d)
  = ...
  | run ((PGMCLO (v, (s', e', c') :: d')) :: v' :: s, e, APPLY :: c, d)
  = run (v :: v' :: s', e', APPLY :: c', d')
fun evaluate0_alt t (* evaluate0_alt : program -> value *)
  = ...

```

Just as in Section 2.1, Burge's specification can be disentangled into four mutually-recursive transition functions. The disentangled specification, however, is not in defunctionalized form. We put it next in defunctionalized form without resorting to a control delimiter, and then outline the rest of the rational deconstruction.

6.2. Burge's specification in defunctionalized form. The disentangled specification of Burge is not in defunctionalized form because the dump does not have a single point of consumption. It is consumed by `run_d` for values yielded by the body of λ -abstractions and in `run_a` for values thrown to program closures. In order to be in the image of defunctionalization and have `run_d` as the apply function, the dump should be solely consumed by `run_d`. We therefore distinguish values yielded by normal evaluation and values thrown to program closures, and we make `run_d` dispatch over these two kinds of returned values. For values yielded by normal evaluation (i.e., in the call from `run_c` to `run_d`), `run_d` proceeds as before. For values thrown to program closures, `run_d` calls `run_a`. Our modification therefore adds one transition (from `run_a` to `run_d`) for values thrown to program closures.

The change only concerns three clauses and ellipses mark what does not change from the evaluator of Section 2.1:

```

datatype returned_value = YIELD of value
                          | THROW of value * value
(* run_c : S * E * C * D -> value *)
(* run_d : returned_value * D -> value *)
(* run_t : term * S * E * C * D -> value *)
(* run_a : value * value * S * E * C * D -> value *)
fun run_c (v :: nil, e, nil, d)
  = run_d (YIELD v, d) (* 1 *)
  | run_c ...
  = ...
and run_d (YIELD v, nil)
  = v
  | run_d (YIELD v, (s, e, c) :: d)
  = run_c (v :: s, e, c, d)
  | run_d (THROW (v, v'), (s, e, c) :: d)
  = run_a (v, v', s, e, c, d) (* 2 *)

```

```

and run_t ...
  = ...
and run_a ...
  = ...
  | run_a (PGMCLO (v, d'), v', s, e, c, d)
    = run_d (THROW (v, v'), d') (* 3 *)
fun evaluate1_alt t (* evaluate1_alt : program -> value *)
  = ...

```

YIELD is used to tag values returned by function closures (in the clause marked “1” above), and THROW is used to tag values sent to program closures (in the clause marked “3”). THROW tags a pair of values, which will be applied in run_d (by calling run_a in the clause marked “2”).

Proposition 6.1 (full correctness). *Given a program, evaluate0_alt and evaluate1_alt either both diverge or both yield values that are structurally equal.*

6.3. A higher-order counterpart. In the modified specification of Section 6.2, the data types of control stacks and dumps are identical to those of the disentangled machine of Section 2.1. These data types, together with run_d and run_c, are in the image of defunctionalization (run_d and run_c are their apply functions). The corresponding higher-order evaluator reads as follows:

```

datatype value = INT of int
               | SUCC
               | FUNCLO of E * string * term
               | STATE_APPENDER of D
               | PGMCLO of value * D
and returned_value = YIELD of value
                   | THROW of value * value

withtype S = value list (* data stack *)
and E = value Env.env (* environment *)
and D = returned_value -> value (* dump continuation *)
and C = S * E * D -> value (* control continuation *)

val e_init = Env.extend ("succ", SUCC, Env.empty)
(* run_t : term * S * E * C * D -> value *)
(* run_a : value * value * S * E * C * D -> value *)
(* where S = value list, E = value Env.env, C = S * E * D -> value *)
(* and D = returned_value -> value *)
fun run_t ...
  = ...

and run_a (SUCC, INT n, s, e, c, d)
  = c ((INT (n+1)) :: s, e, d)
  | run_a (FUNCLO (e', x, t), v, s, e, c, d)
    = run_t (t, nil, Env.extend (x, v, e'),
             fn (v :: nil, e, d) => d (YIELD v),
             fn (YIELD v)
               => c (v :: s, e, d)
             | (THROW (f, v))
               => run_a (f, v, s, e, c, d))

```

```

| run_a (STATE_APPENDER d', v, s, e, c, d)
  = c ((PGMCLO (v, d')) :: s, e, d)
| run_a (PGMCLO (v, d'), v', s, e, c, d)
  = d' (THROW (v, v'))
fun evaluate2_alt t                                (* evaluate2_alt : program -> value *)
  = run_t (t, nil, e_init, fn (v :: nil, e, d) => d (YIELD v),
          fn (YIELD v) => v)

```

As before, the resulting evaluator is in continuation-passing style (CPS), with two layered continuations. It threads a stack of intermediate results, a (callee-save) environment, a control continuation, and a dump continuation. The values sent to dump continuations are tagged to indicate whether they represent the result of a function closure or an application of a program closure. Defunctionalizing this evaluator yields the definition of Section 6.2:

Proposition 6.2 (full correctness). *Given a program, evaluate1_alt and evaluate2_alt either both diverge or yield expressible values; and if these values have an integer type, they are the same integer.*

6.4. The rest of the rational deconstruction. The evaluator of Section 6.3 can be transformed exactly as the higher-order evaluator of Section 2.2:

- (1) Eliminating the data stack and the callee-save environment yields a traditional eval-apply evaluator, with `run_t` as eval and `run_a` as apply. The evaluator is in CPS with two layers of continuations.
- (2) A first direct-style transformation with respect to the dump yields an evaluator that uses shift and reset (or \mathcal{C} and a global reset, or again call/cc and a global reset) to manipulate the implicit dump continuation.
- (3) A second direct-style transformation with respect to the control stack yields an evaluator in direct style that uses the delimited-control operators shift_1 , reset_1 , shift_2 , and reset_2 (or \mathcal{C}_1 , reset_1 , \mathcal{C}_2 , and reset_2) to manipulate the implicit control and dump continuations.
- (4) Refunctionalizing the applicable values yields a compositional, higher-order, direct-style evaluator corresponding to Burge's specification of the J operator. The result is presented as a syntax-directed encoding next.

6.5. Three simulations of the J operator. As in Section 3.6.1, the compositional counterpart of the evaluators of Section 6.4 can be viewed as syntax-directed encodings into their meta-language. Below, we state these encodings as three simulations of J: one in direct style, one in CPS with one layer of continuations, and one in CPS with two layers of continuations. Again, we assume a call-by-value meta-language with right-to-left evaluation and with a sum (to distinguish values returned by functions and values sent to program closures), a case expression (for the body of λ -abstractions) and a destructuring let expression (at the top level).

- In direct style, using either of shift_2 , reset_2 , shift_1 , and reset_1 or \mathcal{C}_2 , reset_2 , \mathcal{C}_1 , and reset_1 , based on the compositional evaluator in direct style:

$$\begin{aligned}
\llbracket n \rrbracket &= n \\
\llbracket x \rrbracket &= x \\
\llbracket t_0 t_1 \rrbracket &= \llbracket t_0 \rrbracket \llbracket t_1 \rrbracket \\
\llbracket \lambda x.t \rrbracket &= \lambda x. \mathbf{case} \langle \mathbf{inL} \llbracket t \rrbracket \rangle_1 \\
&\quad \mathbf{of} \mathbf{inL}(v) \Rightarrow v \\
&\quad \quad | \mathbf{inR}(v, v') \Rightarrow v v' \\
\llbracket \mathbf{J} \rrbracket &= \mathcal{S}_1 \lambda c. \mathcal{S}_2 \lambda d. d (c \lambda v. \boxed{\lambda v'. \mathcal{S}_1 \lambda c'. \mathcal{S}_2 \lambda d'. d (\mathbf{inR}(v, v'))}) \\
&= \mathcal{C}_1 \lambda c. \mathcal{C}_2 \lambda d. d (c \lambda v. \boxed{\lambda v'. d (\mathbf{inR}(v, v'))})
\end{aligned}$$

A program p is translated as $\langle \mathbf{let} \mathbf{inL}(v) = \langle \mathbf{inL}(\llbracket p \rrbracket) \rangle_1 \mathbf{in} v \rangle_2$.

- In CPS with one layer of continuations, using either of shift and reset , \mathcal{C} and reset , or call/cc and reset , based on the compositional evaluator in CPS with one layer of continuations:

$$\begin{aligned}
\llbracket n \rrbracket' &= \lambda c. c n \\
\llbracket x \rrbracket' &= \lambda c. c x \\
\llbracket t_0 t_1 \rrbracket' &= \lambda c. \llbracket t_1 \rrbracket' (\lambda v_1. \llbracket t_0 \rrbracket' \lambda v_0. v_0 v_1 c) \\
\llbracket \lambda x.t \rrbracket' &= \lambda c. c (\lambda x. \lambda c. \mathbf{case} \llbracket t \rrbracket' \lambda v. \mathbf{inL}(v) \\
&\quad \mathbf{of} \mathbf{inL}(v) \Rightarrow c v \\
&\quad \quad | \mathbf{inR}(v, v') \Rightarrow v v' c) \\
\llbracket \mathbf{J} \rrbracket' &= \lambda c. \mathcal{S} \lambda d. d (c \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c'. \mathcal{S} \lambda d'. d (\mathbf{inR}(v, v'))}) \\
&= \lambda c. \mathcal{C} \lambda d. d (c \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c'. d (\mathbf{inR}(v, v'))}) \\
&= \lambda c. \mathbf{call/cc} \lambda d. c \lambda v. \lambda c. c \boxed{\lambda v'. \lambda c'. d (\mathbf{inR}(v, v'))})
\end{aligned}$$

A program p is translated as $\langle \mathbf{let} \mathbf{inL}(v) = \llbracket p \rrbracket' \lambda v. \mathbf{inL}(v) \mathbf{in} v \rangle$.

- In CPS with two layers of continuations, based on the compositional evaluator in CPS with two layers of continuations:

$$\begin{aligned}
\llbracket n \rrbracket'' &= \lambda c. \lambda d. c n d \\
\llbracket x \rrbracket'' &= \lambda c. \lambda d. c x d \\
\llbracket t_0 t_1 \rrbracket'' &= \lambda c. \lambda d. \llbracket t_1 \rrbracket'' (\lambda v_1. \lambda d. \llbracket t_0 \rrbracket'' (\lambda v_0. \lambda d. v_0 v_1 c d) d) d \\
\llbracket \lambda x.t \rrbracket'' &= \lambda c. \lambda d. c (\lambda x. \lambda c. \lambda d. \llbracket t \rrbracket'' (\lambda v. \lambda d. d (\mathbf{inL}(v))) \\
&\quad \lambda v''. \mathbf{case} v'' \\
&\quad \quad \mathbf{of} \mathbf{inL}(v) \Rightarrow c v d \\
&\quad \quad \quad | \mathbf{inR}(v, v') \Rightarrow v v' c d) d \\
\llbracket \mathbf{J} \rrbracket'' &= \lambda c. \lambda d. c (\lambda v. \lambda c. \lambda d''' . c \boxed{(\lambda v'. \lambda c'. \lambda d'. d (\mathbf{inR}(v, v'))})} d''') d
\end{aligned}$$

A program p is translated as $\llbracket p \rrbracket'' (\lambda v. \lambda d. d (\mathbf{inL}(v))) (\lambda v. \mathbf{let} \mathbf{inL}(v') = v \mathbf{in} v')$.

Analysis: The simulation of literals, variables, and applications is standard. The body of each λ -abstraction is evaluated with a control continuation injecting the resulting value into the sum type⁷ to indicate normal completion and resuming the current dump continuation, and with a dump continuation inspecting the resulting sum to determine whether to continue normally or to apply a program closure. Continuing normally consists of invoking

⁷This machine is therefore not properly tail recursive.

the control continuation with the resulting value and the dump continuation. Applying a program closure consists of restoring the components of the dump and then performing the application. The J operator abstracts both the control continuation and the dump continuation and immediately restores them, resuming the computation with a state appender holding the abstracted dump continuation captive. Applying this state appender to a value v yields a program closure (boxed in the three simulations above). Applying this program closure to a value v' has the effect of discarding both the current control continuation and the current dump continuation, injecting v and v' into the sum type to indicate exceptional completion, and resuming the captured dump continuation. It is an error to evaluate J outside of a λ -abstraction.

6.6. Related work. Kiselyov's encoding of dynamic delimited continuations in terms of the static delimited-continuation operators `shift` and `reset` [78] is similar to this alternative encoding of the J operator in that both encodings tag the argument to the meta-continuation to indicate whether it represents a normal return or a value thrown to a first-class continuation. In addition though, Kiselyov uses a recursive meta-continuation in order to encode dynamic delimited continuations.

7. A SYNTACTIC THEORY OF APPLICATIVE EXPRESSIONS WITH THE J OPERATOR: EXPLICIT, CALLEE-SAVE DUMPS

Symmetrically to the functional correspondence between evaluation functions and abstract machines that was sparked by the first rational deconstruction of the SECD machine [3, 4, 6, 7, 13, 16, 35, 36], a syntactic correspondence exists between calculi and abstract machines, as investigated by Biernacka, Danvy, and Nielsen [12, 14, 15, 34, 36, 45]. This syntactic correspondence is also derivational, and hinges not on defunctionalization but on a 'refocusing' transformation that mechanically connects an evaluation function defined as the iteration of one-step reduction, and an abstract machine.

The goal of this section is to present the reduction semantics and the reduction-based evaluation function that correspond to the modernized SECD machine of Section 3.1. We successively present this machine (Section 7.1), the syntactic correspondence (Section 7.2), a reduction semantics for applicative expressions with the J operator (Section 7.3), and the derivation from this reduction semantics to this SECD machine (Section 7.4). We consider a calculus of explicit substitutions because the explicit substitutions directly correspond to the environments of the modernized SECD machine. In turn, this calculus of explicit substitutions directly corresponds to a calculus with actual substitutions.

7.1. The SECD machine with no data stack and caller-save environments, revisited. The terms, values, environments, and contexts are defined as in Section 1.5:

$$\begin{aligned}
(\text{programs}) \quad p &::= t[(succ, SUCC) \cdot \emptyset] \\
(\text{terms}) \quad t &::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid tt \mid J \\
(\text{values}) \quad v &::= \ulcorner n \urcorner \mid SUCC \mid (\lambda x.t, e) \mid \ulcorner D \urcorner \circ v \mid \ulcorner D \urcorner \\
(\text{environments}) \quad e &::= \emptyset \mid (x, v) \cdot e \\
(\text{control contexts}) \quad C &::= [] \mid C[(t, e) []] \mid C[[] v] \\
(\text{dump contexts}) \quad D &::= \bullet \mid C \cdot D
\end{aligned}$$

The following four transition functions are the stackless, caller-save respective counterparts of `run_t`, `run_a`, `run_c`, and `run_d` in Section 2.1. This abstract machine is implemented by the modernized and disentangled evaluator `evaluate1'` in the diagram at the end of Section 3.1:

$$\begin{aligned}
\langle \ulcorner n \urcorner, e, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{cont}} \\
\langle x, e, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, v, D \rangle_{\text{cont}} && \text{if } \textit{lookup}(x, e) = v \\
\langle \lambda x.t, e, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t, e), D \rangle_{\text{cont}} \\
\langle t_0 t_1, e, C, D \rangle_{\text{eval}} &\Rightarrow \langle t_1, e, C[(t_0, e) []], D \rangle_{\text{eval}} \\
\langle J, e, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner D \urcorner, D \rangle_{\text{cont}} \\
\langle \textit{SUCC}, \ulcorner n \urcorner, C, D \rangle_{\text{apply}} &\Rightarrow \langle C, \ulcorner n + 1 \urcorner, D \rangle_{\text{cont}} \\
\langle (\lambda x.t, e), v, C, D \rangle_{\text{apply}} &\Rightarrow \langle t, e', [], C \cdot D \rangle_{\text{eval}} && \text{where } e' = \textit{extend}(x, v, e) \\
\langle \ulcorner D \urcorner \circ v', v, C, D \rangle_{\text{apply}} &\Rightarrow \langle v, v', [], D' \rangle_{\text{apply}} \\
\langle \ulcorner D \urcorner, v, C, D \rangle_{\text{apply}} &\Rightarrow \langle C, \ulcorner D \urcorner \circ v, D \rangle_{\text{cont}} \\
\langle [], v, D \rangle_{\text{cont}} &\Rightarrow \langle D, v \rangle_{\text{dump}} \\
\langle C[(t, e) []], v, D \rangle_{\text{cont}} &\Rightarrow \langle t, e, C[[] v], D \rangle_{\text{eval}} \\
\langle C[[] v'], v, D \rangle_{\text{cont}} &\Rightarrow \langle v, v', C, D \rangle_{\text{apply}} \\
\langle \bullet, v \rangle_{\text{dump}} &\Rightarrow v \\
\langle C \cdot D, v \rangle_{\text{dump}} &\Rightarrow \langle C, v, D \rangle_{\text{cont}}
\end{aligned}$$

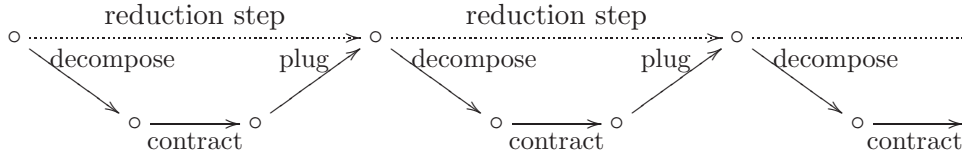
A program t is evaluated by starting in the configuration $\langle t, (\textit{succ}, \textit{SUCC}) \cdot \emptyset, [], \bullet \rangle_{\text{eval}}$. The machine halts with a value v if it reaches a configuration $\langle \bullet, v \rangle_{\text{dump}}$.

7.2. From reduction semantics to abstract machine. Consider a calculus together with a reduction strategy expressed as a Felleisen-style reduction semantics satisfying the unique-decomposition property [50]. In such a reduction semantics, a one-step reduction function is defined as the composition of three functions:

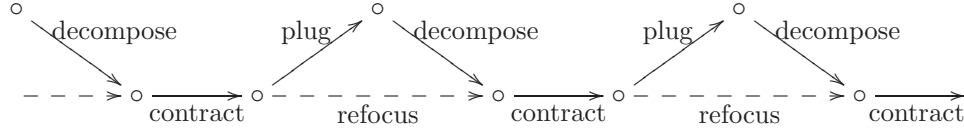
- decomposition:** a total function mapping a value term to itself and decomposing a non-value term into a potential redex and a reduction context (decomposition is a function because of the unique-decomposition property);
- contraction:** a partial function mapping an actual redex to its contractum; and
- plugging:** a total function mapping a term and a reduction context to a new term by filling the hole in the context with the term.

The one-step reduction function is partial because it is the composition of two total functions and a partial function.

An evaluation function is traditionally defined as the iteration of the one-step reduction function:



Danvy and Nielsen have observed that composing the two total functions *plug* and *decompose* into a ‘refocus’ function could avoid the construction of intermediate terms:



The resulting ‘refocused’ evaluation function is defined as the iteration of refocusing and contraction. CPS transformation and defunctionalization make it take the form of a state-transition function, i.e., an abstract machine. Short-circuiting its intermediate transitions yields abstract machines that are often independently known [45].

Biernacka and Danvy then showed that the refocusing technique could be applied to the very first calculus of explicit substitutions, Curien’s simple calculus of closures [31], and that depending on the reduction order, it gave rise to a collection of both known and new environment-based abstract machines such as Felleisen et al.’s CEK machine (for left-to-right applicative order), the Krivine machine (for normal order), Krivine’s machine (for normal order with generalized reduction), and Leroy’s ZINC machine (for right-to-left applicative order with generalized reduction) [14]. They then turned to context-sensitive contraction functions, as first proposed by Felleisen [50], and showed that refocusing mechanically gives rise to an even larger collection of both known and new environment-based abstract machines for languages with computational effects such as Krivine’s machine with call/cc, the $\lambda\mu$ -calculus, static and dynamic delimited continuations, input/output, stack inspection, proper tail-recursion, and lazy evaluation [15].

The next section presents the calculus of closures corresponding to the abstract machine of Section 7.1.

7.3. A reduction semantics for applicative expressions with the J operator. The $\lambda\hat{\rho}J$ -calculus is an extension of Biernacka and Danvy’s $\lambda\hat{\rho}$ -calculus [14], which is itself a minimal extension of Curien’s original calculus of closures $\lambda\rho$ [31] to make it closed under one-step reduction. We use it here to formalize Landin’s applicative expressions with the J operator as a reduction semantics. To this end, we present its syntactic categories (Section 7.3.1); a *plug* function mapping a closure and a two-layered reduction context into a closure by filling the given context with the given closure (Section 7.3.2); a contraction function implementing a context-sensitive notion of reduction (Section 7.3.3) and therefore mapping a potential redex and its reduction context into a contractum and a reduction context (possibly another one); and a decomposition function mapping a non-value term into a potential redex and a reduction context (Section 7.3.4). We are then in position to define a one-step reduction function (Section 7.3.5), and a reduction-based evaluation function (Section 7.3.6).

Before delving into this section, the reader might want to first browse through Section E, in the appendix. This section has the same structure as the present one but instead of the SECD machine, it addresses the CEK machine, which is simpler.

7.3.1. *Syntactic categories.* We consider a variant of the $\lambda\hat{\rho}J$ -calculus with names instead of de Bruijn indices, and with two layers of contexts C and D that embody the right-to-left applicative-order reduction strategy favored by Landin: C is the control context and D is the dump context. In the syntactic category of closures, $\ulcorner D \urcorner$ and $\ulcorner D \urcorner \circ v$ respectively denote a state appender and a program closure, and $\langle c \rangle$ (which is shaded below) marks the boundary between the context of a β -redex that has been contracted, i.e., a function closure that has been applied, and the body of the λ -abstraction in this function closure:

$$\begin{aligned}
(\text{programs}) \quad p &::= t[(succ, SUCC) \cdot \emptyset] \\
(\text{terms}) \quad t &::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid tt \mid J \\
(\text{closures}) \quad c &::= \ulcorner n \urcorner \mid SUCC \mid t[e] \mid cc \mid \ulcorner D \urcorner \mid \ulcorner D \urcorner \circ v \mid \langle c \rangle \\
(\text{values}) \quad v &::= \ulcorner n \urcorner \mid SUCC \mid (\lambda x.t)[e] \mid \ulcorner D \urcorner \mid \ulcorner D \urcorner \circ v \\
(\text{potential redexes}) \quad r &::= x[e] \mid vv \mid J \\
(\text{substitutions}) \quad e &::= \emptyset \mid (x, v) \cdot e \\
(\text{control contexts}) \quad C &::= [] \mid C[c []] \mid C[[] v] \\
(\text{dump contexts}) \quad D &::= \bullet \mid C \cdot D
\end{aligned}$$

Values are therefore a syntactic subcategory of closures, and in this section, we make use of the syntactic coercion \uparrow mapping a value into a closure.

7.3.2. *Plugging.* Plugging a closure in the two layered contexts is defined by induction over these two contexts. We express this definition as a state-transition system with two intermediate states, $\langle C, c, D \rangle_{\text{plug/cont}}$ and $\langle D, c \rangle_{\text{plug/dump}}$, an initial state $\langle C, c, D \rangle_{\text{plug/cont}}$, and a final state c . The transition function from the state $\langle C, c, D \rangle_{\text{plug/cont}}$ incrementally peels off the given control context and the transition function from the state $\langle D, c \rangle_{\text{plug/dump}}$ dispatches over the given dump context:

$$\begin{aligned}
\langle [], c, D \rangle_{\text{plug/cont}} &\rightarrow \langle D, c \rangle_{\text{plug/dump}} \\
\langle C[c_0 [], c_1, D] \rangle_{\text{plug/cont}} &\rightarrow \langle C, c_0 c_1, D \rangle_{\text{plug/cont}} \\
\langle C[[] v_1], c_0, D \rangle_{\text{plug/cont}} &\rightarrow \langle C, c_0 c_1, D \rangle_{\text{plug/cont}} \quad \text{where } c_1 = \uparrow v_1 \\
\langle \bullet, c \rangle_{\text{plug/dump}} &\rightarrow c \\
\langle C \cdot D, c \rangle_{\text{plug/dump}} &\rightarrow \langle \langle c \rangle, C, D \rangle_{\text{plug/cont}}
\end{aligned}$$

We can now define a total function `plug` over closures, control contexts, and dump contexts that fills the given closure into the given control context, and further fills the result into the given dump context:

$$\text{plug} : \text{Closure} \times \text{Control} \times \text{Dump} \rightarrow \text{Closure}$$

Definition 7.1. For any closure c , control context C , and dump context D , `plug` $(C, c, D) = c'$ if and only if $\langle C, c, D \rangle_{\text{plug/cont}} \rightarrow^* c'$.

7.3.3. *Notion of contraction.* The notion of reduction over applicative expressions with the J operator is specified by the following context-sensitive contraction rules over actual redexes:

$$(\text{Var}) \quad \langle x[e], C, D \rangle \mapsto \langle v, C, D \rangle \quad \text{if } \text{lookup}(x, e) = v$$

$$\begin{aligned}
(\text{Beta}_{succ}) \quad & \langle SUCC \ulcorner n \urcorner, C, D \rangle \mapsto \langle \ulcorner n + 1 \urcorner, C, D \rangle \\
(\text{Beta}_{FC}) \quad & \langle ((\lambda x.t)[e]) v, C, D \rangle \mapsto \langle t[e'], [], C \cdot D \rangle \quad \text{where } e' = \text{extend}(x, v, e) = (x, v) \cdot e \\
(\text{Beta}_{SA}) \quad & \langle \ulcorner D \urcorner v, C, D \rangle \mapsto \langle \ulcorner D \urcorner \circ v, C, D \rangle \\
(\text{Beta}_{PC}) \quad & \langle (\ulcorner D \urcorner \circ v') v, C, D \rangle \mapsto \langle v' v, [], D' \rangle \\
(\text{Prop}) \quad & \langle (t_0 t_1)[e], C, D \rangle \mapsto \langle (t_0[e]) (t_1[e]), C, D \rangle \\
(J) \quad & \langle J, C, D \rangle \mapsto \langle \ulcorner D \urcorner, C, D \rangle
\end{aligned}$$

Three of these contraction rules depend on the contexts: the J rule captures a copy of the dump context and yields a state appender; the β -rule for function closures resets the control context and pushes it on the dump context; and the β -rule for program closures resets the control context and reinstates a previously captured copy of the dump context.

Among the potential redexes, only the ones listed above are actual ones. The other applications of one value to another are stuck.

We now can define by cases a partial function **contract** over potential redexes that contracts an actual redex and its two layers of contexts into the corresponding contractum and contexts:

$$\text{contract} : \text{PotRed} \times \text{Control} \times \text{Dump} \rightarrow \text{Closure} \times \text{Control} \times \text{Dump}$$

Definition 7.2. For any potential redex r , control context C , and dump context D , $\text{contract}(r, C, D) = \langle c, C', D' \rangle$ if and only if $\langle r, C, D \rangle \mapsto \langle c, C', D' \rangle$.

7.3.4. *Decomposition.* There are many ways to define a total function mapping a value closure to itself and a non-value closure to a potential redex and a reduction context. In our experience, the following definition is a convenient one. It is a state-transition system with three intermediate states, $\langle c, C, D \rangle_{\text{dec/clos}}$, $\langle C, v, D \rangle_{\text{dec/cont}}$, and $\langle D, v \rangle_{\text{dec/dump}}$, an initial state $\langle c, [], \bullet \rangle_{\text{dec/clos}}$ and two final states VAL(v) and DEC(r, C, D). If possible, the transition function from the state $\langle c, C, D \rangle_{\text{dec/clos}}$ decomposes the given closure c and accumulates the corresponding two layers of reduction context, C and D . The transition function from the state $\langle C, v, D \rangle_{\text{dec/cont}}$ dispatches over the given control context, and the transition function from the state $\langle D, v \rangle_{\text{dec/dump}}$ dispatches over the given dump context.

$$\begin{aligned}
& \langle \ulcorner n \urcorner, C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{dec/cont}} \\
& \langle SUCC, C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, SUCC, D \rangle_{\text{dec/cont}} \\
& \langle \ulcorner n \urcorner[e], C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{dec/cont}} \\
& \langle x[e], C, D \rangle_{\text{dec/clos}} \rightarrow \text{DEC}(x[e], C, D) \\
& \langle (\lambda x.t)[e], C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, (\lambda x.t)[e], D \rangle_{\text{dec/cont}} \\
& \langle (t_0 t_1)[e], C, D \rangle_{\text{dec/clos}} \rightarrow \text{DEC}((t_0 t_1)[e], C, D) \\
& \langle J[e], C, D \rangle_{\text{dec/clos}} \rightarrow \text{DEC}(J, C, D) \\
& \langle c_0 c_1, C, D \rangle_{\text{dec/clos}} \rightarrow \langle c_1, C[c_0 [], D] \rangle_{\text{dec/clos}} \\
& \langle \ulcorner D \urcorner, C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, \ulcorner D \urcorner, D \rangle_{\text{dec/cont}} \\
& \langle \ulcorner D \urcorner \circ v, C, D \rangle_{\text{dec/clos}} \rightarrow \langle C, \ulcorner D \urcorner \circ v, D \rangle_{\text{dec/cont}} \\
& \langle \langle c \rangle, C, D \rangle_{\text{dec/clos}} \rightarrow \langle c, [], C \cdot D \rangle_{\text{dec/clos}}
\end{aligned}$$

$$\begin{aligned}
\langle [], v, D \rangle_{\text{dec/cont}} &\rightarrow \langle D, v \rangle_{\text{dec/dump}} \\
\langle C[c_0 [], v_1, D] \rangle_{\text{dec/cont}} &\rightarrow \langle c_0, C[[] v_1], D \rangle_{\text{dec/clos}} \\
\langle C[[] v_1], v_0, D \rangle_{\text{dec/cont}} &\rightarrow \text{DEC } (v_0 v_1, C, D) \\
\langle \bullet, v \rangle_{\text{dec/dump}} &\rightarrow \text{VAL } (v) \\
\langle C \cdot D, v \rangle_{\text{dec/dump}} &\rightarrow \langle C, v, D \rangle_{\text{dec/cont}}
\end{aligned}$$

We now can define a total function `decompose` over closures that maps a value closure to itself and a non-value closure to a decomposition into a potential redex, a control context, and a dump context. This total function uses three auxiliary functions `decompose'`_{clos}, `decompose'`_{cont}, and `decompose'`_{dump}:

$$\begin{aligned}
\text{decompose} &: \text{Closure} && \rightarrow \text{Value} + (\text{PotRed} \times \text{Control} \times \text{Dump}) \\
\text{decompose}'_{\text{clos}} &: \text{Closure} \times \text{Control} \times \text{Dump} && \rightarrow \text{Value} + (\text{PotRed} \times \text{Control} \times \text{Dump}) \\
\text{decompose}'_{\text{cont}} &: \text{Control} \times \text{Value} \times \text{Dump} && \rightarrow \text{Value} + (\text{PotRed} \times \text{Control} \times \text{Dump}) \\
\text{decompose}'_{\text{dump}} &: \text{Dump} \times \text{Value} && \rightarrow \text{Value} + (\text{PotRed} \times \text{Control} \times \text{Dump})
\end{aligned}$$

Definition 7.3. For any closure c , control context C , and dump context D ,

$$\begin{aligned}
\text{decompose}'_{\text{clos}}(c, C, D) &= \begin{cases} \text{VAL } (v') & \text{if } \langle c, C, D \rangle_{\text{dec/clos}} \rightarrow^* \text{VAL } (v') \\ \text{DEC } (r, C', D') & \text{if } \langle c, C, D \rangle_{\text{dec/clos}} \rightarrow^* \text{DEC } (r, C', D') \end{cases} \\
\text{decompose}'_{\text{cont}}(C, v, D) &= \begin{cases} \text{VAL } (v') & \text{if } \langle C, v, D \rangle_{\text{dec/cont}} \rightarrow^* \text{VAL } (v') \\ \text{DEC } (r, C', D') & \text{if } \langle C, v, D \rangle_{\text{dec/cont}} \rightarrow^* \text{DEC } (r, C', D') \end{cases} \\
\text{decompose}'_{\text{dump}}(D, v) &= \begin{cases} \text{VAL } (v') & \text{if } \langle D, v \rangle_{\text{dec/dump}} \rightarrow^* \text{VAL } (v') \\ \text{DEC } (r, C', D') & \text{if } \langle D, v \rangle_{\text{dec/dump}} \rightarrow^* \text{DEC } (r, C', D') \end{cases}
\end{aligned}$$

and $\text{decompose}(c) = \text{decompose}'_{\text{clos}}(c, [], \bullet)$.

7.3.5. One-step reduction. We are now in position to define a partial function `reduce` over closed closures that maps a value closure to itself and a non-value closure to the next closure in the reduction sequence. This function is defined by composing the three functions above:

$$\begin{aligned}
\text{reduce}(c) &= \text{case } \text{decompose}(c) \\
&\quad \text{of VAL } (v) && \Rightarrow \uparrow v \\
&\quad \mid \text{DEC } (r, C, D) && \Rightarrow \text{plug}(\text{contract}(r, C, D))
\end{aligned}$$

The function `reduce` is partial because of `contract`, which is undefined for stuck closures.

Definition 7.4 (One-step reduction). For any closure c , $c \rightarrow c'$ if and only if $\text{reduce}(c) = c'$.

7.3.6. Reduction-based evaluation. Iterating `reduce` defines a reduction-based evaluation function. The definition below uses `decompose` to distinguish between values and non-values, and implements iteration (tail-) recursively with the partial function `iterate`:

$$\text{evaluate}(c) = \text{iterate}(\text{decompose}(c))$$

where

$$\begin{cases} \text{iterate}(\text{VAL } (v)) & = v \\ \text{iterate}(\text{DEC } (r, C, D)) & = \text{iterate}(\text{decompose}(\text{plug}(\text{contract}(r, C, D)))) \end{cases}$$

The function `evaluate` is partial because a given closure might be stuck or reducing it might not converge.

Definition 7.5 (Reduction-based evaluation). For any closure c , $c \rightarrow^* v$ if and only if `evaluate` $(c) = v$.

To close, let us adjust the definition of `evaluate` by exploiting the fact that for any closure c , `plug` $(c, [], \bullet) = c$:

$$\text{evaluate}(c) = \text{iterate}(\text{decompose}(\text{plug}(c, [], \bullet)))$$

In this adjusted definition, `decompose` is always applied to the result of `plug`.

7.4. From the reduction semantics for applicative expressions to the SECD machine. Deforesting the intermediate terms in the reduction-based evaluation function of Section 7.3.6 yields a reduction-free evaluation function in the form of a small-step abstract machine (Section 7.4.1). We simplify this small-step abstract machine by fusing a part of its driver loop with the contraction function (Section 7.4.2) and compressing its ‘corridor’ transitions (Section 7.4.3). Unfolding the recursive data type of closures precisely yields the caller-save, stackless SECD abstract machine of Section 7.1 (Section 7.4.4).

7.4.1. Refocusing: from reduction-based to reduction-free evaluation. Following Danvy and Nielsen [45], we deforest the intermediate closure in the reduction sequence by replacing the composition of `plug` and `decompose` by a call to a composite function `refocus`:

$$\text{evaluate}(c) = \text{iterate}(\text{refocus}(c, [], \bullet))$$

where

$$\begin{cases} \text{iterate}(\text{VAL}(v)) & = v \\ \text{iterate}(\text{DEC}(r, C, D)) & = \text{iterate}(\text{refocus}(\text{contract}(r, C, D))) \end{cases}$$

and `refocus` is optimally defined as continuing the decomposition in the current reduction context [45]:

$$\text{refocus}(c, C, D) = \text{decompose}'_{\text{clos}}(c, C, D)$$

Definition 7.6 (Reduction-free evaluation). For any closure c , $c \mapsto_J^* v$ if and only if `evaluate` $(c) = v$.

7.4.2. Lightweight fusion: making do without driver loop. In effect, `iterate` is as the ‘driver loop’ of a small-step abstract machine that refocuses and contracts. Instead, let us fuse `contract` and `iterate` and express the result with rewriting rules over a configuration $\langle r, C, D \rangle_{\text{iter}}$. We clone the rewriting rules for `decompose'`_{clos}, `decompose'`_{cont}, and `decompose'`_{dump} into refocusing rules, respectively indexing the configuration $\langle c, C, D \rangle_{\text{dec/clos}}$ as $\langle c, C, D \rangle_{\text{eval}}$, the configuration $\langle C, v, D \rangle_{\text{dec/cont}}$ as $\langle C, v, D \rangle_{\text{cont}}$, and the configuration $\langle D, v \rangle_{\text{dec/dump}}$ as $\langle D, v \rangle_{\text{dump}}$:

- instead of rewriting to `VAL` (v) , the cloned rules rewrite to v ;
- instead of rewriting to `DEC` (r, C, D) , the cloned rules rewrite to $\langle r, C, D \rangle_{\text{iter}}$.

The result reads as follows:

$$\begin{aligned}
\langle \ulcorner n \urcorner, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{cont}} \\
\langle \text{SUCC } C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \text{SUCC } D \rangle_{\text{cont}} \\
\langle \ulcorner n \urcorner[e], C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{cont}} \\
\langle x[e], C, D \rangle_{\text{eval}} &\Rightarrow \langle x[e], C, D \rangle_{\text{iter}} \\
\langle (\lambda x.t)[e], C, D \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t)[e], D \rangle_{\text{cont}} \\
\langle (t_0 t_1)[e], C, D \rangle_{\text{eval}} &\Rightarrow \langle (t_0 t_1)[e], C, D \rangle_{\text{iter}} \\
\langle J[e], C, D \rangle_{\text{eval}} &\Rightarrow \langle J, C, D \rangle_{\text{iter}} \\
\langle c_0 c_1, C, D \rangle_{\text{eval}} &\Rightarrow \langle c_1, C[c_0 []], D \rangle_{\text{eval}} \\
\langle \ulcorner D \urcorner, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner D \urcorner, D \rangle_{\text{cont}} \\
\langle \ulcorner D \urcorner \circ v, C, D \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner D \urcorner \circ v, D \rangle_{\text{cont}} \\
\langle \langle c \rangle, C, D \rangle_{\text{eval}} &\Rightarrow \langle c, [], C \cdot D \rangle_{\text{eval}} \\
\langle [], v, D \rangle_{\text{cont}} &\Rightarrow \langle D, v \rangle_{\text{dump}} \\
\langle C[c_0 []], v_1, D \rangle_{\text{cont}} &\Rightarrow \langle c_0, C[[] v_1], D \rangle_{\text{eval}} \\
\langle C[[] v_1], v_0, D \rangle_{\text{cont}} &\Rightarrow \langle v_0 v_1, C, D \rangle_{\text{iter}} \\
\langle \bullet, v \rangle_{\text{dump}} &\Rightarrow v \\
\langle C \cdot D, v \rangle_{\text{dump}} &\Rightarrow \langle C, v, D \rangle_{\text{cont}} \\
\langle x[e], C, D \rangle_{\text{iter}} &\Rightarrow \langle v, C, D \rangle_{\text{eval}} && \text{if } \text{lookup}(x, e) = v \\
\langle \text{SUCC } \ulcorner n \urcorner, C, D \rangle_{\text{iter}} &\Rightarrow \langle \ulcorner n + 1 \urcorner, C, D \rangle_{\text{eval}} \\
\langle ((\lambda x.t)[e]) v, C, D \rangle_{\text{iter}} &\Rightarrow \langle t[e'], [], C \cdot D \rangle_{\text{eval}} && \text{where } e' = \text{extend}(x, v, e) = (x, v) \cdot e \\
\langle \ulcorner D \urcorner v, C, D \rangle_{\text{iter}} &\Rightarrow \langle \ulcorner D \urcorner \circ v, C, D \rangle_{\text{eval}} \\
\langle (\ulcorner D \urcorner \circ v') v, C, D \rangle_{\text{iter}} &\Rightarrow \langle v' v, [], D' \rangle_{\text{eval}} \\
\langle (t_0 t_1)[e], C, D \rangle_{\text{iter}} &\Rightarrow \langle (t_0[e]) (t_1[e]), C, D \rangle_{\text{eval}} \\
\langle J, C, D \rangle_{\text{iter}} &\Rightarrow \langle \ulcorner D \urcorner, C, D \rangle_{\text{eval}}
\end{aligned}$$

The following proposition summarizes the situation:

Proposition 7.7. *For any closure c , evaluate(c) = v if and only if $\langle c, [], \bullet \rangle_{\text{eval}} \Rightarrow^* v$.*

Proof: straightforward. The two machines operate in lockstep. \square

7.4.3. Inlining and transition compression. The abstract machine of Section 7.4.2, while interesting in its own right (it is ‘staged’ in that the contraction rules are implemented separately from the congruence rules [14,69]), is not minimal: a number of transitions yield a configuration whose transition is uniquely determined. Let us carry out these hereditary, “corridor” transitions once and for all:

- $\langle x[e], C, D \rangle_{\text{eval}} \Rightarrow \langle x[e], C, D \rangle_{\text{iter}} \Rightarrow \langle v, C, D \rangle_{\text{eval}} \Rightarrow \langle C, v, D \rangle_{\text{cont}}$ if $\text{lookup}(x, e) = v$
- $\langle (t_0 t_1)[e], C, D \rangle_{\text{eval}} \Rightarrow \langle (t_0 t_1)[e], C, D \rangle_{\text{iter}} \Rightarrow \langle (t_0[e]) (t_1[e]), C, D \rangle_{\text{eval}} \Rightarrow \langle t_1[e], C[(t_0[e]) []], D \rangle_{\text{eval}}$
- $\langle J[e], C, D \rangle_{\text{eval}} \Rightarrow \langle J, C, D \rangle_{\text{iter}} \Rightarrow \langle \ulcorner D \urcorner, C, D \rangle_{\text{eval}} \Rightarrow \langle C, \ulcorner D \urcorner, D \rangle_{\text{cont}}$
- $\langle \text{SUCC } \ulcorner n \urcorner, C, D \rangle_{\text{iter}} \Rightarrow \langle \ulcorner n + 1 \urcorner, C, D \rangle_{\text{eval}} \Rightarrow \langle C, \ulcorner n + 1 \urcorner, D \rangle_{\text{cont}}$

- $\langle \ulcorner D \urcorner v, C, D \rangle_{\text{iter}} \Rightarrow \langle \ulcorner D \urcorner \circ v, C, D \rangle_{\text{eval}} \Rightarrow \langle C, \ulcorner D \urcorner \circ v, D \rangle_{\text{cont}}$

The result reads as follows:

$$\begin{aligned}
& \langle \ulcorner n \urcorner [e], C, D \rangle_{\text{eval}} \Rightarrow \langle C, \ulcorner n \urcorner, D \rangle_{\text{cont}} \\
& \langle x[e], C, D \rangle_{\text{eval}} \Rightarrow \langle C, v, D \rangle_{\text{cont}} \quad \text{if } \textit{lookup}(x, e) = v \\
& \langle (\lambda x.t)[e], C, D \rangle_{\text{eval}} \Rightarrow \langle C, (\lambda x.t)[e], D \rangle_{\text{cont}} \\
& \langle (t_0 t_1)[e], C, D \rangle_{\text{eval}} \Rightarrow \langle t_1[e], C[(t_0[e]) []], D \rangle_{\text{eval}} \\
& \langle J[e], C, D \rangle_{\text{eval}} \Rightarrow \langle C, \ulcorner D \urcorner, D \rangle_{\text{cont}} \\
& \langle \textit{SUCC} \ulcorner n \urcorner, C, D \rangle_{\text{iter}} \Rightarrow \langle C, \ulcorner n + 1 \urcorner, D \rangle_{\text{cont}} \\
& \langle ((\lambda x.t)[e]) v, C, D \rangle_{\text{iter}} \Rightarrow \langle t[e'], [], C \cdot D \rangle_{\text{eval}} \quad \text{where } e' = \textit{extend}(x, v, e) \\
& \langle (\ulcorner D \urcorner \circ v') v, C, D \rangle_{\text{iter}} \Rightarrow \langle v v', [], D' \rangle_{\text{iter}} \\
& \langle \ulcorner D \urcorner v, C, D \rangle_{\text{iter}} \Rightarrow \langle C, \ulcorner D \urcorner \circ v, D \rangle_{\text{cont}} \\
& \langle [], v, D \rangle_{\text{cont}} \Rightarrow \langle D, v \rangle_{\text{dump}} \\
& \langle C[(t[e]) []], v, D \rangle_{\text{cont}} \Rightarrow \langle t[e], C[[] v], D \rangle_{\text{eval}} \\
& \langle C[[] v'], v, D \rangle_{\text{cont}} \Rightarrow \langle v v', C, D \rangle_{\text{iter}} \\
& \langle \bullet, v \rangle_{\text{dump}} \Rightarrow v \\
& \langle C \cdot D, v \rangle_{\text{dump}} \Rightarrow \langle C, v, D \rangle_{\text{cont}}
\end{aligned}$$

The eval-clauses for $\ulcorner n \urcorner$, *SUCC* (which only occurs in the initial environment), $c_0 c_1$, $\ulcorner D \urcorner$, and $\ulcorner D \urcorner \circ v$ and the iter-clauses for $x[e]$, $(t_0 t_1)[e]$, and *J* all have disappeared: they were only transitory. The eval-clause for $\langle c \rangle$ has also disappeared: it is a dead clause here since *plug* has been refocused away.

Proposition 7.8. *For any closure c , $\textit{evaluate}(c) = v$ if and only if $\langle c, [], \bullet \rangle_{\text{eval}} \Rightarrow^* v$.*

Proof: immediate. We have merely compressed corridor transitions and removed one dead clause. \square

7.4.4. *Opening closures: from explicit substitutions to terms and environments.* The abstract machine above solely operates on ground closures and the iter-clauses solely dispatch on applications of one value to another. If we (1) open the closures $t[e]$ into pairs (t, e) and flatten the configuration $\langle (t, e), C, D \rangle_{\text{eval}}$ into a quadruple $\langle t, e, C, D \rangle_{\text{eval}}$ and (2) flatten the configuration $\langle v v', C, D \rangle_{\text{iter}}$ into a quadruple $\langle v, v', C, D \rangle_{\text{apply}}$, we obtain an abstract machine that coincides with the caller-save, stackless SECD machine of Section 7.1.

The following proposition captures that the SECD machine implements the reduction semantics of Section 7.3.

Proposition 7.9 (syntactic correspondence). *For any program t in the $\lambda\hat{p}J$ -calculus,*

$$t[(\textit{succ}, \textit{SUCC}) \cdot \emptyset] \rightarrow^* v \quad \text{if and only if} \quad \langle t[(\textit{succ}, \textit{SUCC}) \cdot \emptyset], [], \bullet \rangle_{\text{eval}} \Rightarrow^* v.$$

Proof: this proposition is a simple corollary of the above series of propositions and of the observation just above. \square

7.5. Summary and conclusion. All in all, the syntactic and the functional correspondences provide a method to mechanically build compatible small-step semantics in the form of calculi (reduction semantics) and abstract machines, and big-step semantics in the form of evaluation functions. We have illustrated this method here for applicative expressions with the J operator, providing their first big-step semantics and their first reduction semantics.

8. A SYNTACTIC THEORY OF APPLICATIVE EXPRESSIONS WITH THE J OPERATOR: IMPLICIT, CALLER-SAVE DUMPS

The J operator capture the continuation of the caller and accordingly, the SECD machine is structured as the expression continuation of the current function up to its point of call (the C component) and as a list of the delimited expression continuations of the previously called functions (the D component). This architecture stands both for the original SECD machine (Section 2) and for its modernized instances, whether the dump is managed in a callee-save fashion (Section 3) or in a caller-save fashion (Section 4). In this section, we study a single representation of the context that is dynamically scanned in search for the context of the caller, as in Felleisen et al.'s initial take on delimited continuations [54] and in John Clements's PhD thesis work on continuation marks [27]. We start from a reduction semantics (Section 8.1) and refocus it into an abstract machine (Section 8.2).

8.1. Reduction semantics. We specify the reduction semantics as in Sections 7.3 and E.1, i.e., with its syntactic categories, a plugging function, a notion of contraction, a decomposition function, a one-step reduction function, and a reduction-based evaluation function.

8.1.1. Syntactic categories. We consider a variant of the $\lambda\hat{\rho}J$ -calculus with one layer of context C and with delimiters $\langle c \rangle$ and $\langle C \rangle$ (shaded below) to mark the boundary between the context of a β -redex that has been contracted, i.e., a function closure that has been applied, and the body of the λ -abstraction in this function closure which is undergoing reduction:

$$\begin{aligned}
 (\text{programs}) \quad p &::= t[(succ, SUCC) \cdot \emptyset] \\
 (\text{terms}) \quad t &::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid tt \mid J \\
 (\text{closures}) \quad c &::= \ulcorner n \urcorner \mid SUCC \mid t[e] \mid cc \mid \ulcorner C \urcorner \mid \ulcorner C \urcorner \circ v \mid \langle c \rangle \\
 (\text{values}) \quad v &::= \ulcorner n \urcorner \mid SUCC \mid (\lambda x.t)[e] \mid \ulcorner C \urcorner \mid \ulcorner C \urcorner \circ v \\
 (\text{potential redexes}) \quad r &::= x[e] \mid vv \mid J \\
 (\text{substitutions}) \quad e &::= \emptyset \mid (x, v) \cdot e \\
 (\text{contexts}) \quad C &::= [] \mid C[c []] \mid C[[] v] \mid \langle C \rangle
 \end{aligned}$$

Again, in the syntactic category of closures, $\ulcorner C \urcorner$ and $\ulcorner C \urcorner \circ v$ respectively denote a state appender and a program closure. Also again, values are therefore a syntactic subcategory of closures, and we make use of the syntactic coercion \uparrow mapping a value into a closure.

8.1.2. *Plugging.* Plugging a closure in a context is defined by induction over this context:

$$\begin{aligned}
\langle [], c \rangle_{\text{plug/cont}} &\rightarrow c \\
\langle C[c_0 [], c_1] \rangle_{\text{plug/cont}} &\rightarrow \langle C, c_0 c_1 \rangle_{\text{plug/cont}} \\
\langle C[[] v_1], c_0 \rangle_{\text{plug/cont}} &\rightarrow \langle C, c_0 c_1 \rangle_{\text{plug/cont}} \quad \text{where } c_1 = \uparrow v_1 \\
\langle \langle C \rangle, c \rangle_{\text{plug/cont}} &\rightarrow \langle C, \langle c \rangle \rangle_{\text{plug/cont}}
\end{aligned}$$

Definition 8.1. For any closure c and context C , $\text{plug}(C, c) = c'$ if and only if $\langle C, c \rangle_{\text{plug/cont}} \rightarrow^* c'$.

8.1.3. *Notion of contraction.* The notion of reduction is specified by the following context-sensitive contraction rules over actual redexes:

$$\begin{aligned}
(\text{Var}) \quad \langle x[e], C \rangle &\mapsto \langle v, C \rangle && \text{if } \text{lookup}(x, e) = v \\
(\text{Beta}_{\text{succ}}) \quad \langle \text{SUCC} \ulcorner n \urcorner, C \rangle &\mapsto \langle \ulcorner n + 1 \urcorner, C \rangle \\
(\text{Beta}_{FC}) \quad \langle ((\lambda x.t)[e]) v, C \rangle &\mapsto \langle \langle t[e'] \rangle, C \rangle && \text{where } e' = \text{extend}(x, v, e) = (x, v) \cdot e \\
(\text{Beta}_{SA}) \quad \langle \ulcorner C \urcorner v, C \rangle &\mapsto \langle \ulcorner C \urcorner \circ v, C \rangle \\
(\text{Beta}_{PC}) \quad \langle (\ulcorner C \urcorner \circ v') v, C \rangle &\mapsto \langle v' v, C' \rangle \\
(\text{Prop}) \quad \langle (t_0 t_1)[e], C \rangle &\mapsto \langle (t_0[e]) (t_1[e]), C \rangle \\
(J) \quad \langle J, C \rangle &\mapsto \langle \ulcorner C \urcorner, C \rangle && \text{where } C' = \text{previous}(C)
\end{aligned}$$

where *previous* maps a context to its most recent delimited context, if any:

$$\begin{aligned}
\text{previous}(C[c []]) &= \text{previous}(C) \\
\text{previous}(C[[] v]) &= \text{previous}(C) \\
\text{previous}(\langle C \rangle) &= C
\end{aligned}$$

Two of the contraction rules depend on the context: the J rule captures a copy of the context of the most recent caller and yields a state appender, and the β -rule for program closures reinstates a previously captured copy of the context. As for the β -rule for function closures, it introduces a delimiter.

Definition 8.2. For any potential redex r and context C , $\text{contract}(r, C) = \langle c, C' \rangle$ if and only if $\langle r, C' \rangle \mapsto \langle c, C' \rangle$.

8.1.4. *Decomposition.* Decomposition is essentially as in Section 7.3.4, except that there is no explicit dump component:

$$\begin{aligned}
\langle \ulcorner n \urcorner, C \rangle_{\text{dec/clos}} &\rightarrow \langle C, \ulcorner n \urcorner \rangle_{\text{dec/cont}} \\
\langle \text{SUCC}, C \rangle_{\text{dec/clos}} &\rightarrow \langle C, \text{SUCC} \rangle_{\text{dec/cont}} \\
\langle \ulcorner n \urcorner[e], C \rangle_{\text{dec/clos}} &\rightarrow \langle C, \ulcorner n \urcorner \rangle_{\text{dec/cont}} \\
\langle x[e], C \rangle_{\text{dec/clos}} &\rightarrow \text{DEC}(x[e], C) \\
\langle ((\lambda x.t)[e]), C \rangle_{\text{dec/clos}} &\rightarrow \langle C, (\lambda x.t)[e] \rangle_{\text{dec/cont}} \\
\langle (t_0 t_1)[e], C \rangle_{\text{dec/clos}} &\rightarrow \text{DEC}((t_0 t_1)[e], C) \\
\langle J[e], C \rangle_{\text{dec/clos}} &\rightarrow \text{DEC}(J, C) \\
\langle c_0 c_1, C \rangle_{\text{dec/clos}} &\rightarrow \langle c_1, C[c_0 []] \rangle_{\text{dec/clos}}
\end{aligned}$$

$$\begin{aligned}
\langle \ulcorner C' \urcorner, C \rangle_{\text{dec/clos}} &\rightarrow \langle C, \ulcorner C' \urcorner \rangle_{\text{dec/cont}} \\
\langle \ulcorner C' \urcorner \circ v, C \rangle_{\text{dec/clos}} &\rightarrow \langle C, \ulcorner C' \urcorner \circ v \rangle_{\text{dec/cont}} \\
\langle \langle c \rangle, C \rangle_{\text{dec/clos}} &\rightarrow \langle c, \langle C \rangle \rangle_{\text{dec/clos}} \\
\langle [], v \rangle_{\text{dec/cont}} &\rightarrow \text{VAL}(v) \\
\langle C[c_0 [], v_1] \rangle_{\text{dec/cont}} &\rightarrow \langle c_0, C[[] v_1] \rangle_{\text{dec/clos}} \\
\langle C[[] v_1], v_0 \rangle_{\text{dec/cont}} &\rightarrow \text{DEC}(v_0 v_1, C) \\
\langle \langle C \rangle, v \rangle_{\text{dec/cont}} &\rightarrow \langle C, v \rangle_{\text{dec/cont}}
\end{aligned}$$

Definition 8.3. For any closure c ,

$$\text{decompose}(c) = \begin{cases} \text{VAL}(v) & \text{if } \langle c, [], [] \rangle_{\text{dec/clos}} \rightarrow^* \text{VAL}(v) \\ \text{DEC}(r, C) & \text{if } \langle c, [], [] \rangle_{\text{dec/clos}} \rightarrow^* \text{DEC}(r, C) \end{cases}$$

8.1.5. *One-step reduction and reduction-based evaluation.* We are now in position to define a one-step reduction function (as in Sections 7.3.5 and E.1.5) and an evaluation function iterating this reduction function (as in Section 7.3.6 and E.1.6).

8.2. **From reduction semantics to abstract machine.** Repeating mutatis mutandis the derivation illustrated in Sections 7.4 and E.2 leads one to the following variant of the SECD machine:

$$\begin{aligned}
(\text{programs}) \quad p &::= t[(\text{succ}, \text{SUCC}) \cdot \emptyset] \\
(\text{terms}) \quad t &::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid tt \mid J \\
(\text{values}) \quad v &::= \ulcorner n \urcorner \mid \text{SUCC} \mid (\lambda x.t, e) \mid \ulcorner C' \urcorner \circ v \mid \ulcorner C' \urcorner \\
(\text{environments}) \quad e &::= \emptyset \mid (x, v) \cdot e \\
(\text{contexts}) \quad C &::= [] \mid C[(t, e) []] \mid C[[] v] \mid \langle C \rangle \\
\langle \ulcorner n \urcorner, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner n \urcorner \rangle_{\text{cont}} \\
\langle x, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, v \rangle_{\text{cont}} && \text{if } \text{lookup}(x, e) = v \\
\langle \lambda x.t, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t, e) \rangle_{\text{cont}} \\
\langle t_0 t_1, e, C \rangle_{\text{eval}} &\Rightarrow \langle t_1, e, C[(t_0, e) []] \rangle_{\text{eval}} \\
\langle J, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, \ulcorner C' \urcorner \rangle_{\text{cont}} && \text{if } C' = \text{previous}(C) \\
\langle \text{SUCC}, \ulcorner n \urcorner, C \rangle_{\text{apply}} &\Rightarrow \langle C, \ulcorner n + 1 \urcorner \rangle_{\text{cont}} \\
\langle (\lambda x.t, e), v, C \rangle_{\text{apply}} &\Rightarrow \langle t, e', \langle C \rangle \rangle_{\text{eval}} && \text{where } e' = \text{extend}(x, v, e) \\
\langle \ulcorner C' \urcorner \circ v', v, C \rangle_{\text{apply}} &\Rightarrow \langle v, v', C' \rangle_{\text{apply}} \\
\langle \ulcorner C' \urcorner, v, C \rangle_{\text{apply}} &\Rightarrow \langle C, \ulcorner C' \urcorner \circ v \rangle_{\text{cont}} \\
\langle [], v \rangle_{\text{cont}} &\Rightarrow v \\
\langle C[(t, e) []], v \rangle_{\text{cont}} &\Rightarrow \langle t, e, C[[] v] \rangle_{\text{eval}} \\
\langle C[[] v'], v \rangle_{\text{cont}} &\Rightarrow \langle v, v', C \rangle_{\text{apply}} \\
\langle \langle C \rangle, v \rangle_{\text{cont}} &\Rightarrow \langle C, v \rangle_{\text{cont}}
\end{aligned}$$

Starting in the configuration $\langle t, (\text{succ}, \text{SUCC}) \cdot \emptyset, [] \rangle_{\text{eval}}$ makes this machine evaluate the program t . The machine halts with a value v if it reaches a configuration $\langle [], v \rangle_{\text{cont}}$.

Alternatively (if we allow J to be used outside the body of a lambda-term and we let it denote the empty context), this machine evaluates a program t by starting in the configuration $\langle t, (succ, SUCC) \cdot \emptyset, \langle \langle \rangle \rangle_{eval} \rangle$. It halts with a value v if it reaches a configuration $\langle \langle \rangle, v \rangle_{cont}$.

In either case, the machine is not in defunctionalized form [43, 44]. Therefore, one cannot immediately map it into an evaluation function in CPS, as in Sections 2, 3, and 4. The next two sections present two alternatives, each of which is in defunctionalized form and operates in lockstep with the present abstract machine.

9. A SYNTACTIC THEORY OF APPLICATIVE EXPRESSIONS WITH THE J OPERATOR: EXPLICIT, CALLER-SAVE DUMPS

Instead of marking the context and the intermediate closures, as in Section 8, one can cache the context of the caller in a separate register, which leads one towards `evaluate1'_alt` in Section 4.2. For an analogy, in some formal specifications of Prolog [17, 49], the cut continuation denotes the previous failure continuation and is cached in a separate register.

10. A SYNTACTIC THEORY OF APPLICATIVE EXPRESSIONS WITH THE J OPERATOR: INHERITING THE DUMP THROUGH THE ENVIRONMENT

Instead of marking the context and the intermediate closures, as in Section 8, or of caching the context of the caller in a separate register, as in Section 9, one can cache the context of the caller in the environment, which leads one towards Felleisen's simulation (Section 4.5) and a lightweight extension of the CEK machine. Let us briefly outline this reduction semantics and this abstract machine.

10.1. Reduction semantics. We specify the reduction semantics as in Section 8.1.

10.1.1. Syntactic categories. We consider a variant of the $\lambda\hat{\rho}J$ -calculus which is essentially that of Section 8.1.1, except that J is now an identifier and there are no delimiters:

$$\begin{aligned}
(\text{programs}) \quad p &::= t[(succ, SUCC) \cdot \emptyset] \\
(\text{terms}) \quad t &::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t t \\
(\text{closures}) \quad c &::= \ulcorner n \urcorner \mid SUCC \mid t[e] \mid c c \mid \ulcorner C \urcorner \mid \ulcorner C \urcorner \circ v \\
(\text{values}) \quad v &::= \ulcorner n \urcorner \mid SUCC \mid (\lambda x.t)[e] \mid \ulcorner C \urcorner \mid \ulcorner C \urcorner \circ v \\
(\text{potential redexes}) \quad r &::= x[e] \mid v v \\
(\text{substitutions}) \quad e &::= \emptyset \mid (x, v) \cdot e \\
(\text{contexts}) \quad C &::= \langle \rangle \mid C[c \langle \rangle] \mid C[\langle \rangle v]
\end{aligned}$$

10.1.2. Plugging. The notion of reduction is essentially as that of Section 8.1.2, except that there is no control delimiter:

$$\begin{aligned}
\langle \langle \rangle, c \rangle_{plug/cont} &\rightarrow c \\
\langle C[c_0 \langle \rangle], c_1 \rangle_{plug/cont} &\rightarrow \langle C, c_0 c_1 \rangle_{plug/cont} \\
\langle C[\langle \rangle v_1], c_0 \rangle_{plug/cont} &\rightarrow \langle C, c_0 c_1 \rangle_{plug/cont} \quad \text{where } c_1 = \uparrow v_1
\end{aligned}$$

10.1.3. *Notion of contraction.* The notion of reduction is essentially as that of Section 8.1.3, except that there is no rule for J and there are no delimiters:

$$\begin{array}{lll}
(\text{Var}) & \langle x[e], C \rangle \mapsto \langle v, C \rangle & \text{if } \textit{lookup}(x, e) = v \\
(\text{Beta}_{succ}) & \langle SUCC \ulcorner n \urcorner, C \rangle \mapsto \langle \ulcorner n + 1 \urcorner, C \rangle \\
(\text{Beta}_{FC}) & \langle ((\lambda x.t)[e]) v, C \rangle \mapsto \langle t[e'], C \rangle & \text{where } e' = (J, \ulcorner C \urcorner) \cdot (x, v) \cdot e \\
(\text{Beta}_{SA}) & \langle \ulcorner C \urcorner v, C \rangle \mapsto \langle \ulcorner C \urcorner \circ v, C \rangle \\
(\text{Beta}_{PC}) & \langle (\ulcorner C \urcorner \circ v') v, C \rangle \mapsto \langle v' v, C' \rangle \\
(\text{Prop}) & \langle (t_0 t_1)[e], C \rangle \mapsto \langle (t_0[e]) (t_1[e]), C \rangle
\end{array}$$

In the β -rule for function closures, J is dynamically bound to the current context.

10.1.4. *Decomposition.* Decomposition is essentially as in Section 8.1.4, except that there is no rule for J and there are no delimiters:

$$\begin{array}{ll}
\langle \ulcorner n \urcorner, C \rangle_{\text{dec/clos}} & \rightarrow \langle C, \ulcorner n \urcorner \rangle_{\text{dec/cont}} \\
\langle SUCC, C \rangle_{\text{dec/clos}} & \rightarrow \langle C, SUCC \rangle_{\text{dec/cont}} \\
\langle \ulcorner n \urcorner[e], C \rangle_{\text{dec/clos}} & \rightarrow \langle C, \ulcorner n \urcorner \rangle_{\text{dec/cont}} \\
\langle x[e], C \rangle_{\text{dec/clos}} & \rightarrow \text{DEC}(x[e], C) \\
\langle ((\lambda x.t)[e]), C \rangle_{\text{dec/clos}} & \rightarrow \langle C, (\lambda x.t)[e] \rangle_{\text{dec/cont}} \\
\langle (t_0 t_1)[e], C \rangle_{\text{dec/clos}} & \rightarrow \text{DEC}((t_0 t_1)[e], C) \\
\langle c_0 c_1, C \rangle_{\text{dec/clos}} & \rightarrow \langle c_1, C[c_0 []] \rangle_{\text{dec/clos}} \\
\langle \ulcorner C \urcorner, C \rangle_{\text{dec/clos}} & \rightarrow \langle C, \ulcorner C \urcorner \rangle_{\text{dec/cont}} \\
\langle \ulcorner C \urcorner \circ v, C \rangle_{\text{dec/clos}} & \rightarrow \langle C, \ulcorner C \urcorner \circ v \rangle_{\text{dec/cont}} \\
\langle [], v \rangle_{\text{dec/cont}} & \rightarrow \text{VAL}(v) \\
\langle C[c_0 []], v_1 \rangle_{\text{dec/cont}} & \rightarrow \langle c_0, C[[] v_1] \rangle_{\text{dec/clos}} \\
\langle C[[] v_1], v_0 \rangle_{\text{dec/cont}} & \rightarrow \text{DEC}(v_0 v_1, C)
\end{array}$$

10.2. **From reduction semantics to abstract machine.** Repeating mutatis mutandis the derivation illustrated in Sections 7.4 and E.2 leads one to the following variant of the CEK machine:

$$\begin{array}{ll}
(\text{programs}) & p ::= t[(succ, SUCC) \cdot \emptyset] \\
(\text{terms}) & t ::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t t \\
(\text{values}) & v ::= \ulcorner n \urcorner \mid SUCC \mid (\lambda x.t, e) \mid \ulcorner C \urcorner \circ v \mid \ulcorner C \urcorner \\
(\text{environments}) & e ::= \emptyset \mid (x, v) \cdot e \\
(\text{contexts}) & C ::= [] \mid C[(t, e) []] \mid C[[] v]
\end{array}$$

$$\begin{aligned}
\langle \lceil n \rceil, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, \lceil n \rceil \rangle_{\text{cont}} \\
\langle x, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, v \rangle_{\text{cont}} && \text{if } \text{lookup}(x, e) = v \\
\langle \lambda x.t, e, C \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t, e) \rangle_{\text{cont}} \\
\langle t_0 t_1, e, C \rangle_{\text{eval}} &\Rightarrow \langle t_1, e, C[(t_0, e) []] \rangle_{\text{eval}} \\
\langle \text{SUCC}, \lceil n \rceil, C \rangle_{\text{apply}} &\Rightarrow \langle C, \lceil n + 1 \rceil \rangle_{\text{cont}} \\
\langle (\lambda x.t, e), v, C \rangle_{\text{apply}} &\Rightarrow \langle t, e', C \rangle_{\text{eval}} && \text{where } e' = \text{extend}(J, \lceil C \rceil, \text{extend}(x, v, e)) \\
\langle \lceil C \rceil \circ v', v, C \rangle_{\text{apply}} &\Rightarrow \langle v, v', C' \rangle_{\text{apply}} \\
\langle \lceil C \rceil, v, C \rangle_{\text{apply}} &\Rightarrow \langle C, \lceil C \rceil \circ v \rangle_{\text{cont}} \\
\langle [], v \rangle_{\text{cont}} &\Rightarrow v \\
\langle C[(t, e) []], v \rangle_{\text{cont}} &\Rightarrow \langle t, e, C[[] v] \rangle_{\text{eval}} \\
\langle C[[] v'], v \rangle_{\text{cont}} &\Rightarrow \langle v, v', C \rangle_{\text{apply}}
\end{aligned}$$

This machine evaluates a program t by starting in the configuration

$$\langle t, (\text{succ}, \text{SUCC}) \cdot \emptyset, [] \rangle_{\text{eval}}.$$

It halts with a value v if it reaches a configuration $\langle [], v \rangle_{\text{cont}}$.

Alternatively (if we allow J to be used outside the body of a lambda-term and we let it denote the empty context), this machine evaluates a program t by starting in the configuration

$$\langle t, (J, []) \cdot (\text{succ}, \text{SUCC}) \cdot \emptyset, [] \rangle_{\text{eval}}.$$

It halts with a value v if it reaches a configuration $\langle [], v \rangle_{\text{cont}}$.

In either case, the machine is in defunctionalized form. Refunctionalizing it yields a continuation-passing evaluation function. Refunctionalizing its closures and mapping the result back to direct style yields the compositional evaluation functions displayed in Section 4.5, i.e., Felleisen's embedding of the J operator in Scheme [51].

11. SUMMARY AND CONCLUSION

We have presented a rational deconstruction of the SECD machine with the J operator, through a series of alternative implementations, in the form of abstract machines and compositional evaluation functions, all of which are new. We have also presented the first syntactic theories of applicative expressions with the J operator. In passing, we have shown new applications of refocusing and defunctionalization and new examples of control delimiters and of both pushy and jumpy delimited continuations in programming practice.

Even though they were the first of their kind, the SECD machine and the J operator remain computationally relevant today:

- Architecturally, and until the advent of JavaScript run-time systems [57], the SECD machine has been superseded by abstract machines with a single control component instead of two (namely C and D). In some JavaScript run-time systems, however, methods have a local stack similar to C to implement and manage their expression continuation, and a global stack similar to D to implement and manage command continuations, i.e., the continuation of their caller.

- Programmatically, and until the advent of first-class continuations in JavaScript [28], the J operator has been superseded by control operators that capture the current continuation (i.e., both C and D) instead of the continuation of the caller (i.e., D). In the Rhino implementation of JavaScript, however, the control operator captures the continuation of the caller of the current method, i.e., the command continuation instead of both the expression continuation and the command continuation.

At any rate, as we have shown here, both the SECD machine and the J operator fit the functional correspondence [3,4,6,7,13,16,35,36] as well as the syntactic correspondence [12,14,15,34,36,45], which made it possible for us to mechanically characterize them in new and precise ways.

All of the points above make us conclude that new abstract machines should be defined in defunctionalized form today, or at least be made to work in lockstep with an abstract machine in defunctionalized form.

12. ON THE ORIGIN OF FIRST-CLASS CONTINUATIONS

We have shown that jumping and labels are not essentially connected with strings of imperatives and in particular, with assignment. Second, that jumping is not essentially connected with labels. In performing this piece of logical analysis we have provided a precisely limited sense in which the “value of a label” has meaning. Also, we have discovered a new language feature, not present in current programming languages, that promises to clarify and simplify a notoriously untidy area of programming—that concerned with success/failure situations, and the actions needed on failure.

– Peter J. Landin, 1965 [82, page 133]

It was Strachey who coined the term “first-class functions” [113, Section 3.5.1].⁸ In turn it was Landin who, through the J operator, invented what we know today as first-class continuations [58]: like Reynolds for escape [102], Landin defined J in an unconstrained way, i.e., with no regard for it to be compatible with the last-in, first-out allocation discipline prevalent for control stacks since Algol 60.⁹

Today, ‘continuation’ is an overloaded term, that may refer

- to the original semantic description technique for representing ‘the meaning of the rest of the program’ as a function, the continuation, as multiply co-discovered in the early 1970’s [103]; or
- to the programming-language feature of first-class continuations as typically provided by a control operator such as J, escape, or call/cc, as invented by Landin.

Whether a semantic description technique or a programming-language feature, the goal of continuations was the same: to formalize Algol’s labels and jumps. But where Wadsworth and Abdali gave a continuation semantics to Algol, and as illustrated in the beginning of Section 1, Landin translated Algol programs into applicative expressions in direct style. In turn, he specified the semantics of applicative expressions with the SECD machine, i.e., using first-order means. The meaning of an Algol label was an ISWIM ‘program closure’

⁸“*Out of Quine’s dictum: To be is to be the value of a variable, grew Strachey’s ‘first-class citizens’.*” Peter J. Landin, 2000 [86, page 75]

⁹“*Dumps and program-closures are data-items, with all the implied latency for unruly multiple use and other privileges of first-class-citizenship.*” Peter J. Landin, 1997 [85, Section 1]

as obtained by the J operator. Program closures were defined by extending the SECD machine, i.e., still using first-order means.

Landin did not use an explicit representation of the rest of the computation in his direct semantics of Algol 60, and for that reason he is not listed among the co-discoverers of continuations [103]. Such an explicit representation, however, exists in the SECD machine, in first-order form—the dump—which represents the rest of the computation after returning from the current function call.

In an earlier work [35], Danvy has shown that the SECD machine, even though it is first-order, directly corresponds to a compositional evaluation function in CPS—the tool of choice for specifying control operators since Reynolds’s work [102]. In particular, the dump directly corresponds to a functional representation of control, since it is a defunctionalized continuation. In the light of defunctionalization, Landin therefore did use an explicit representation of the rest of the computation that corresponds to a function, and for that reason we wish to see his name added to the list of co-discoverers of continuations.

ACKNOWLEDGMENTS

Thanks are due to Małgorzata Biernacka, Dariusz Biernacki, Julia L. Lawall, Johan Munk, Kristian Støvring, and the anonymous reviewers of IFL’05 and LMCS for comments. We are also grateful to Andrzej Filinski, Dan Friedman, Lockwood Morris, John Reynolds, Guy Steele, Carolyn Talcott, Bob Tennent, Hayo Thielecke, and Chris Wadsworth for their feedback on Section 12 in November 2005.

This work was partly carried out while the two authors visited the TOPPS group at DIKU (<http://www.diku.dk/topps>). It is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

Appendices

Appendix A demonstrates how two programs, before and after defunctionalization, do not just yield the same result but also operate in lockstep. The three following appendices illustrate the callee-save, stack-threading features of the evaluator corresponding to the SECD machine by contrasting them with a caller-save, stackless evaluator for the pure λ -calculus. We successively consider a caller-save, stackless evaluator and the corresponding abstract machine (Appendix B), a callee-save, stackless evaluator and the corresponding abstract machine (Appendix C), and a caller-save, stack-threading evaluator and the corresponding abstract machine (Appendix D). Finally, Appendix E demonstrates how to go from a reduction semantics of the $\lambda\hat{\rho}$ -calculus to the CEK machine.

APPENDIX A. DEFUNCTIONALIZING A CONTINUATION-PASSING VERSION OF THE FIBONACCI FUNCTION

We start with the traditional Fibonacci function in direct style (Section A.1), and then present its continuation-passing counterpart before (Section A.2) and after (Section A.3) defunctionalization. To pinpoint that these two functions operate in lockstep, we equip them with a trace recording their calling sequence, and we show that they yield the same

result and the same trace. (One can use the same tracing technique to prove Proposition 2.2 in Section 2.2.)

A.1. The traditional Fibonacci function. We start from the traditional definition of the Fibonacci function in ML:

```
fun fib n
  = if n <= 1
    then n
    else (fib (n - 1)) + (fib (n - 2))

fun main0 n
  = fib n
```

So for example, evaluating `main0 5` yields 5.

A.2. The Fibonacci function in CPS. To CPS-transform, we first name all intermediate results and sequentialize their computation, assuming a left-to-right order of evaluation [32]:

```
fun fib n
  = if n <= 1
    then n
    else let val v1 = fib (n - 1)
           val v2 = fib (n - 2)
         in v1 + v2
        end

fun main0' n
  = let val v = fib n
    in v
    end
```

We then give `fib` an extra argument, the continuation:

```
fun fib_c (n, k)
  = if n <= 1
    then k n
    else fib_c (n - 1,
               fn v1 => fib_c (n - 2,
                              fn v2 => k (v1 + v2)))

fun main1 n
  = fib_c (n, fn v => v)
```

So for example, evaluating `main1 5` yields 5.

A.3. The Fibonacci function in CPS, defunctionalized. To defunctionalize the Fibonacci function in CPS, we consider its continuation, which has type `int -> int`. Each inhabitant of this function space arises as an instance of the initial continuation in `main1` or of the two continuations in `fib_c`. We therefore represent the function space as a sum with three summands, one for each λ -abstraction, and we interpret each summand with the body of each of these λ -abstractions, using `apply_cont`:


```

type res = int

datatype cont = C0
              | C1 of res * cont
              | C2 of int * cont

fun apply_cont (C0, v)
  = v
  | apply_cont (C1 (v1, c), v2)
  = apply_cont (c, v1 + v2)
  | apply_cont (C2 (n, c), v1)
  = fib_c_def (n - 2, C1 (v1, c))
and fib_c_def (n, c)
  = if n <= 1
    then apply_cont (c, n)
    else fib_c_def (n - 1, C2 (n, c))

fun main2 n
  = fib_c_def (n, C0)

```

Defunctionalization is summarized with the following two tables, the first one for the function abstractions and the corresponding sum injections into the data type `cont`,¹⁰ and the second one for the function applications and the corresponding calls to the `apply` function dispatching over summands:

- introduction

function abstraction	sum injection
<code>fn v => v</code>	<code>C0</code>
<code>fn v2 => k (v1 + v2)</code>	<code>C1 (v1, c)</code>
<code>fn v1 => fib_c (n - 2, fn v2 => k (v1 + v2))</code>	<code>C2 (n, c)</code>

- elimination

function application	case dispatch
<code>k n</code>	<code>apply_cont (c, n)</code>
<code>k (v1 + v2)</code>	<code>apply_cont (c, v1 + v2)</code>

So for example, evaluating `main2 5` yields 5.

A.4. The Fibonacci function in CPS with a trace. We can easily show that applying `main1` and `main2` as defined above to the same integer yields the same result, but we want to show a stronger property, namely that they operate in lockstep. To this end, we equip `fib_c` with a trace recording its calls with the value of its first argument. (It would be simple to trace its returns as well, i.e., the calls to the continuation.)

Representing the trace as a list, the Fibonacci function in CPS reads as follows:

```

type res = int

(* fib_c : int * (res * int list -> 'a) -> 'a *)
fun fib_c (n, k, T)
  = if n <= 1
    then k (n, T)

```

¹⁰Which the cognoscenti will recognize as Daniel P. Friedman’s “data-structure continuations” [59, 119].

```

    else fib_c (n - 1,
               fn (v1, T) => fib_c (n - 2,
                                   fn (v2, T) => k (v1 + v2, T),
                                   (n - 2) :: T),
               (n - 1) :: T)

(* main3 : int -> res * int list *)
fun main3 n
  = fib_c (n, fn (v, T) => (v, T), n :: nil)

```

So for example, evaluating `main3 5` yields `(5, [1,0,1,2,3,0,1,2,1,0,1,2,3,4,5])`.

A.5. The Fibonacci function in CPS with a trace, defunctionalized. Proceeding as in Section A.3, the corresponding defunctionalized version reads as follows; `fib_c_def` is equipped with a trace recording its calls with the value of its first argument. (Its returns, i.e., the calls to `apply_cont`, could be traced as well.)

```

type res = int

datatype cont = C0
              | C1 of res * cont
              | C2 of int * cont

(* apply_cont : cont * res * int list -> res * int list *)
fun apply_cont (C0, v, T)
  = (v, T)
  | apply_cont (C1 (v1, c), v2, T)
  = apply_cont (c, v1 + v2, T)
  | apply_cont (C2 (n, c), v1, T)
  = fib_c_def (n - 2, C1 (v1, c), (n - 2) :: T)
(* fib_c_def : int * cont * int list -> res * int list *)
and fib_c_def (n, c, T)
  = if n <= 1
    then apply_cont (c, n, T)
    else fib_c_def (n - 1, C2 (n, c), (n - 1) :: T)

(* main4 : int -> res * int list *)
fun main4 n
  = fib_c_def (n, C0, n :: nil)

```

So for example, evaluating `main4 5` yields `(5, [1,0,1,2,3,0,1,2,1,0,1,2,3,4,5])`.

A.6. Lockstep correspondence.

Definition A.1. We define $\mathcal{R}(k, c)$ as

$$\forall v. \forall T. k(v, T) = a \Leftrightarrow \text{apply_cont}(c, v, T) = a$$

where “ $e = a$ ” means “there exists an ML value a such that evaluating the ML expression e yields a .”

Lemma A.2. $\mathcal{R}(\text{fn } (v, T) \Rightarrow (v, T), C0)$

Proof: immediate. □

Lemma A.3. $\forall v1. \forall k \wedge c$ such that $\mathcal{R}(k, c). \mathcal{R}(\text{fn } (v2, T) \Rightarrow k (v1 + v2, T), C1 (v1, c))$.

Proof:

By β_v reduction, $(\text{fn } (v2, T) \Rightarrow k (v1 + v2, T)) (v2, T)$ yields the same value as $k (v1 + v2, T)$.

By definition, $\text{apply_cont } (C1 (v1, c), v2, T)$ yields the same value as $\text{apply_cont } (c, v1 + v2, T)$.

Suppose that $k (v1 + v2, T) = a$ holds. Then since $\mathcal{R}(k, c)$, $\text{apply_cont } (c, v1 + v2, T) = a$ also holds, and vice-versa. \square

Lemma A.4. $\forall n. \forall k \wedge c$ such that $\mathcal{R}(k, c)$.

1. $\text{fib_c } (n, k, T) = a \Leftrightarrow \text{fib_c_def } (n, c, T) = a$
2. $\mathcal{R}(\text{fn } (v1, T) \Rightarrow \text{fib_c } (n, \text{fn } (v2, T) \Rightarrow k (v1 + v2, T), n :: T), C2 (n+2, c))$

Proof: by simultaneous course-of-value induction. \square

Theorem A.5. $\forall n. \text{main3 } n = a \Leftrightarrow \text{main4 } n = a$

Proof. a consequence of Lemmas A.2 and A.4. \square

The two versions, before and after defunctionalization, therefore operate in lockstep, since they yield the same trace and the same result.

APPENDIX B. A CALLER-SAVE, STACKLESS EVALUATOR AND THE CORRESPONDING ABSTRACT MACHINE

B.1. The evaluator. The following evaluator for the pure call-by-value λ -calculus (i.e., the language of Section 1.5 without constants and the J operator) is standard. As pointed out by Reynolds [102], it depends on the evaluation order of its metalanguage (here, call by value):

```

datatype value = FUN of value -> value

(* eval : term * value Env.env -> value *)
fun eval (VAR x, e)
  = Env.lookup (x, e)
  | eval (LAM (x, t), e)
  = FUN (fn v => eval (t, Env.extend (x, v, e)))
  | eval (APP (t0, t1), e)
  = let val (FUN f) = eval (t0, e)
      in f (eval (t1, e))
      end

fun evaluate t
  = eval (t, Env.mt)

```

The evaluator is stackless because it does not thread any data stack. It is also caller-save because in the clause for applications, when t_0 is evaluated, the environment is implicitly saved in the context in order to evaluate t_1 later on. In other words, the environment is solely an inherited attribute.

B.2. The abstract machine. As initiated by Reynolds [4, 102], closure-converting the data values of an evaluator, CPS transforming its control flow, and defunctionalizing its continuations yields an abstract machine. For the evaluator above, this machine is the CEK machine [53], i.e., an eval-continue abstract machine where the evaluation contexts and the continue transition function are the defunctionalized counterparts of the continuations of the evaluator just above:

$$\begin{aligned}
& \text{(terms)} \quad t ::= x \mid \lambda x.t \mid tt \\
& \text{(values)} \quad v ::= [x, t, e] \\
& \text{(environments)} \quad e ::= \emptyset \mid (x, v) \cdot e \\
& \text{(contexts)} \quad k ::= \text{END} \mid \text{ARG}(t, e, k) \mid \text{FUN}(v, k) \\
& \langle x, e, k \rangle_{\text{eval}} \Rightarrow \langle k, v \rangle_{\text{cont}} \quad \text{if } \text{lookup}(x, e) = v \\
& \langle \lambda x.t, e, k \rangle_{\text{eval}} \Rightarrow \langle k, [x, t, e] \rangle_{\text{cont}} \\
& \langle t_0 t_1, e, k \rangle_{\text{eval}} \Rightarrow \langle t_0, e, \text{ARG}(t_1, e, k) \rangle_{\text{eval}} \\
& \langle \text{END}, v \rangle_{\text{cont}} \Rightarrow v \\
& \langle \text{ARG}(t, e, k), v \rangle_{\text{cont}} \Rightarrow \langle t, e, \text{FUN}(v, k) \rangle_{\text{eval}} \\
& \langle \text{FUN}([x, t, e], k), v \rangle_{\text{cont}} \Rightarrow \langle t, e', k \rangle_{\text{eval}} \quad \text{where } e' = \text{extend}(x, v, e)
\end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle t, \emptyset, \text{END} \rangle_{\text{eval}}$. It halts with a value v if it reaches a configuration $\langle \text{END}, v \rangle_{\text{cont}}$.

APPENDIX C. A CALLEE-SAVE, STACKLESS EVALUATOR AND THE CORRESPONDING ABSTRACT MACHINE

C.1. The evaluator. The following evaluator is a callee-save version of the evaluator of Appendix B. Whereas the evaluator of Appendix B maps a term and an environment to the corresponding value, this evaluator maps a term and an environment to the corresponding value *and the environment*. This way, in the clause for applications, the environment does not need to be implicitly saved since it is explicitly returned together with the value of t_0 . In other words, the environment is not solely an inherited attribute as in the evaluator of Appendix B: it is a synthesized attribute as well.

Functional values are passed the environment of their caller, and eventually they return it. The body of function abstractions is still evaluated in an extended lexical environment, which is returned but then discarded. Otherwise, environments are threaded through the evaluator as inherited attributes:

```

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

(* eval : term * value Env.env -> value * value Env.env *)
fun eval (VAR x, e)
  = (Env.lookup (x, e), e)
  | eval (LAM (x, t), e)
  = (FUN (fn (v0, e0) => let val (v1, e1) = eval (t, Env.extend (x, v0, e))
                        in (v1, e0) end),
    e)

```

```

| eval (APP (t0, t1), e)
  = let val (FUN f, e0) = eval (t0, e)
        val (v, e1) = eval (t1, e0)
        in f (v, e1) end

fun evaluate t
  = let val (v, e) = eval (t, Env.mt)
      in v end

```

Operationally, one may wish to note that unlike the evaluator of Appendix B, this evaluator is not properly tail recursive since the evaluation of the body of a function abstraction no longer occurs in tail position [30, 101].

C.2. The abstract machine. As in Appendix B, closure-converting the data values of this evaluator, CPS-transforming its control flow, and defunctionalizing its continuations yields an abstract machine. This machine is a variant of the CEK machine with callee-save environments; its terms, values, and environments remain the same:

$$\begin{aligned}
(\text{contexts}) \quad k ::= & \text{END} \mid \text{ARG}(t, k) \mid \text{FUN}(v, k) \mid \text{RET}(e, k) \\
\langle x, e, k \rangle_{\text{eval}} \Rightarrow_E & \langle k, v, e \rangle_{\text{cont}} && \text{if } \text{lookup}(x, e) = v \\
\langle \lambda x.t, e, k \rangle_{\text{eval}} \Rightarrow_E & \langle k, [x, t, e], e \rangle_{\text{cont}} \\
\langle t_0 t_1, e, k \rangle_{\text{eval}} \Rightarrow_E & \langle t_0, e, \text{ARG}(t_1, k) \rangle_{\text{eval}} \\
\langle \text{END}, v, e \rangle_{\text{cont}} \Rightarrow_E & v \\
\langle \text{ARG}(t, k), v, e \rangle_{\text{cont}} \Rightarrow_E & \langle t, e, \text{FUN}(v, k) \rangle_{\text{eval}} \\
\langle \text{FUN}([x, t, e'], k), v, e \rangle_{\text{cont}} \Rightarrow_E & \langle t, e'', \text{RET}(e, k) \rangle_{\text{eval}} \text{ where } e'' = \text{extend}(x, v, e') \\
\langle \text{RET}(e', k), v, e \rangle_{\text{cont}} \Rightarrow_E & \langle k, v, e' \rangle_{\text{eval}}
\end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle t, \emptyset, \text{END} \rangle_{\text{eval}}$. It halts with a value v if it reaches a configuration $\langle \text{END}, v, e \rangle_{\text{cont}}$.

C.3. Analysis. Compared to the CEK machine in Section B.2, there are two differences in the datatype of contexts and one new transition rule. The first difference is that environments are no longer saved by the caller in ARG contexts. The second difference is that there is an extra context constructor, RET, to represent the continuation of the non-tail call to the evaluator over the body of function abstractions. The new transition interprets a RET constructor by restoring the environment of the caller before returning.

It is simple to construct a bisimulation between this callee-save machine and the CEK machine.

APPENDIX D. A CALLER-SAVE, STACK-THREADING EVALUATOR
AND THE CORRESPONDING ABSTRACT MACHINE

D.1. The evaluator. In a stack-threading evaluator, a data stack stores intermediate values after they have been computed but before they are used. Evaluating an expression leaves its value on top of the data stack. Applications therefore expect to find their argument and function on top of the data stack.¹¹

Several design possibilities arise. First, one can choose between a single global data stack used for all intermediate values (i.e., as in Forth) or one can use a local data stack for each function application (i.e., as in the SECD machine and in the JVM). For the purpose of illustration, we adopt the latter since it matches the design of the SECD machine.

Since there is one local data stack per function application, then this data stack can be chosen to be saved by the caller or by the callee. Though the former design might be more natural, we again adopt the latter in this illustration since it matches the design of the SECD machine.

If there is a local, callee-save data stack, then functional values are passed their argument and a data stack, and return a value and a data stack. One can choose instead to pass the argument to the function on top of the stack and leave the return value on top of the stack (i.e., as in Forth). We adopt this design here, for a local callee-save data stack:

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

(* eval : term * value list * value Env.env -> value *)
fun eval (VAR x, s, e)
  = Env.lookup (x, e) :: s
| eval (LAM (x, t), s, e)
  = FUN (fn (v0 :: s0)
        => let val (v1 :: s1) = eval (t, nil, Env.extend (x, v0, e))
            in (v1 :: s0) end) :: s
| eval (APP (t0, t1), s, e)
  = let val      s0 = eval (t0, s, e)
        val (v :: FUN f :: s1) = eval (t1, s0, e)
        in f (v :: s1) end

fun evaluate t
  = let val (v :: s) = eval (t, nil, Env.mt)
        in v end
```

Functional values are now passed the data stack of their caller and they find their argument on top of it. The body of a function abstraction is evaluated with an empty data stack, and yields a stack with the value of the body on top. This value is returned to the caller on top of its stack.

¹¹If evaluation is left-to-right, the argument will be evaluated after the function and thus will be on top of the data stack. Some shuffling of the stack can be avoided if the evaluation order is right-to-left, as in the SECD machine or the ZINC abstract machine.

D.2. The abstract machine. As in Appendix C, one may wish to note that functions using local callee-save data stacks are not properly tail-recursive, though functions using global or local caller-save data stacks can be made to be.

As in Appendix B and C, closure converting the data values of this evaluator, CPS transforming its control flow, and defunctionalizing its continuations yields an abstract machine. This machine is another variant of the CEK machine with a data stack; its terms, values, and environments remain the same:

$$\begin{aligned}
 (\text{contexts}) \quad k ::= & \text{END} \mid \text{ARG}(t, e, k) \mid \text{FUN}(k) \mid \text{RET}(s, k) \\
 \langle x, s, e, k \rangle_{\text{eval}} \Rightarrow_S & \langle k, v :: s \rangle_{\text{cont}} && \text{if } \text{lookup}(x, e) = v \\
 \langle \lambda x.t, s, e, k \rangle_{\text{eval}} \Rightarrow_S & \langle k, [x, t, e] :: s \rangle_{\text{cont}} \\
 \langle t_0 t_1, s, e, k \rangle_{\text{eval}} \Rightarrow_S & \langle t_0, s, e, \text{ARG}(t_1, e, k) \rangle_{\text{eval}} \\
 \langle \text{END}, v :: s \rangle_{\text{cont}} \Rightarrow_S & v \\
 \langle \text{ARG}(t, e, k), s \rangle_{\text{cont}} \Rightarrow_S & \langle t, s, e, \text{FUN}(k) \rangle_{\text{eval}} \\
 \langle \text{FUN}(k), v :: [x, t, e] :: s \rangle_{\text{cont}} \Rightarrow_S & \langle t, \text{nil}, e', \text{RET}(s, k) \rangle_{\text{eval}} \quad \text{where } e' = \text{extend}(x, v, e) \\
 \langle \text{RET}(s', k), v :: s \rangle_{\text{cont}} \Rightarrow_S & \langle k, v :: s' \rangle_{\text{cont}}
 \end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle t, \text{nil}, \emptyset, \text{END} \rangle_{\text{eval}}$. It halts with a value v if it reaches a configuration $\langle \text{END}, v :: s \rangle_{\text{cont}}$.

D.3. Analysis. Compared to the CEK machine in Section B.2, there are two differences in the datatype of contexts and one new transition rule. The first difference is that intermediate values are no longer saved in **FUN** contexts, since they are stored on the data stack instead. The second difference is that there is an extra context constructor, **RET**, to represent the continuation of the non-tail call to the evaluator over the body of function abstractions (i.e., a continuation that restores the caller's data stack and pushes the function return value on top). The new transition interprets a **RET** constructor by restoring the data stack of the caller and pushing the returned value on top of it before returning.

It is simple to construct a bisimulation between this stack-threading machine and the CEK machine.

APPENDIX E. FROM REDUCTION SEMANTICS TO ABSTRACT MACHINE

As a warmup to Sections 7.3 and 7.4, we present a reduction semantics for applicative expressions (Section E.1) and we derive the CEK machine from this reduction semantics (Section E.2).

E.1. A reduction semantics for applicative expressions. The $\lambda\hat{\rho}$ -calculus is a minimal extension of Curien's original calculus of closures $\lambda\rho$ [31] to make it closed under one-step reduction [14]. We use it here to illustrate how to go from a reduction semantics to an abstract machine. To this end, we present its syntactic categories (Section E.1.1); a plug function mapping a closure and a reduction context into a closure by filling the given context with the given closure (Section E.1.2); a contraction function implementing a context-insensitive notion of reduction (Section E.1.3) and therefore mapping a potential

redex into a contractum; and a decomposition function mapping a non-value term into a potential redex and a reduction context (Section E.1.4). We are then in position to define a one-step reduction function (Section E.1.5) and a reduction-based evaluation function (Section E.1.6).

E.1.1. Syntactic categories. We consider a variant of the $\lambda\hat{\rho}$ -calculus with names instead of de Bruijn indices, and with the usual reduction context C embodying a left-to-right applicative-order reduction strategy.

$$\begin{aligned}
& \text{(terms)} & t & ::= x \mid \lambda x.t \mid t t \\
& \text{(closures)} & c & ::= t[e] \mid c c \\
& \text{(values)} & v & ::= (\lambda x.t)[e] \\
& \text{(potential redexes)} & r & ::= x[e] \mid v v \\
& \text{(substitutions)} & e & ::= \emptyset \mid (x, v) \cdot e \\
& \text{(contexts)} & C & ::= [] \mid C[[] c] \mid C[v []]
\end{aligned}$$

Values are therefore a syntactic subcategory of closures, and in this section, we make use of the syntactic coercion \uparrow mapping a value into a closure.

E.1.2. Plugging. Plugging a closure in a context is defined by induction over this context. We express this definition as a state-transition system with one intermediate state, $\langle c, C \rangle_{\text{plug}}$, an initial state $\langle c, C \rangle_{\text{plug}}$, and a final state c . The transition function incrementally peels off the given control context:

$$\begin{aligned}
\langle [], c \rangle_{\text{plug}} & \rightarrow c \\
\langle C[[] c_1], c_0 \rangle_{\text{plug}} & \rightarrow \langle C, c_0 c_1 \rangle_{\text{plug}} \\
\langle C[v_0 [], c_1] \rangle_{\text{plug}} & \rightarrow \langle C, c_0 c_1 \rangle_{\text{plug}} \quad \text{where } c_0 = \uparrow v_0
\end{aligned}$$

We now can define a total function **plug** over closures and contexts that fills the given closure into the given context:

$$\text{plug} : \text{Closure} \times \text{Control} \rightarrow \text{Closure}$$

Definition E.1. For any closure c and context C , $\text{plug}(C, c) = c'$ if and only if $\langle c, C \rangle_{\text{plug}} \rightarrow^* c'$.

E.1.3. Notion of contraction. The notion of reduction over applicative expressions is specified by the following context-insensitive contraction rules over actual redexes:

$$\begin{aligned}
& \text{(Var)} & x[e] & \mapsto v & \text{if } \text{lookup}(x, e) = v \\
& \text{(Beta)} & ((\lambda x.t)[e]) v & \mapsto t[s'] & \text{where } s' = \text{extend}(x, v, e) = (x, v) \cdot e \\
& \text{(Prop)} & (t_0 t_1)[e] & \mapsto (t_0[e]) (t_1[e])
\end{aligned}$$

For closed closures (i.e., closures with no free variables), all potential redexes are actual ones.

We now can define by cases a total function **contract** that maps a redex to the corresponding contractum:

contract : PotRed \rightarrow Closure

Definition E.2. For any potential redex r , $\text{contract}(r) = c$ if and only if $r \mapsto c$.

E.1.4. *Decomposition.* There are many ways to define a total function mapping a value closure to itself and a non-value closure to a potential redex and a reduction context. In our experience, the following definition is a convenient one. It is a state-transition system with two intermediate states, $\langle c, C \rangle_{\text{dec/clos}}$ and $\langle C, v \rangle_{\text{dec/cont}}$, an initial state $\langle c, [] \rangle_{\text{dec/clos}}$ and two final states VAL (v) and DEC (r, C). If possible, the transition function from the state $\langle c, C \rangle_{\text{dec/clos}}$ decomposes the given closure c and accumulates the corresponding reduction context C . The transition function from the state $\langle C, v \rangle_{\text{dec/cont}}$ dispatches over the given context.

$$\begin{aligned}
\langle x[e], C \rangle_{\text{dec/clos}} &\rightarrow \text{DEC}(x[e], C) \\
\langle (\lambda x.t)[e], C \rangle_{\text{dec/clos}} &\rightarrow \langle C, (\lambda x.t)[e] \rangle_{\text{dec/cont}} \\
\langle (t_0 t_1)[e], C \rangle_{\text{dec/clos}} &\rightarrow \text{DEC}((t_0 t_1)[e], C) \\
\langle c_0 c_1, C \rangle_{\text{dec/clos}} &\rightarrow \langle c_0, C[[] c_1] \rangle_{\text{dec/clos}} \\
\langle [], v \rangle_{\text{dec/cont}} &\rightarrow \text{VAL}(v) \\
\langle C[[] c_1], v_0 \rangle_{\text{dec/cont}} &\rightarrow \langle c_1, C[v_0 []] \rangle_{\text{dec/clos}} \\
\langle C[v_0 []], v_1 \rangle_{\text{dec/cont}} &\rightarrow \text{DEC}(v_0 v_1, C)
\end{aligned}$$

We now can define a total function `decompose` over closures that maps a value closure to itself and a non-value closure to a decomposition into a potential redex, a control context, and a dump context. This total function uses two auxiliary functions `decompose'_{clos}` and `decompose'_{cont}`:

$$\begin{aligned}
\text{decompose} &: \text{Closure} &&\rightarrow \text{Value} + (\text{PotRed} \times \text{Context}) \\
\text{decompose}'_{\text{clos}} &: \text{Closure} \times \text{Context} &&\rightarrow \text{Value} + (\text{PotRed} \times \text{Context}) \\
\text{decompose}'_{\text{cont}} &: \text{Context} \times \text{Value} &&\rightarrow \text{Value} + (\text{PotRed} \times \text{Context})
\end{aligned}$$

Definition E.3. For any closure c , value v , and context C ,

$$\begin{aligned}
\text{decompose}'_{\text{clos}}(c, C) &= \begin{cases} \text{VAL}(v') & \text{if } \langle c, C \rangle_{\text{dec/clos}} \rightarrow^* \text{VAL}(v') \\ \text{DEC}(r, C') & \text{if } \langle c, C \rangle_{\text{dec/clos}} \rightarrow^* \text{DEC}(r, C') \end{cases} \\
\text{decompose}'_{\text{cont}}(C, v) &= \begin{cases} \text{VAL}(v') & \text{if } \langle C, v \rangle_{\text{dec/cont}} \rightarrow^* \text{VAL}(v') \\ \text{DEC}(r, C') & \text{if } \langle C, v \rangle_{\text{dec/cont}} \rightarrow^* \text{DEC}(r, C') \end{cases}
\end{aligned}$$

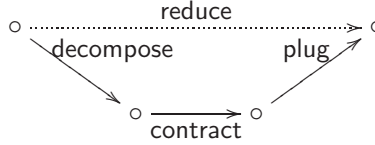
and $\text{decompose}(c) = \text{decompose}'_{\text{clos}}(c, [])$.

E.1.5. *One-step reduction.* We are now in position to define a total function `reduce` over closed closures that maps a value closure to itself and a non-value closure to the next closure in the reduction sequence. This function is defined by composing the three functions above:

$$\begin{aligned}
\text{reduce}(c) &= \text{case } \text{decompose}(c) \\
&\quad \text{of } \text{VAL}(v) &&\Rightarrow \uparrow v \\
&\quad | \text{DEC}(r, C) &&\Rightarrow \text{plug}(\text{contract}(r), C)
\end{aligned}$$

The function `reduce` is partial because of `contract`, which is undefined for stuck closures.

Graphically:



Definition E.4 (One-step reduction). For any closure c , $c \rightarrow c'$ if and only if $\text{reduce}(c) = c'$.

E.1.6. *Reduction-based evaluation.* Iterating `reduce` defines a reduction-based evaluation function. The definition below uses `decompose` to distinguish between values and non-values, and implements iteration (tail-) recursively with the partial function `iterate`:

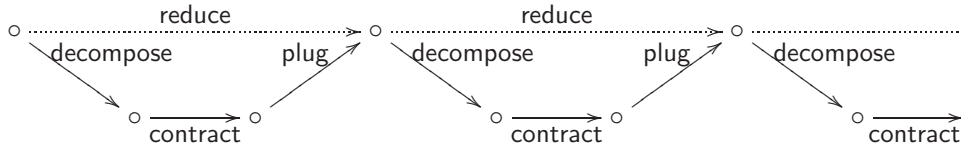
$$\text{evaluate}(c) = \text{iterate}(\text{decompose}(c))$$

where

$$\begin{cases} \text{iterate}(\text{VAL}(v)) & = v \\ \text{iterate}(\text{DEC}(r, C)) & = \text{iterate}(\text{decompose}(\text{plug}(\text{contract}(r), C))) \end{cases}$$

The function `evaluate` is partial because reducing a given closure might not converge.

Graphically:



Definition E.5 (Reduction-based evaluation). For any closure c , $c \rightarrow^* v$ if and only if $\text{evaluate}(c) = v$.

To close, let us adjust the definition of `evaluate` by exploiting the fact that for any closure c , $\text{plug}(c, []) = c$:

$$\text{evaluate}(c) = \text{iterate}(\text{decompose}(\text{plug}(c, [])))$$

In this adjusted definition, `decompose` is always applied to the result of `plug`.

E.2. From the reduction semantics for applicative expressions to the CEK machine. Deforesting the intermediate terms in the reduction-based evaluation function of Section E.1.6 yields a reduction-free evaluation function in the form of a small-step abstract machine (Section E.2.1). We simplify this small-step abstract machine by fusing a part of its driver loop with the contraction function (Section E.2.2) and compressing its ‘corridor’ transitions (Section E.2.3). Unfolding the recursive data type of closures precisely yields the caller-save, stackless CEK machine of Section B.2 (Section E.2.4).

E.2.1. *Refocusing: from reduction-based to reduction-free evaluation.* Following Danvy and Nielsen [45], we deforest the intermediate closure in the reduction sequence by replacing the composition of `plug` and `decompose` by a call to a composite function `refocus`:

$$\text{evaluate}(c) = \text{iterate}(\text{refocus}(c, []))$$

where

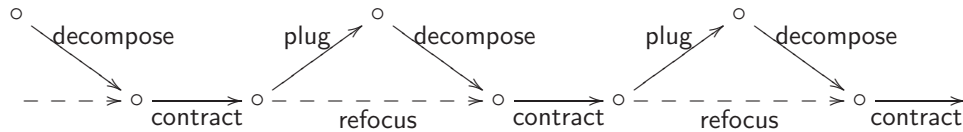
$$\begin{cases} \text{iterate}(\text{VAL}(v)) & = v \\ \text{iterate}(\text{DEC}(r, C)) & = \text{iterate}(\text{refocus}(\text{contract}(r), C)) \end{cases}$$

and `refocus` is optimally defined as continuing the decomposition in the current reduction context [45]:

$$\text{refocus}(c, C) = \text{decompose}'_{\text{clos}}(c, C)$$

This evaluation function is reduction-free because it no longer constructs each intermediate closure in the reduction sequence.

Graphically:



Definition E.6 (Reduction-free evaluation). For any closure c , $c \rightarrow^* v$ if and only if $\text{evaluate}(c) = v$.

E.2.2. *Lightweight fusion: making do without driver loop.* In effect, `iterate` is as the ‘driver loop’ of a small-step abstract machine that refocuses and contracts. Instead, let us fuse `contract` and `iterate` and express the result with rewriting rules over a configuration $\langle r, C \rangle_{\text{iter}}$. We clone the rewriting rules for $\text{decompose}'_{\text{clos}}$ and $\text{decompose}'_{\text{cont}}$ into refocusing rules, indexing their configurations as $\langle c, C \rangle_{\text{eval}}$ and $\langle C, v \rangle_{\text{cont}}$ instead of as $\langle c, C \rangle_{\text{dec/clos}}$ and $\langle C, v \rangle_{\text{dec/cont}}$, respectively:

- instead of rewriting to $\text{VAL}(v)$, the cloned rules rewrite to v ;
- instead of rewriting to $\text{DEC}(r, C)$, the cloned rules rewrite to $\langle r, C \rangle_{\text{iter}}$.

The result reads as follows:

$$\begin{aligned}
\langle x[e], C \rangle_{\text{eval}} &\Rightarrow \langle x[e], C \rangle_{\text{iter}} \\
\langle (\lambda x.t)[e], C \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t)[e] \rangle_{\text{cont}} \\
\langle (t_0 t_1)[e], C \rangle_{\text{eval}} &\Rightarrow \langle (t_0 t_1)[e], C \rangle_{\text{iter}} \\
\langle c_0 c_1, C \rangle_{\text{eval}} &\Rightarrow \langle c_0, C[[] c_1] \rangle_{\text{eval}} \\
\langle [], v \rangle_{\text{cont}} &\Rightarrow v \\
\langle C[[] c_1], v_0 \rangle_{\text{cont}} &\Rightarrow \langle c_1, C[v_0 []] \rangle_{\text{eval}} \\
\langle C[v_0 []], v_1 \rangle_{\text{cont}} &\Rightarrow \langle v_0 v_1, C \rangle_{\text{iter}} \\
\langle x[e], C \rangle_{\text{iter}} &\Rightarrow \langle v, C \rangle_{\text{eval}} && \text{if } \textit{lookup}(x, e) = v \\
\langle ((\lambda x.t)[e]) v, C \rangle_{\text{iter}} &\Rightarrow \langle t[e'], C \rangle_{\text{eval}} && \text{where } e' = \textit{extend}(x, v, e) \\
\langle (t_0 t_1)[e], C \rangle_{\text{iter}} &\Rightarrow \langle (t_0[e]) (t_1[e]), C \rangle_{\text{eval}}
\end{aligned}$$

The following proposition summarizes the situation:

Proposition E.7. *For any closure c , $\text{evaluate}(c) = v$ if and only if $\langle c, [] \rangle_{\text{eval}} \Rightarrow^* v$.*

Proof: straightforward. The two machines operate in lockstep. \square

E.2.3. Inlining and transition compression. The abstract machine of Section E.2.2, while interesting in its own right (it is ‘staged’ in that the contraction rules are implemented separately from the congruence rules [14, 69]), is not minimal: a number of transitions yield a configuration whose transition is uniquely determined. Let us carry out these hereditary, “corridor” transitions once and for all:

- $\langle x[e], C \rangle_{\text{eval}} \Rightarrow \langle x[e], C \rangle_{\text{iter}} \Rightarrow \langle v, C \rangle_{\text{eval}} \Rightarrow \langle C, v \rangle_{\text{cont}}$ if $\textit{lookup}(x, e) = v$
- $\langle (t_0 t_1)[e], C \rangle_{\text{eval}} \Rightarrow \langle (t_0 t_1)[e], C \rangle_{\text{iter}} \Rightarrow \langle (t_0[e]) (t_1[e]), C \rangle_{\text{eval}} \Rightarrow \langle (t_0[e]), C[[] (t_1[e])] \rangle_{\text{eval}}$
- $\langle C[(\lambda x.t)[e] []], v \rangle_{\text{cont}} \Rightarrow \langle ((\lambda x.t)[e]) v, C \rangle_{\text{iter}} \Rightarrow \langle t[e'], C \rangle_{\text{eval}}$ where $e' = \textit{extend}(x, v, e)$

The result reads as follows:

$$\begin{aligned}
\langle x[e], C \rangle_{\text{eval}} &\Rightarrow \langle C, v \rangle_{\text{cont}} && \text{if } \textit{lookup}(x, e) = v \\
\langle (\lambda x.t)[e], C \rangle_{\text{eval}} &\Rightarrow \langle C, (\lambda x.t)[e] \rangle_{\text{cont}} \\
\langle (t_0 t_1)[e], C \rangle_{\text{eval}} &\Rightarrow \langle (t_0[e]), C[[] (t_1[e])] \rangle_{\text{eval}} \\
\langle [], v \rangle_{\text{cont}} &\Rightarrow v \\
\langle C[[] c_1], v_0 \rangle_{\text{cont}} &\Rightarrow \langle c_1, C[v_0 []] \rangle_{\text{eval}} \\
\langle C[(\lambda x.t)[e] []], v \rangle_{\text{cont}} &\Rightarrow \langle t[e'], C \rangle_{\text{eval}} && \text{where } e' = \textit{extend}(x, v, e)
\end{aligned}$$

The configuration $\langle r, C \rangle_{\text{iter}}$ has disappeared and so is the case for $c_0 c_1$: they were only transitory.

Proposition E.8. *For any closure c , $\text{evaluate}(c) = v$ if and only if $\langle c, [] \rangle_{\text{eval}} \Rightarrow^* v$.*

Proof: immediate. We have merely compressed corridor transitions. \square

E.2.4. *Opening closures: from explicit substitutions to terms and environments.* The abstract machine above solely operates on ground closures. If we open the closures $t[e]$ into pairs (t, e) and flatten the configuration $\langle (t, e), C \rangle_{\text{eval}}$ into a triple $\langle t, e, C \rangle_{\text{eval}}$, we obtain an abstract machine that coincides with the caller-save, stackless CEK machine of Section B.2.

E.3. Conclusion and perspectives. Appendix B illustrated the functional correspondence between the functional implementation of a denotational or natural semantics and of an abstract machine, the CEK machine, for the λ -calculus with left-to-right applicative order. The present appendix illustrates the syntactic correspondence between the functional implementation of a reduction semantics and of an abstract machine, again the CEK machine, for the λ -calculus with left-to-right applicative order. Together, the functional correspondence and the syntactic correspondence therefore demonstrate the natural fit of the CEK machine in the semantic spectrum of the λ -calculus with explicit substitutions and left-to-right applicative order.

REFERENCES

- [1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1&2):3–57, 1992.
- [2] Samson Abramsky and R. Sykes. SECD-M: a virtual machine for applicative programming. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 81–98, Nancy, France, September 1985. Springer-Verlag.
- [3] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006.
- [4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.
- [5] Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. A symmetric approach to compilation and de-compilation. 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.
- [6] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.
- [7] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.
- [8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.
- [9] Anindya Banerjee. *The Semantics and Implementation of Bindings in Higher-Order Programming Languages*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, July 1995.
- [10] Fred Bayer. LispMe: An implementation of Scheme for the PalmPilot. In Manuel Serrano, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Scheme and Functional Programming*, Firenze, Italy, September 2001.
- [11] Gavin Bierman. Observations on a linear PCF. Technical Report 412, Computer Laboratory, University of Cambridge, Cambridge, UK, January 1997.
- [12] Malgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006.

- [13] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW’04).
- [14] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.
- [15] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.
- [16] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2005.
- [17] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.
- [18] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 16(3):269–280, 2006.
- [19] Graham Birtwistle and Brian T. Graham. Verifying SECD in HOL. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, pages 129–177. North-Holland, 1990.
- [20] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 226–237, La Jolla, California, June 1995. ACM Press.
- [21] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
- [22] Rod M. Burstall. Writing search algorithms in functional form. In Donald Michie, editor, *Machine Intelligence*, volume 5, pages 373–385. Edinburgh University Press, 1969.
- [23] Luca Cardelli. The functional abstract machine. *Polymorphism*, 1(1), January 1983.
- [24] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.
- [25] Jaeyoun Chung. An explicit polymorphic type system for verifying untrusted low-level codes. Master’s thesis, Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 1999.
- [26] Anthony Neil Clark. *Semantic Primitives for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Queen Mary and Westfield College, University of London, 1996.
- [27] John Clements. *Portable and High-Level Access to the Stack with Continuation Marks*. PhD thesis, College of Computer Science, Northeastern University, Boston, Massachusetts, February 2006.
- [28] John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in JavaScript. In Will Clinger, editor, *Proceedings of the 2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 1–9, Victoria, British Columbia, September 2008.
- [29] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.
- [30] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN’98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.
- [31] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.
- [32] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991.
- [33] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

- [34] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.
- [35] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in *Lecture Notes in Computer Science*, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33.
- [36] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006.
- [37] Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, Victoria, British Columbia, September 2008. ACM Press. Invited talk.
- [38] 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.
- [39] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
- [40] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in *Lecture Notes in Computer Science*, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.
- [41] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.
- [42] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [24], pages 327–341.
- [43] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 200?. In press. Extended version available as the research report BRICS RS-08-04.
- [44] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.
- [45] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), *Electronic Notes in Theoretical Computer Science*, Vol. 59.4.
- [46] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in *Lecture Notes in Computer Science*, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.
- [47] Antony J. T. Davie. *Introduction to Functional Programming Systems Using Haskell*, volume 27 of *Cambridge Computer Science Texts*. Cambridge University Press, 1992.
- [48] Antony J. T. Davie and David J. McNally. CASE - a lazy version of an SECD machine with a flat environment. In *Proceedings of the Fourth IEEE Region 10 International Conference (TENCON 1989)*, pages 864–872, Bombay, India, November 1989.
- [49] Arie de Bruin and Erik P. de Vink. Continuation semantics for Prolog with cut. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 351 in *Lecture Notes in Computer Science*, pages 178–192, Barcelona, Spain, March 1989. Springer-Verlag.

- [50] Matthias Felleisen. *The Calculi of λ -v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.
- [51] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.
- [52] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at <http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html> and last accessed in April 2008, 1989-2001.
- [53] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ -calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.
- [54] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [24], pages 52–62.
- [55] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, 1988.
- [56] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.
- [57] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, California, fifth edition, 2006.
- [58] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, New Orleans, Louisiana, January 1985. ACM Press.
- [59] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. The MIT Press, third edition, 2008.
- [60] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems · Computers · Controls*, 2(5):45–50, 1971. Reprinted in *Higher-Order and Symbolic Computation* 12(4):381–391, 1999, with an interview [61].
- [61] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
- [62] Michael Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Transactions on Programming Languages and Systems*, 6(4):603–631, 1984.
- [63] Hugh Glaser, Chris Hankin, and David Till. *Principles of Functional Programming*. Prentice-Hall International, 1984.
- [64] Carsten K. Gomard and Peter Sestoft. Globalization and live variables. In Hudak and Jones [72], pages 166–177.
- [65] Brian T. Graham. *The SECD microprocessor: a verification case study*. Kluwer Academic Publishers, 1992.
- [66] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.
- [67] John Hannan. Staging transformations for abstract machines. In Hudak and Jones [72], pages 130–141.
- [68] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
- [69] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.
- [70] Peter Henderson. *Functional Programming – Application and Implementation*. Prentice-Hall International, 1980.
- [71] Martin C. Henson. *Elements of Functional Languages*. Computer Science Texts. Blackwell Scientific Publications, 1987.
- [72] Paul Hudak and Neil D. Jones, editors. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM Press.
- [73] Jacob Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, June 2008. BRICS research report RS-08-6.

- [74] Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In Shimon Even and Oded Kariv, editors, *Automata, Languages, and Programming, 8th Colloquium*, number 115 in Lecture Notes in Computer Science, pages 114–128, Acre (Akko), Israel, July 1981. Springer-Verlag.
- [75] Yukiyoishi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.
- [76] Yukiyoishi Kameyama. Axioms for control operators in the CPS hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW’04).
- [77] Yukiyoishi 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 (ICFP’03)*, SIGPLAN Notices, Vol. 38, No. 9, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.
- [78] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, March 2005.
- [79] Werner E. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.
- [80] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- [81] Peter J. Landin. A correspondence between Algol 60 and Church’s lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.
- [82] Peter J. Landin. A generalization of jumps and labels. Research report, UNIVAC Systems Programming Research, 1965. Reprinted in *Higher-Order and Symbolic Computation* 11(2):125–143, 1998, with a foreword [115].
- [83] Peter J. Landin. A λ -calculus approach. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation*, Symposium Publication Division, chapter 5, pages 97–141. Pergamon Press, 1966.
- [84] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
- [85] Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW’97)*, Technical report BRICS NS-96-13, Aarhus University, pages 1:1–9, Paris, France, January 1997.
- [86] Peter J. Landin. My years with Strachey. *Higher-Order and Symbolic Computation*, 13(1/2):75–76, 2000.
- [87] Clement L. McGowan. The correctness of a modified SECD machine. In *Proceedings of the Second Annual ACM Symposium in the Theory of Computing*, pages 149–157, Northampton, Massachusetts, May 1970.
- [88] Erik Meijer. Generalised expression evaluation. Technical Report 88-5, Department of Informatics, University of Nijmegen, Nijmegen, The Netherlands, 1988.
- [89] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, June 2007.
- [90] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, May 2007.
- [91] F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.
- [92] Peter D. Mosses. A foreword to ‘Fundamental concepts in programming languages’. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.
- [93] Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master’s thesis, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2007. BRICS research report RS-08-3.
- [94] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the First ACM SIGPLAN*

- Workshop on Continuations (CW'92)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.
- [95] Peter Møller Neergaard. *Complexity Aspects of Programming Language Design—From Logspace to Elementary Time via Proofnets and Intersection Types*. PhD thesis, MIT School of Computer Science, Brandeis University, Waltham, Massachusetts, October 2004.
- [96] Flemming Nielson and Hanne Riis Nielson. Comments on Georgeff's 'transformations and reduction strategies for typed lambda expressions'. *ACM Transactions on Programming Languages and Systems*, 8(3):406–407, 1984.
- [97] Michel Parigot. $\lambda\mu$ -calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, number 624 in Lecture Notes in Artificial Intelligence, pages 190–201, St. Petersburg, Russia, July 1992. Springer-Verlag.
- [98] Larry Paulson. *A Compiler Generator for Semantic Grammars*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, December 1981. Report No. STAN-CS-81-893.
- [99] Uwe Pleban. Compiler prototyping using formal semantics. In Susan L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 94–105, Montréal, Canada, June 1984. ACM Press.
- [100] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [101] John D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.
- [102] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in *Higher-Order and Symbolic Computation* 11(4):363–397, 1998, with a foreword [104].
- [103] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.
- [104] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
- [105] Colin Runciman and Ian Toyn. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4(4):339–363, 1986.
- [106] Peter Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991. DIKU Rapport 92/6.
- [107] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.
- [108] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007. A preliminary version was presented at the 2004 Workshop on Scheme and Functional Programming [107].
- [109] Mike Spivey. The SECD machine – a tutorial reconstruction. Unpublished lecture notes, Oxford University, Easter 2003.
- [110] 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.
- [111] 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.
- [112] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
- [113] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in *Higher-Order and Symbolic Computation* 13(1/2):11–49, 2000, with a foreword [92].

- [114] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in *Higher-Order and Symbolic Computation* 13(1/2):135–152, 2000, with a foreword [118].
- [115] Hayo Thielecke. An introduction to Landin’s “A generalization of jumps and labels”. *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.
- [116] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, 2002.
- [117] Vasco Thudichum Vasconcelos. Lambda and pi calculi, CAM and SECD machines. *Journal of Functional Programming*, 15(1):101–127, 2005.
- [118] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.
- [119] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.
- [120] Hongwei Xi. Evaluation under lambda abstraction. In Hugh Glaser, H. Hartel, and Herbert Kuchen, editors, *Ninth International Symposium on Programming Language Implementation and Logic Programming*, number 1292 in Lecture Notes in Computer Science, pages 259–273, Southampton, UK, September 1997. Springer-Verlag.