A Rational Deconstruction of Landin's J Operator

Olivier Danvy
Kevin Millikin

BRICS Report Series RS-06-4

ISSN 0909-0878 February 2006

A Rational Deconstruction of Landin's J Operator[*]

Olivier Danvy and Kevin Millikin
BRICS[†]
Department of Computer Science
University of Aarhus[‡]

February 28, 2006

Abstract

Landin's J operator was the first control operator for functional languages, and was specified with an extension of the SECD machine. Through a series of meaning-preserving transformations (transformation into continuation-passing style (CPS) and defunctionalization) and their left inverses (transformation into direct style and refunctionalization), we present a compositional evaluation function corresponding to this extension of the SECD machine. We then characterize the J operator in terms of CPS and in terms of delimited-control operators in the CPS hierarchy. Finally, we present a motivated wish to see Landin's name added to the list of co-discoverers of continuations.

[*]Extended version of an article to appear in the proceedings of the 17th International Workshop on the Implementation and Application of Functional Languages (IFL'05), Dublin, Ireland, September 2005.

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

[‡]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `<danvy@brics.dk>`, `<kmillikin@brics.dk>`

Contents

1 Introduction

Forty years ago, Peter Landin unveiled the first control operator, J, to a heretofore unsuspecting world [26–28]. He did so to generalize the notion of jumps and labels and showed that the resulting 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 [25].

At IFL'04, Danvy presented a 'rational deconstruction' of Landin's SECD machine into a compositional evaluation function [9]. The goal of this work is to extend this rational deconstruction to the J operator.

1.1 Deconstruction of the SECD machine with the J operator

Let us outline our deconstruction of the SECD machine before substantiating it in Section 2. We essentially follow the order of Danvy's deconstruction [9], though with a twist: in the middle of the derivation, we abandon the stack-threading, callee-save features of the SECD machine for the more familiar register-based, caller-save features of traditional definitional interpreters [19,31, 34].

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

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

The definition of this transition function is complicated because it has several induction variables, i.e., it dispatches on several components of the quadruple.

- We disentangle it into four transition functions, each of which has one induction variable, i.e., dispatches on one component of the quadruple:

  ```
  run_c :        S * E * C * D -> value
  run_d :            value * D -> value
  run_t : term * S * E * C * D -> value
  run_a :        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 is in defunctionalized form: 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 :        S * E * C * D -> value
  where C = S * E * D -> value and D = value -> value
  ```

As identified in the first rational deconstruction [9], the resulting program is a stack-threading, callee-save interpreter in continuation-passing style (CPS).

- In order to focus on the nature of the J operator, we eliminate the data stack and adopt the more familiar caller-save evaluation strategy:

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

  The interpreter is still in CPS.

- The direct-style transformation eliminates the dump continuation:

  ```
  run_t :      term * E * C -> value
  run_a : 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 [10]. The resulting evaluator still threads an explicit continuation, even though it is not tail-recursive.

- The direct-style transformation eliminates the control continuation:

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

  The clauses catering for the non-tail-recursive uses of the control continuation are expressed using the delimited-control operators shift_1, reset_1, shift_2, and reset_2 [4, 10, 14, 23, 33]. The resulting evaluator is in direct style. It is also in closure-converted form: the applicable values are a defunctionalized data type and `run_a` is the corresponding apply function.

- Refunctionalization eliminates the apply function:

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

  The resulting evaluator is compositional.

There is plenty of room for variation in the present reconstruction. The path we are taking seems reasonably pedagogical—in particular, the departure from threading a data stack and managing the environment in a callee-save fashion. Each of the steps is reversible: one can CPS-transform and defunctionalize an evaluator into an abstract machine [1–4, 9].

1.2 Prerequisites and domain of discourse

Up to Section 2.4, we use pure ML as a meta-language. We assume a basic familiarity with Standard ML and with reasoning about ML programs as well as an elementary understanding of defunctionalization [13, 34], the CPS transformation [10, 11, 19, 31, 34, 37], and delimited continuations [4, 10, 14, 17, 23]. From Section 2.5, we use pure ML with delimited-control operators.

The source language of the SECD machine. The source language is the λ-calculus, extended with literals (as observables) and the J operator. 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. 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`.

Values. There are five kinds of values: integers, the successor function, function closures, program closures, and "state appenders" [6, page 84]:

```
datatype value = INT of int
               | SUCC
               | FUNCLO of E * string * term
               | PGMCLO of value * D
               | STATE_APPENDER of D
withtype S = value list                                   (* 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 program closure is a first-class continuation. A state appender is an intermediate value; applying it yields a program closure.

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

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

1.3 Overview

We first detail the deconstruction of the SECD machine into a compositional evaluator in direct style (Section 2). We then analyze the J operator (Sections 3 and 4), review related work (Sections 6 and 5), and conclude (Sections 8 and 9).

2 Deconstruction of the SECD machine with the J operator

2.1 The starting specification

Several formulations of the SECD machine with the J operator have been published [6,16,27]. We take the most recent one, i.e., Felleisen's [16], as our starting point, and we consider the others in Section 6.

```
(*  run : S * E * C * D -> value                                    *)
(*  where S = value list, E = value Env.env, C = directive list,    *)
(*    and D = (S * E * C) list                                      *)
fun run (v :: nil, e, nil, nil)
    = v
  | run (v :: nil, 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)                                (* 1 *)
    = 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 ((PGMCLO (v, d')) :: v' :: s, e, APPLY :: c, d)         (* 2 *)
    = run (v :: v' :: nil, e_init, APPLY :: nil, d')
  | run ((STATE_APPENDER d') :: v :: s, e, APPLY :: c, d)       (* 3 *)
    = run ((PGMCLO (v, d')) :: s, e, c, d)

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

The SECD machine does not terminate for divergent source terms. If it becomes stuck, an ML pattern-matching error is raised (alternatively, the codomain of `run` could be made `value option` and a fallthrough `else` clause could be added). Otherwise, the result of the evaluation is `v` for some ML value `v : value`. 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 "3"), yields a program closure. Applying a program closure (in the clause marked "2") restores the captured dump.

2.2 A disentangled specification

In the definition of Section 2.1, all the possible transitions are meshed together in one recursive function, `run`. As in the first rational deconstruction [9], we factor `run` into four mutually recursive functions, each of them with one induction variable. In this disentangled definition,

- `run_c` interprets the list of control directives, i.e., it specifies which transition to take 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.
- `run_a` interprets the top value in the current data stack.

```
(*  run_c :        S * E * C * D -> value                          *)
(*  run_d :            value * D -> value                          *)
(*  run_t : term * S * E * C * D -> value                          *)
(*  run_a :        S * E * C * D -> value                          *)
(*  where S = value list, E = value Env.env, C = directive list,   *)
(*    and D = (S * E * C) list                                     *)
fun run_c (v :: nil, e, nil, d)
    = run_d (v, d)
  | run_c (s, e, (TERM t) :: c, d)
    = run_t (t, s, e, c, d)
  | run_c (s, e, APPLY :: c, d)
    = run_a (s, e, c, d)
and run_d (v, nil)
    = 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_t (t1, s, e, (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_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')
  | run_a ((STATE_APPENDER d') :: v :: s, e, c, d)
    = run_c ((PGMCLO (v, d')) :: s, e, c, d)

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

Proposition 1 (full correctness) *Given a program,* `evaluate0` *and* `evaluate1` *either both diverge or both yield values that are structurally equal.*

2.3 A higher-order counterpart

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

These data types, together with `run_d` and `run_c`, are in the image of defunctionalization (`run_d` and `run_c` are the apply functions of these two data types). The corresponding higher-order evaluator reads as follows; it is higher-order because `c` and `d` now denote functions:

```
(*  run_t : term * S * E * C * D -> value                             *)
(*  run_a :        S * E * C * D -> value                             *)
(*  where S = value list, E = value Env.env, C = S * E * D -> value    *)
(*    and D = value -> 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 (s, e, d) => run_a (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 :: nil, e'', d) => d v,
             fn v => c (v :: s, e, d))
  | run_a ((PGMCLO (v, d')) :: v' :: s, e, c, d)
    = run_a (v :: v' :: nil, e_init, fn (v :: nil, e, d) => d v, d')
  | run_a ((STATE_APPENDER d') :: v :: s, e, c, d)
    = c ((PGMCLO (v, d')) :: s, e, d)
```

```
fun evaluate2 t                     (*  evaluate2 : program -> value  *)
    = run_t (t, nil, e_init, fn (v :: nil, 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 2.2.

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

2.4 A stack-less, caller-save counterpart

We want to focus on J, and the non-standard aspects of the evaluator of Section 2.3 (the data stack and the callee-save environment) are a distraction. We therefore transmogrify the evaluator into the more familiar register-based, caller-save form [19,31,34], renaming `run_t` as `eval` and `run_a` as `apply`. Intermediate values are explicitly passed instead of being stored on the data stack, and environments are no longer passed to `apply` and to the control continuation:

```
(*  eval  :      term * E * C * D -> value                          *)
(*  apply : value * value * C * D -> value                          *)
(*  where E = value Env.env, C = value * D -> value,                *)
(*    and D = value -> 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 (PGMCLO (v, d'), v', c, d)
    = apply (v, v', fn (v, d) => d v, d')
  | apply (STATE_APPENDER d', v, c, d)
    = c (PGMCLO (v, d'), d)

fun evaluate3 t                     (*  evaluate3 : 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.

Proposition 3 (full correctness) *Given a program,* `evaluate2` *and* `evaluate3` *either both diverge or both yield values that are structurally equal.*

2.5 A dump-less direct-style counterpart

The evaluator of Section 2.4 is in continuation-passing style, and therefore it is in the image of the CPS transformation. The clause for J captures the current continuation (i.e., the dump), and therefore its direct-style counterpart naturally uses call/cc [11]. With an eye on our next step, we do not, however, use call/cc but its cousins shift and reset [10, 14] 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 [14, 17]: reset delimits the (now implicit) dump continuation in `evaluate`, 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:

```
structure SR = Shift_and_Reset (type intermediate_answer = value)

(*  eval  :      term * E * C -> value                            *)
(*  apply : value * value * C -> value                            *)
(*  where E = value Env.env and C = value -> 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 ((PGMCLO (v, d)), v', c)
    = SR.shift (fn d' => d (apply (v, v', fn v => v)))         (* * *)
  | apply (STATE_APPENDER d, v, c)
    = c (PGMCLO (v, d))

fun evaluate4 t                       (*  evaluate4 : program -> value  *)
    = SR.reset (fn () => eval (t, e_init, fn v => v))
```

The dump continuation is now implicit and is accessed using shift. CPS-transforming this evaluator yields the evaluator of Section 2.4.

Proposition 4 (full correctness) *Given a program,* `evaluate3` *and* `evaluate4` *either both diverge or both yield values that are structurally equal.*

2.6 A control-less direct-style counterpart

The evaluator of Section 2.5 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 "`*`") and for the occurrences of shift and reset. This pattern of control is characteristic of the CPS hierarchy [4, 10, 14, 23]. We therefore use the delimited-control operators shift_1, reset_1, shift_2, and reset_2 to write the direct-style counterpart of this evaluator (shift_2 and reset_2 are the direct-style counterparts of shift_1 and reset_1, and shift_1 and reset_1 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 [14, 17]: reset_2 delimits the (now twice implicit) dump continuation in `evaluate`; shift_2 captures it in the clauses where J is evaluated and where a program closure is applied; reset_1 delimits the (now implicit) control continuation in `evaluate` and in `apply`, and corresponds to its initialization with the identity function; and shift_1 captures it in the clause where J is evaluated:

```
structure SR1 = Shift_and_Reset (type intermediate_answer = value)

structure SR2 = Shift_and_Reset_next (type intermediate_answer = value
                                      structure over = SR1)

(*  eval  :      term * E -> value                                  *)
(*  apply : value * value -> 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)
    = FUNCLO (e, x, t)
  | eval (APP (t0, t1), e)
    = let val v1 = eval (t1, e)
          val v0 = eval (t0, e)
      in apply (v0, v1) end
  | eval (J, e)
    = SR1.shift (fn c => SR2.shift (fn d => d (c (STATE_APPENDER d))))
and apply (SUCC, INT n)
    = INT (n+1)
  | apply (FUNCLO (e', x, t), v)
    = SR1.reset (fn () => eval (t, Env.extend (x, v, e')))
  | apply (PGMCLO (v, d), v')
    = SR1.shift (fn c' => SR2.shift (fn d' =>
      d (SR1.reset (fn () => apply (v, v')))))
  | apply (STATE_APPENDER d, v)
    = PGMCLO (v, d)

fun evaluate5 t                       (*  evaluate5 : program -> value  *)
    = SR2.reset (fn () => SR1.reset (fn () => eval (t, e_init)))
```

The control continuation is now implicit and is accessed using shift$_1$. The dump continuation is still implicit and is accessed using shift$_2$. CPS-transforming this evaluator yields the evaluator of Section 2.5.

Proposition 5 (full correctness) *Given a program,* `evaluate4` *and* `evaluate5` *either both diverge or both yield values that are structurally equal.*

2.7 A compositional counterpart

We now turn to the data flow of the evaluator of Section 2.6. As for the SECD machine without J [9], this evaluator is in defunctionalized form: each of the values constructed with `SUCC`, `FUNCLO`, `PGMCLO`, and `STATE_APPENDER` are constructed at one place and consumed at another (the `apply` function). We therefore refunctionalize them into the function space `value -> value`:

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

val e_init = Env.extend ("succ",
                         FUN (fn (INT n) => INT (n+1)),
                         Env.empty)

structure SR1 = Shift_and_Reset (type intermediate_answer = value)

structure SR2 = Shift_and_Reset_next (type intermediate_answer = value
                                      structure over = SR1)

(*  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 (FUN f) => FUN (fn v' => SR1.shift (fn c' =>
                                             SR2.shift (fn d' =>
                                             d (SR1.reset (fn () =>
                                                f v'))))))))))

fun evaluate5' t                   (*  evaluate5' : program -> value  *)
    = SR2.reset (fn () => SR1.reset (fn () => eval (t, e_init)))
```

Defunctionalizing this evaluator yields the evaluator of Section 2.6.

Proposition 6 (full correctness) *Given a program,* `evaluate5` *and* `evaluate5'` *either both diverge or both yield values that are related by defunctionalization.*

2.8 Summary

We graphically summarize the derivations as follows. The evaluators in the top row are the defunctionalized counterparts of the evaluators in the bottom row.

$$\begin{tikzcd}
\texttt{evaluate3} \arrow[r, shift right] \arrow[d, shift right, "\text{refunct.}"'] & \texttt{evaluate4} \arrow[l, shift right, "\text{CPS transf.}"'] \arrow[r, shift right] \arrow[d, shift right] & \texttt{evaluate5} \arrow[l, shift right, "\text{CPS transf.}"'] \arrow[d, shift right] \\
\texttt{evaluate3'} \arrow[u, shift right] \arrow[r, shift right, "\text{DS transf.}"'] & \texttt{evaluate4'} \arrow[u, shift right] \arrow[l, shift right] \arrow[r, shift right, "\text{DS transf.}"'] & \texttt{evaluate5'} \arrow[u, shift right, "\text{defunct.}"'] \arrow[l, shift right]
\end{tikzcd}$$

3 Three simulations of the J operator

The evaluator of Section 2.7 and the refunctionalized counterparts of the evaluators of Sections 2.5 and 2.4 are compositional. They can be viewed as syntax-directed encodings into their meta-language, as embodied in the first Futamura projection [20]. 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$_2$ ($\langle\!\langle\cdot\rangle\!\rangle_2$), shift$_1$ ($\mathcal{S}_1$), and reset$_1$ ($\langle\!\langle\cdot\rangle\!\rangle_1$):

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

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

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

$$\begin{aligned}
\llbracket n \rrbracket' &= \lambda c.c\ n\\
\llbracket x \rrbracket' &= \lambda c.c\ x\\
\llbracket t_0\ t_1 \rrbracket' &= \lambda c.\llbracket t_1 \rrbracket'\ \lambda x_1.\llbracket t_0 \rrbracket'\ \lambda x_0.x_0\ x_1\ c\\
\llbracket \lambda x.t \rrbracket' &= \lambda c.c\ \lambda x.\lambda c.c\ (\llbracket t \rrbracket'\ \lambda x.x)\\
\llbracket \mathbf{J} \rrbracket' &= \lambda c.\mathcal{S} \lambda d.d\ (c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.\mathcal{S} \lambda d'.d\ (x\ x'\ \lambda x''.x'')})
\end{aligned}$$

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

- In CPS with two layers of continuations (the outer continuation, i.e., the dump continuation, can be η-reduced in the first three clauses):

$$\begin{aligned}
\llbracket n \rrbracket'' &= \lambda c.\lambda d.c\ n\ d\\
\llbracket x \rrbracket'' &= \lambda c.\lambda d.c\ x\ d\\
\llbracket t_0\ t_1 \rrbracket'' &= \lambda c.\lambda d.\llbracket t_1 \rrbracket''\ (\lambda x_1.\lambda d.\llbracket t_0 \rrbracket''\ (\lambda x_0.\lambda d.x_0\ x_1\ c\ d)\ d)\ d\\
\llbracket \lambda x.t \rrbracket'' &= \lambda c.\lambda d.c\ (\lambda x.\lambda c.\lambda d.\llbracket t \rrbracket''\ (\lambda x.\lambda d.d\ x)\ \lambda x.c\ x\ d)\ d\\
\llbracket \mathbf{J} \rrbracket'' &= \lambda c.\lambda d.c\ (\lambda x.\lambda c.\lambda d'''.c\ \boxed{(\lambda x'.\lambda c'.\lambda d'.x\ x'\ (\lambda x''.\lambda d''.d''\ x'')\ d)}\ d''')\ d
\end{aligned}$$

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

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.

The first rational deconstruction [9] 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.

4 Four more simulations of the J operator

4.1 The $\mathcal{C}$ operator and the CPS hierarchy

In the terminology of reflective towers [12], 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, the term $\mathcal{S}\lambda d'.d\ (x\ x'\ \lambda x''.x'')$ 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.

On the other hand, still using the terminology of reflective towers, continuations captured with call/cc [8] or with Felleisen's $\mathcal{C}$ operator [15] are "jumpy"—at their point of invocation, they discard the current continuation. If the continuation d were captured with $\mathcal{C}$, then the term $d\ (x\ x'\ \lambda x''.x'')$ would suffice to discard the current continuation.

The first encoding of J in Section 3 uses the pushy control operators $\mathcal{S}_1$ (i.e., $\mathcal{S}$) and $\mathcal{S}_2$. Murthy [33] and Kameyama [23] 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. As before, we assume a call-by-value meta-language with right-to-left evaluation.

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

$$[\![\mathbf{J}]\!] \quad = \quad \mathcal{C}_1\lambda c.\mathcal{C}_2\lambda d.d\ (c\ \lambda x.\boxed{\lambda x'.d\ \langle x\ x'\rangle_1})$$

  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 reset ($\langle\!\langle\cdot\rangle\!\rangle$):

$$[\![\mathbf{J}]\!]' = \lambda c.\mathcal{C}\lambda d.d\ (c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.d\ (x\ x'\ \lambda x''.x'')})$$

The corresponding CPS simulation of J with two layers of continuations coincides with the one in Section 3.

4.2 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, $1 + (\textbf{call/cc}\ \lambda k.10)$ 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:

$$[\![\mathbf{J}]\!]' = \lambda c.\textbf{call/cc}\ \lambda d.c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.d\ (x\ x'\ \lambda x''.x'')}$$

The obvious generalization of call/cc to the CPS hierarchy does not fit the bill as well, however. One needs an abort operator as well in order for call/cc$_2$ to capture the initial continuation and the current meta-continuation. We leave the rest of this train of thoughts to the imagination of the reader.

5 On the design of control operators

We note that replacing $\mathcal{C}$ with $\mathcal{S}$ above (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 acid test for a new control operator lies elsewhere, for example:

1. Which programming idiom does this control operator reflect?

2. What is the logical content of this control operator [22]?

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

1. J is a generalized return [39, Section 2.1], and

2. the type of $\mathbf{J}\ \lambda x.x$ is the law of the excluded middle [40, Section 5.2].

On the other hand, despite its remarkable fit to Algol labels and jumps [16,28], J is unintuitive to use. For example, if a let expression is the syntactic sugar of a beta-redex (and x_1 is fresh), the equivalence

$$t_0\, t_1 \;\cong\; \textbf{let}\, x_1 = t_1 \,\textbf{in}\, t_0\; x_1$$

does *not* hold in the presence of J, even though it does in the presence of call/cc, $\mathcal{C}$, and shift for right-to-left evaluation.

6 Related work

6.1 Landin and Burge

Landin [27] introduced the J operator as a new language feature motivated by three questions about labels and jumps:

- Can a language have jumps without having assignment?
- 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?

He gave the semantics of the J operator by extending the SECD machine. In addition to using J to model jumps in Algol 60 [26], 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 [6, 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 [39] gave a detailed introduction to the J operator as presented by Landin and Burge. Burstall [7] 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.

6.2 Reynolds

Reynolds [34] gave a comparison of J to escape, the binder form of Scheme's call/cc [8].[1] 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:

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

As Thielecke notes [39], this encoding is only valid immediately inside an abstraction. Indeed, the dump continuation captured by J only coincides with

[1] $\textbf{escape}\, k \,\textbf{in}\, t \;\equiv\; \textbf{call/cc}\; \lambda k.t$

the continuation captured by escape if the control continuation is the initial one (i.e., immediately inside a control delimiter). Thielecke generalized the encoding by adding a dummy abstraction:

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

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

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

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

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.

6.3 Felleisen

Felleisen showed how to embed Landin's extension of applicative expressions with J into the Scheme programming language [16]. The embedding is defined as Scheme syntactic extensions (i.e., macros). J is treated as a dynamic identifier that is bound in the body of every abstraction. Its control aspect is handled through Scheme's control operator call/cc.

As pointed out by Thielecke [39], Felleisen's simulation can be stated in direct style, assuming a call-by-value meta-language with right-to-left evaluation and call/cc. In addition, we present the corresponding simulations using $\mathcal{C}$ and reset, using shift and reset, and in CPS:

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

$$\begin{aligned}
[\![x]\!] &= x\\
[\![t_0\; t_1]\!] &= [\![t_0]\!]\; [\![t_1]\!]\\
[\![\lambda x.t]\!] &= \lambda x.\textbf{call/cc}\; \lambda d.\textbf{let}\; \textbf{J} = \lambda x.\boxed{\lambda x'.d\; (x\; x')}\; \textbf{in}\; [\![t]\!]\\
&= \lambda x.\mathcal{C}\lambda d.\textbf{let}\; \textbf{J} = \lambda x.\boxed{\lambda x'.d\; (x\; x')}\; \textbf{in}\; d\; [\![t]\!]\\
&= \lambda x.\mathcal{S}\lambda d.\textbf{let}\; \textbf{J} = \lambda x.\boxed{\lambda x'.\mathcal{S}\lambda c'.d\; (x\; x')}\; \textbf{in}\; d\; [\![t]\!]
\end{aligned}$$

  A program p is translated as $\langle\!\langle [\![p]\!] \rangle\!\rangle$.

- In CPS:

$$\begin{aligned}
[\![x]\!]' &= \lambda c.c\; x\\
[\![t_0\; t_1]\!]' &= \lambda c.[\![t_1]\!]'\; \lambda x_1.[\![t_0]\!]'\; \lambda x_0.x_0\; x_1\; c\\
[\![\lambda x.t]\!]' &= \lambda c.c\; (\lambda x.\lambda d.\textbf{let}\; \textbf{J} = \lambda x.\lambda c.c\; \boxed{\lambda x'.\lambda c'.x\; x'\; d}\; \textbf{in}\; [\![t]\!]'\; d)
\end{aligned}$$

  A program p is translated as $[\![p]\!]'\; \lambda x.x$.

Analysis: The simulation of variables and applications is standard. The continuation of the body of each λ-abstraction is captured, and the identifier 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.

6.4 Felleisen and Burge

Felleisen's version of the SECD machine with the J operator differs from Burge's. In the notation of Section 2.1, 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 [5].

Burge's version, however, is not in defunctionalized form. In Section 7, we put it in defunctionalized form without inserting a control delimiter and we outline the corresponding compositional evaluation functions and simulations.

7 An alternative deconstruction

7.1 Our starting point: Burge's specification

As pointed out in Section 6.4, 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 2.1:

```
(*  run : S * E * C * D -> value                                    *)
(*  where S = value list, E = value Env.env, C = directive list,    *)
(*    and D = (S * E * C) list                                      *)

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

fun evaluate0_alt t              (*  evaluate0_alt : program -> value  *)
    = ...
```

Just as in Section 2.2, 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 inserting a control delimiter, and then repeat the rest of the rational deconstruction.

7.2 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 Reynolds's 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 on 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 2.2:

```
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 :          S * E * C * D -> value                             *)
(*  where S = value list, E = value Env.env, C = directive list,        *)
(*    and D = (S * E * C) list                                          *)
fun run_c (v :: nil, e, nil, d)
    = run_d (YIELD v, d)                                        (* 1 *)
  | run_c (s, e, (TERM t) :: c, d)
    = run_t (t, s, e, c, d)
  | run_c (s, e, APPLY :: c, d)
    = run_a (s, e, c, d)

and run_d (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 7 (full correctness) *Given a program,* `evaluate0_alt` *and* `evaluate1_alt` *either both diverge or both yield values that are structurally equal.*

7.3 A higher-order counterpart

In the modified specification of Section 7.2, the data types of control stacks and dumps are identical to those of the disentangled machine of Section 2.2. 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:

```
(*  run_t : term * S * E * C * D -> value                               *)
(*  run_a :          S * E * C * D -> value                             *)
(*  where S = value list, E = value Env.env, C = S * E * D -> value     *)
(*    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 ((PGMCLO (v, d')) :: v' :: s, e, c, d)
    = d' (THROW (v, v'))
  | run_a ((STATE_APPENDER d') :: v :: s, e, c, d)
    = c ((PGMCLO (v, d')) :: s, e, d)

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 7.2.

Proposition 8 (full correctness) *Given a program,* `evaluate1_alt` *and* `evaluate2_alt` *either both diverge or yield expressible values that are structurally equal.*

7.4 The rest of the rational deconstruction

The evaluator of Section 7.3 can be transformed exactly as the higher-order evaluator of Section 2.3:

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. 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.

7.5 Three alternative simulations of the J operator

As in Section 3, the compositional counterpart of the evaluators of Section 7.4 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 of $\mathcal{C}_2$, reset$_2$, $\mathcal{C}_1$, and reset$_1$:

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

  A program p is translated as $\langle\mathbf{let}\ \mathbf{inL}(x) = \langle\mathbf{inL}([\![p]\!])\rangle_1\ \mathbf{in}\ x\rangle_2$.

- In CPS with one layer of continuations, using either of shift and reset, of $\mathcal{C}$ and reset, or of call/cc and reset:

$$
\begin{aligned}
[\![n]\!]' &= \lambda c.c\ n \\
[\![x]\!]' &= \lambda c.c\ x \\
[\![t_0\ t_1]\!]' &= \lambda c.[\![t_1]\!]'\ (\lambda x_1.[\![t_0]\!]'\ \lambda x_0.x_0\ x_1\ c) \\
[\![\lambda x.t]\!]' &= \lambda c.c\ (\lambda x.\lambda c.\mathbf{case}\ \ [\![t]\!]'\ \lambda x.\mathbf{inL}(x) \\
&\qquad\qquad\qquad\quad\ \mathbf{of}\ \ \mathbf{inL}(x) \Rightarrow c\ x \\
&\qquad\qquad\qquad\qquad\mid\ \ \mathbf{inR}(x, x') \Rightarrow x\ x'\ c) \\
[\![\mathbf{J}]\!]' &= \lambda c.\mathcal{S}\lambda d.d\ (c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.\mathcal{S}\lambda d'.d\ (\mathbf{inR}(x, x'))}) \\
&= \lambda c.\mathcal{C}\lambda d.d\ (c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.d\ (\mathbf{inR}(x, x'))}) \\
&= \lambda c.\mathbf{call/cc}\ \lambda d.c\ \lambda x.\lambda c.c\ \boxed{\lambda x'.\lambda c'.d\ (\mathbf{inR}(x, x'))}
\end{aligned}
$$

  A program p is translated as $\langle\mathbf{let}\ \mathbf{inL}(x) = [\![p]\!]'\ \lambda x.\mathbf{inL}(x)\ \mathbf{in}\ x\rangle$.

- In CPS with two layers of continuations:

$$
\begin{aligned}
[\![n]\!]'' &= \lambda c.\lambda d.c\ n\ d \\
[\![x]\!]'' &= \lambda c.\lambda d.c\ x\ d \\
[\![t_0\ t_1]\!]'' &= \lambda c.\lambda d.[\![t_1]\!]''\ (\lambda x_1.\lambda d.[\![t_0]\!]''\ (\lambda x_0.\lambda d.x_0\ x_1\ c\ d)\ d)\ d \\
[\![\lambda x.t]\!]'' &= \lambda c.\lambda d.c\ (\lambda x.\lambda c.\lambda d.[\![t]\!]''\ (\lambda x.\lambda d.d\ (\mathbf{inL}(x))) \\
&\qquad\qquad\qquad\qquad\quad \lambda x''.\mathbf{case}\ \ x'' \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{of}\ \ \mathbf{inL}(x) \Rightarrow c\ x\ d \\
&\qquad\qquad\qquad\qquad\qquad\qquad\mid\ \ \mathbf{inR}(x, x') \Rightarrow x\ x'\ c\ d)\ d \\
[\![\mathbf{J}]\!]'' &= \lambda c.\lambda d.c\ (\lambda x.\lambda c.\lambda d'''.c\ \boxed{(\lambda x'.\lambda c'.\lambda d'.d\ (\mathbf{inR}(x, x')))}\ d''')\ d
\end{aligned}
$$

A program p is translated as $[\![p]\!]'' \; (\lambda x.\lambda d.d \; (\mathbf{inL}(x)))$
$\lambda x.\mathbf{let} \; \mathbf{inL}(x') = x \; \mathbf{in} \; x'$.

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 type 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 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.

7.6 Related work

Kiselyov's encoding of dynamic delimited continuations in terms of the static delimited-continuation operators shift and reset [24] is similar to this alternative encoding of the J operator. 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. However, in order to encode dynamic delimited continuations, Kiselyov uses a recursive meta-continuation.

8 Summary and conclusion

We have extended the rational deconstruction of the SECD machine to the J operator, and we have presented a series of alternative implementations, including a compositional evaluation function in CPS. In passing, we have also presented new applications of defunctionalization and new examples of control delimiters and of both pushy and jumpy delimited continuations in programming practice.

9 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 [27, page 133]

It was Strachey who coined the term "first-class functions" [38, Section 3.5.1].[2] In turn it was Landin who, through the J operator, invented what we know today as first-class continuations [18]. Indeed, like Reynolds for escape, 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.[3]

Today, 'continuations' 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 at the turn of the 1970's [35]; 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, 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' 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 so he is not listed among the co-discoverers of continuations [35]. 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 this article, we have shown that, though it is first-order, the SECD machine directly corresponds to a compositional evaluation function in CPS—the tool of choice for specifying control operators since Reynolds's work [34]. As a corollary, the dump directly corresponds to a functional representation of control, since it is a defunctionalized continuation. Therefore, in the light of defunctionalization, we wish to see Landin's name added to the list of co-discoverers of continuations.

[2] *"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 [30, page 75]

[3] *"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 [29, Section 1]

Acknowledgments: Thanks are due to Dariusz Biernacki, Julia L. Lawall, Kristian Støvring, and the anonymous reviewers of IFL'05 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 9.

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 and by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

References

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

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS RS-04-3.

[3] 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 technical report BRICS RS-04-28.

[4] 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).

[5] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. Research Report BRICS RS-05-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2005. Theoretical Pearl to appear in the Journal of Functional Programming.

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

[7] Rod M. Burstall. Writing search algorithms in functional form. In Donald Michie, editor, *Machine Intelligence*, volume 5, pages 373–385. Edinburgh University Press, 1969.

[8] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, edi-

tors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[9] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the technical report BRICS RS-03-33.

[10] 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.

[11] 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.

[12] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 327–341, Snowbird, Utah, July 1988. ACM Press.

[13] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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 [21].

[21] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[22] 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.

[23] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[24] 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.

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

[26] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[27] 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 [39].

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

[29] 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, University of Aarhus, pages 1:1–9, Paris, France, January 1997.

[30] Peter J. Landin. My years with Strachey. *Higher-Order and Symbolic Computation*, 13(1/2):75–76, 2000.

[31] 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.

[32] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

[33] 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.

[34] 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 [36].

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

[36] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[37] 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.

[38] 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 [32].

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

[40] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, 2002.