Basic Research in Computer Science

A Rational Deconstruction of Landin's J Operator

Olivier Danvy
Kevin Millikin

BRICS Report Series **RS-06-17**

ISSN 0909-0878 **December 2006**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/06/17/`

A Rational Deconstruction of Landin's J Operator*

Olivier Danvy and Kevin Millikin
Department of Computer Science
University of Aarhus†

December 15, 2006

Abstract

Landin's J operator was the first control operator for functional languages. It was specified with an extension of the SECD machine, which was the first abstract machine for functional languages. We present a family of compositional 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 lock step 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 then 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 a reduction semantics for applicative expressions with the J operator, based on Curien's original calculus of explicit substitutions. This reduction semantics mechanically corresponds to the modernized version of the SECD machine and to the best of our knowledge, it provides the first syntactic theory 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 (a) to account for the first control operator and the first abstract machine for functional languages and (b) to connect them to their successors.

(A preliminary version appears in the proceedings of IFL 2005 [38].)

*Revised version of BRICS RS-06-4.

†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: <danvy@daimi.au.dk>, <kmillikin@daimi.au.dk>

Contents

1 Introduction

Forty years ago, Peter Landin unveiled the first control operator, J, to a heretofore unsuspecting world [67, 68, 70]. 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

$$\textbf{begin}\ s_1\ ;\ \textbf{goto}\ \mathrm{L}\ ;\ \mathrm{L}:\ s_2\ \textbf{end}$$

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

$$\lambda().\textbf{let}\ \mathrm{L} = \mathbf{J}\ s_2'\ \textbf{in let}\ () = s_1'\ ()\ \textbf{in}\ \mathrm{L}\ ()$$

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 [66, 69].

Over the years, the SECD machine has been the topic of considerable study [1, 2, 5, 9–11, 18–20, 22–24, 42, 43, 45–47, 53–55, 57–60, 62, 65, 73, 74, 79, 80, 82–85, 89, 90, 99, 100, 102]. Our angle here is derivational; it follows the 'rational deconstruction' of the SECD machine into a compositional evaluation function presented by Danvy at IFL'04 [30]. This deconstruction laid the ground for a functional correspondence between evaluators and abstract machines [3, 4,6,7,13,16,30,31]. Our goal here is to show that this functional correspondence also applies to the SECD machine with the J operator, which too can be deconstructed into a compositional evaluation function. As a corollary, we present several simulations of the J operator as well as the first reduction semantics for applicative expressions with 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 follow the order of the first deconstruction [30], though with a twist: in the middle of the derivation, we abandon the stack-threading, callee-save features of the SECD machine, which are non-standard, for the more familiar stackless, caller-save features of traditional definitional interpreters [50, 76, 86, 94]. (These points are reviewed in Appendix.)

The SECD machine is defined as the iteration of a state-transition function operating 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 first deconstruction 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. Since Landin's work, the C and D components of his abstract machine have been unified into one component, and reflecting this unification, control operators capture both what used to be C and D instead of only what used to be D.

The definition of `run` looks complicated because it has several induction variables, i.e., it dispatches on several components of the quadruple. The deconstruction proceeds as follows:

- We disentangle `run` into four mutually recursive transition functions, each of which has one induction variable, i.e., dispatches on 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 [32, 39, 86]:[1] 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 [30], the resulting program is an interpreter in continuation-passing style (CPS).[2] This interpreter threads a data stack and uses a callee-save convention to process subterms. (See Appendices A, B, and C.)

- In order to focus on the nature of the J operator, we eliminate the data stack and adopt the more familiar caller-save convention (renaming `run_t` as `eval` and `run_a` as `apply` in passing):

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

The interpreter is still in CPS.

- 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 [33].[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
```

[1] In the early 1970's [86], John Reynolds introduced defunctionalization as a variation of Landin's 'function closures' [66], 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 on the tags. The left inverse of defunctionalization is 'refunctionalization' [32, 39].

[2] The term 'CPS' is due to Steele [93]. In a CPS program, all calls are tail calls and functions thread a functional accumulator, the continuation, that represents 'the rest of the computation' [97]. CPS programs are either written directly or the result of a CPS transformation [34, 84]. CPS-transforming a program twice yields two layers of continuations, like here: `C` is the first layer and `D` is the second [33]. (This programming pattern is also used for 'success' and 'failure' continuations in the functional approach to backtracking.) The left inverse of the CPS transformation is the direct-style transformation [28, 36].

[3] Delimited continuations (e.g., a success continuation) represent part of the rest of the computation: the control operator reset delimits control and the control operator shift captures the current delimited continuation.

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 [13, 33, 41, 63, 78]. 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.

There is plenty of room for variation in the present deconstruction. 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 way. Each step is reversible: one can CPS-transform and defunctionalize an evaluator and (re)construct an abstract machine [3, 4, 6, 7, 13, 16, 30, 31].

1.2 Prerequisites and domain of discourse

Up to Section 2.3, we 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 [32, 39, 86] and its left inverse, refunctionalization; of the CPS transformation [33, 36, 50, 76, 86, 93] and its left inverse, the direct-style transformation; and of delimited continuations [13, 33, 41, 48, 63]. From Section 2.4, 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 total and therefore throughout, we call them as if they were written in direct style [35].

Values. There are five kinds of values: integers, the successor function, function closures, "state appenders" [20, 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)
```

The starting specification: Several formulations of the SECD machine with the J operator have been published [20, 45, 68]. We take the most recent one, i.e., Felleisen's [45], as our starting point, and we consider the others in Section 4:

```
(*  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)                                        (* 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 ((STATE_APPENDER d') :: v :: s, e, APPLY :: c, d)               (* 2 *)
    = run ((PGMCLO (v, d')) :: s, e, c, d)
  | run ((PGMCLO (v, d')) :: v' :: s, e, APPLY :: c, d)                 (* 3 *)
    = run (v :: v' :: nil, e_init, APPLY :: nil, d')

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

[4]The terms 'function closures' and 'program closures' are due to Landin [68]. The term 'state appender' is due to Burge [20]. The term 'continuation' is due to Wadsworth [101]. The term 'first-class' is due to Strachey [96]. The term 'first-class continuation' is due to Friedman and Haynes [49].

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.

1.3 Overview

We first detail the deconstruction of Felleisen's version of the SECD machine into a compositional evaluator in direct style (Section 2). 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 analyze the J operator (Section 3), review related work (Section 4), outline the deconstruction of Burge's version of the SECD machine (Section 5), present a reduction semantics for the J operator (Section 6), and conclude (Sections 7 and 8).

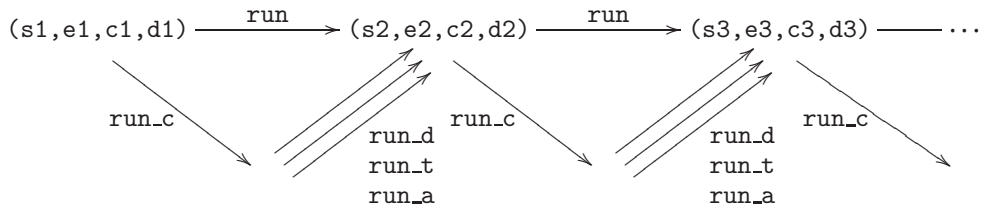
2 Deconstruction of the SECD machine with the J operator

2.1 A disentangled specification

In the starting specification of Section 1.2, all the possible transitions are meshed together in one recursive function, `run`. As in the first rational deconstruction [30], 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 current 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 1 (full correctness) *Given a program,* `evaluate0` *and* `evaluate1` *either both diverge or both yield values that are structurally equal.*

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`, are in the image of defunctionalization (`run_d` and `run_c` are the apply functions of these two data types). After refunctionalization and β_v-contraction,[5] the higher-order evaluator reads as follows; it is higher-order because `c` and `d` now denote functions:

[5]The resulting higher-order evaluator contains four β_v-redexes. Contracting these redexes corresponds to short-circuiting state transitions in the abstract machine, as done in the first rational deconstruction [30].

```
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 *)


(*  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))
  | 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:

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

2.3 A modernized, caller-save and stackless version

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 the more familiar caller-save, stackless form [50, 76, 86, 94]. Let us describe this modernization in two steps: first we transform the evaluator to use a caller-save convention for environments (as also illustrated in Appendices A and B), and second we transform it to not use a data stack (as also illustrated in Appendices A and C).

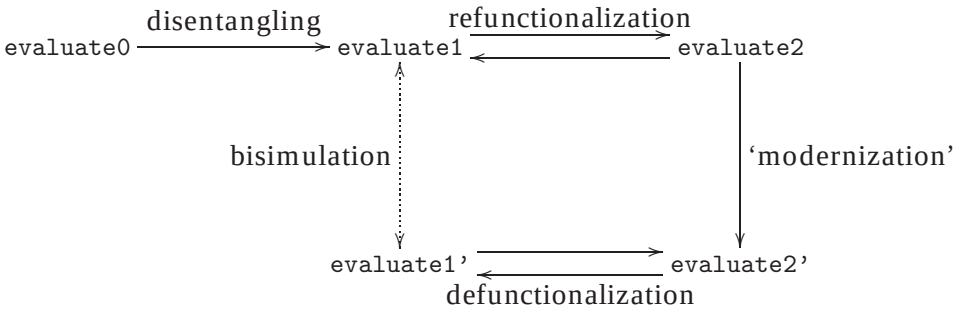
The environments of the evaluator of Section 2.2 are callee-save because the apply function `run_a` receives an environment `e` as an argument and "returns" one to its continuation `c` [8, pages 404–408]. Inspecting the evaluator shows that whenever `run_a` is passed a `c` and a `e` and applies `c`, `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 -> value` to `value * S * E * D -> 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.

The caller-save, stackless counterpart of the evaluator of Section 2.2 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 * D -> value                         (* control continuation *)

(*  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 6.3; 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 [75]. Graphically:



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

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

2.4 A dump-less direct-style counterpart

The evaluator of Section 2.3 is in continuation-passing style, and therefore it is in the image of the CPS transformation. The clause for J captures the current continuation (i.e., the dump) in a state appender, and therefore its direct-style counterpart naturally uses call/cc [36]. With an eye on our next step, we do not, however, use call/cc but its cousins shift and reset [13, 33, 41] 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 [41, 48]: 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:

```
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 *)

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. CPS-transforming this evaluator yields the evaluator of Section 2.3:

Proposition 4 (full correctness) *Given a program,* `evaluate2'` *and* `evaluate3'` *either both diverge or both yield values that are structurally equal.*

2.5 A control-less direct-style counterpart

The evaluator of Section 2.4 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 the occurrences of shift and reset. This pattern of control is characteristic of the CPS hierarchy [13, 33, 41, 63]. 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 [41, 48]: reset_2 delimits the (now twice implicit) dump continuation in `eval`; 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 `eval` and in `apply`, and corresponds to its initialization with the identity function; and shift_1 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 *)

structure SR1 = make_Shift_and_Reset (type intermediate_answer = value)

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

(*  eval  :      term * E -> value                                          *)
(*  apply : value * value -> value                                         *)
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 (STATE_APPENDER d, v)
    = PGMCLO (v, d)
  | apply (PGMCLO (v, d), v')
    = SR1.shift (fn c' => SR2.shift (fn d' =>
        d (SR1.reset (fn () => apply (v, v')))))

fun evaluate4' t                          (*  evaluate4' : 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 2.4:

Proposition 5 (full correctness) *Given a program,* `evaluate3'` *and* `evaluate4'` *either both diverge or both yield values that are structurally equal.*

2.6 A compositional counterpart

We now turn to the data flow of the evaluator of Section 2.5. As for the SECD machine without J [30], 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
withtype E = value Env.env

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

structure SR1 = make_Shift_and_Reset (type intermediate_answer = value)

structure SR2 = make_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
```

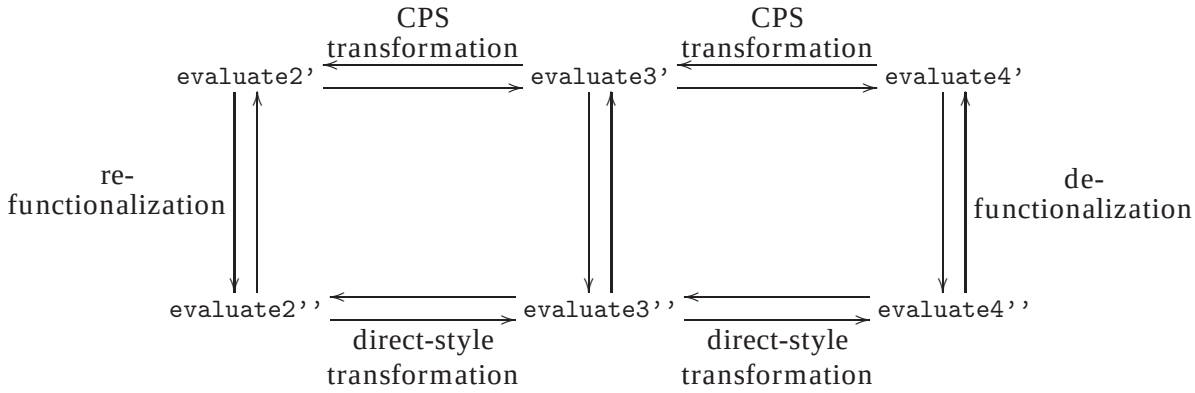
```
  | eval (J, e)
    = SR1.shift (fn c => SR2.shift (fn d =>
        d (c (FUN (fn (FUN f) =>
                FUN (fn v' => SR1.shift (fn c' =>
                                SR2.shift (fn d' =>
                                  d (SR1.reset (fn () => f v'))))))))))

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

This evaluation function is compositional. Defunctionalizing it yields the evaluator of Section 2.5:

Proposition 6 (full correctness) *Given a program,* `evaluate4'` *and* `evaluate4''` *either both diverge or both yield values that are related by defunctionalization.*

2.7 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. (The ML code for `evaluate2''` and `evaluate3''` is not shown here.)



3 On the J operator

3.1 Three simulations of the J operator

The evaluator of Section 2.6 (`evaluate4''`) and the refunctionalized counterparts of the evaluators of Sections 2.4 and 2.3 (`evaluate3''` and `evaluate2''`) are compositional. They can be viewed as syntax-directed encodings into their meta-language, as embodied in the first Futamura projection [51] and the original approach to denotational semantics [95]. 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 `evaluate4''`:

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

  A program p is translated as $\langle\langle [\![p]\!] \rangle_1\rangle_2$.

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

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

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

- 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 `evaluate2''`:

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

  A program p is translated as $[\![p]\!]''\ (\lambda x.\lambda d.d\ x)\ \lambda x.x$.

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.

Assessment: The first rational deconstruction [30] 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.

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

In the terminology of reflective towers [37], 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 3.1, 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 [25] or with Felleisen's $\mathcal{C}$ operator [44] 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 3.1 uses the pushy control operators $\mathcal{S}_1$ (i.e., $\mathcal{S}$) and $\mathcal{S}_2$. Murthy [78] and Kameyama [63] 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 3.1. 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 ($\langle\cdot\rangle_1$):

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

  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 ($\langle\cdot\rangle$):

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

The corresponding CPS simulation of J with two layers of continuations coincides with the one in Section 3.1.

3.3 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, $1 + (\mathbf{call}/\mathbf{cc}\ \lambda k.10)$ 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:

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

The obvious generalization of call/cc to the CPS hierarchy does not work, however. One needs an abort operator as well in order for $\mathrm{call}/\mathrm{cc}_2$ 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.

3.4 On the design of control operators

We note that replacing $\mathcal{C}$ with $\mathcal{S}$ in Section 3.2 (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 [25, 33, 36, 86, 92]?
2. What is the logical content of this control operator [56, 81]?

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 [67], J provides a generalized return [98, Section 2.1], and
2. the type of $\mathbf{J}\ \lambda x.x$ is the law of the excluded middle [99, 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

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

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. For example, given $C[\,] = (\lambda x_2.\mathrm{succ}\ [\,])\ 10$, $t_0 = \mathbf{J}\ (\lambda k.k)\ 0$, and $t_1 = 100$, $C[t_0\ t_1]$ yields 0 whereas $C[\textbf{let}\ x_1 = t_1\ \textbf{in}\ t_0\ x_1]$ yields 1.

4 Related work

4.1 Landin and Burge

Landin [68] 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 [67], 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 [20, 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 [98] gave a detailed introduction to the J operator as presented by Landin and Burge. Burstall [21] 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.

4.2 Reynolds

Reynolds [86] gave a comparison of J to escape, the binder form of Scheme's call/cc [25].[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:

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

As Thielecke notes [98], 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:

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

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

From the point of view of the rational deconstruction of Section 2, 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.

4.3 Felleisen

Felleisen showed how to embed Landin's extension of applicative expressions with J into the Scheme programming language [45]. 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, similarly to the dynamically bound identifier 'self' in an embedding of Smalltalk into Scheme [70]. The control aspect of J is handled through Scheme's control operator call/cc.

As pointed out by Thielecke [98], 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 ($\langle\cdot\rangle$):

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

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

- In CPS:

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

  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.

4.4 Felleisen and Burge

Felleisen's version of the SECD machine with the J operator differs from Burge's. In the notation of Section 1.2, 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 [17].

Burge's version, however, is not in defunctionalized form. In Section 5, we put it in defunctionalized form without resorting to a control delimiter and we outline the corresponding compositional evaluation functions and simulations.

5 An alternative deconstruction

5.1 Our starting point: Burge's specification

As pointed out in Section 4.4, 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 1.2:

```
(*  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.

5.2 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 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 2.1:

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

5.3 A higher-order counterpart

In the modified specification of Section 5.2, 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 *)

(*  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 5.2:

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

5.4 The rest of the rational deconstruction

The evaluator of Section 5.3 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.

5.5 Three alternative simulations of the J operator

As in Section 3.1, the compositional counterpart of the evaluators of Section 5.4 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:

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

  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:

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

  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:

$$\begin{aligned}
[\![n]\!]'' &= \lambda c.\lambda d.c\ n\ d \\
[\![x]\!]'' &= \lambda c.\lambda d.c\ x\ d \\
[\![t_0\ t_1]\!]'' &= \lambda c.\lambda d.[\![t_1]\!]''\ (\lambda x_1.\lambda d.[\![t_0]\!]''\ (\lambda x_0.\lambda d.x_0\ x_1\ c\ d)\ d)\ d
\end{aligned}$$

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

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.

5.6 Related work

Kiselyov's encoding of dynamic delimited continuations in terms of the static delimited-continuation operators shift and reset [64] 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. In addition though, Kiselyov uses a recursive meta-continuation in order to encode dynamic delimited continuations.

6 A syntactic theory of applicative expressions with the J operator

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, 30, 31], a syntactic correspondence exists between calculi and abstract machines, as investigated by Biernacka, Danvy, and Nielsen [12, 14, 15, 29, 31, 40]. 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 one-step reduction function and the reduction semantics that correspond to the modernized SECD machine of Section 2.3. We successively present this machine (Section 6.1), the syntactic correspondence (Section 6.2), and finally the reduction semantics and the one-step reduction function corresponding to this machine (Section 6.3).

6.1 The stackless, caller-save version of the SECD machine with the J operator

The terms, values, environments, and contexts are defined as in Section 1.2:

$$\begin{array}{rl}
\text{(terms)} & t ::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t\ t \mid J \\
\text{(values)} & v ::= \ulcorner n \urcorner \mid \mathrm{SUCC} \mid (\lambda x.t, s) \mid \ulcorner D \urcorner \circ v \mid \ulcorner D \urcorner \\
\text{(environments)} & s ::= \varnothing \mid (x, v) \cdot s \\
\text{(control contexts)} & C ::= [\,] \mid C[(t, s)\ [\,]] \mid C[[\,]\ v] \\
\text{(dump contexts)} & D ::= \bullet \mid C \cdot D
\end{array}$$

The following four transition functions are the stackless, caller-save respective counterparts of `run_c`, `run_d`, `run_t`, and `run_a` in Section 2.1. It is implemented by `evaluate1'` at the end of Section 2.3:

$$\begin{array}{rcll}
\langle [\,], v, D\rangle & \Rightarrow_J & \langle D, v\rangle & \\
\langle C[(t, s)\ [\,]], v, D\rangle & \Rightarrow_J & \langle t, s, C[[\,]\ v], D\rangle & \\
\langle C[[\,]\ v'], v, D\rangle & \Rightarrow_J & \langle v, v', C, D\rangle & \\
 & & & \\
\langle \bullet, v\rangle & \Rightarrow_J & v & \\
\langle C \cdot D, v\rangle & \Rightarrow_J & \langle C, v, D\rangle & \\
 & & & \\
\langle \ulcorner n \urcorner, s, C, D\rangle & \Rightarrow_J & \langle C, \ulcorner n \urcorner, D\rangle & \\
\langle x, s, C, D\rangle & \Rightarrow_J & \langle C, v, D\rangle & \text{if } \mathrm{lookup}(x, s) = v \\
\langle \lambda x.t, s, C, D\rangle & \Rightarrow_J & \langle C, (\lambda x.t, s), D\rangle & \\
\langle t_0\ t_1, s, C, D\rangle & \Rightarrow_J & \langle t_1, s, C[(t_0, s)\ [\,]], D\rangle & \\
\langle J, s, C, D\rangle & \Rightarrow_J & \langle C, \ulcorner D \urcorner, D\rangle & \\
 & & & \\
\langle \mathrm{SUCC}, \ulcorner n \urcorner, C, D\rangle & \Rightarrow_J & \langle C, \ulcorner n + 1 \urcorner, D\rangle & \\
\langle (\lambda x.t, s), v, C, D\rangle & \Rightarrow_J & \langle t, s', [\,], C \cdot D\rangle & \text{where } s' = \mathrm{extend}(x, v, s) \\
\langle \ulcorner D' \urcorner \circ v', v, C, D\rangle & \Rightarrow_J & \langle v, v', [\,], D'\rangle & \\
\langle \ulcorner D' \urcorner, v, C, D\rangle & \Rightarrow_J & \langle C, \ulcorner D' \urcorner \circ v, D\rangle &
\end{array}$$

This machine evaluates a program t by starting in the configuration $\langle t, (\mathrm{succ}, \mathrm{SUCC}) \cdot \varnothing, [\,], \bullet\rangle$. It halts with a value v if it reaches a configuration $\langle \bullet, v\rangle$.

6.2 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 [44]. A one-step reduction function is defined as the composition of three functions:

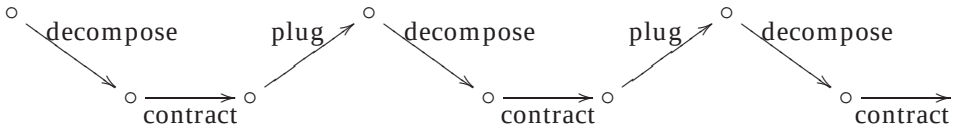
decomposition: a total function mapping a value term to itself and decomposing a non-value term into a potential redex and a reduction context (decomposition is a function because of the unique-decomposition property);

contraction: a partial function mapping an actual redex to its contractum; and

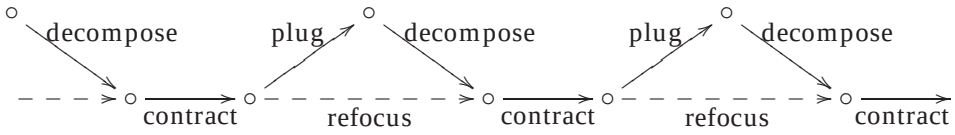
plugging: a total function mapping a term and a reduction context to a new term by filling the hole in the context with the term.

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 [40].

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 [27], 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) [15]. They then turned to context-sensitive contraction functions, as first proposed by Felleisen [44], 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, i/o, stack inspection, proper tail-recursion, and lazy evaluation [14].

The next section presents the calculus of closures corresponding to the abstract machine of Section 6.1.

6.3 A reduction semantics for applicative expressions with the J operator

The $\lambda\widehat{\rho}$J-calculus is an extension of Biernacka and Danvy's $\lambda\widehat{\rho}$-calculus (itself a minimal extension of Curien's original calculus of closures $\lambda\rho$ to make it closed under one-step reduction [15, 27]). It uses names rather than de Bruijn indices, and features two layers of contexts, C and D, that embody the right-to-left applicative-order reduction strategy. C is the control context and D is the dump context. $\ulcorner D \urcorner$ and $\ulcorner D \urcorner \circ v$ respectively denote a state appender and a program closure.

$$
\begin{array}{rl}
\text{(terms)} & t ::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t\,t \mid J \\
\text{(closures)} & c ::= \ulcorner n \urcorner \mid \mathrm{SUCC} \mid t[s] \mid c\,c \mid \ulcorner D \urcorner \mid \ulcorner D \urcorner \circ v \\
\text{(values)} & v ::= \ulcorner n \urcorner \mid \mathrm{SUCC} \mid (\lambda x.t)[s] \mid \ulcorner D \urcorner \mid \ulcorner D \urcorner \circ v \\
\text{(potential redexes)} & r ::= x[s] \mid v\,v \mid J \\
\text{(substitutions)} & s ::= \varnothing \mid (x, v) \cdot s \\
\text{(control contexts)} & C ::= [\,] \mid C[c\,[\,]] \mid C[[\,]\,v] \\
\text{(dump contexts)} & D ::= \bullet \mid C \cdot D
\end{array}
$$

The notions of reduction are specified by the following context-sensitive contraction rules over actual redexes:

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

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.

The one-step reduction function $\mapsto_J$ is defined as the composition of three functions:

decomposition: a non-value closure is decomposed into a potential redex, a control context C, and a dump context D; this function is defined by induction over a closure, the control context, and the dump context;

contraction: an actual redex is contracted as specified just above; and

plugging: plugging a closure in the two layered contexts is defined by induction over these two contexts.

The iteration of $\mapsto_J$ defines an evaluation function. As abundantly illustrated elsewhere [14, 15], deforesting the intermediate terms yields a refocused evaluation function in the form of an abstract machine. Simplifying this machine (again as abundantly illustrated elsewhere [14, 15]) precisely yields the caller-save, stackless SECD abstract machine of Section 6.1.

The following proposition, whose proof is routine [14, 15], captures the raison d'être of this reduction semantics:

Proposition 9 (syntactic correspondence) *For any program* t *in the* $\lambda\widehat{\rho}$J*-calculus,*

$$t[(\mathrm{succ}, \mathrm{SUCC})\cdot\varnothing] \mapsto_J^* v \quad \textit{if and only if} \quad \langle t,\, (\mathrm{succ}, \mathrm{SUCC})\cdot\varnothing,\, [\,],\, \bullet\rangle \Rightarrow_J^* v.$$

Together, 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.

7 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, in the form of abstract machines and compositional evaluation functions, all of which are new. We have also presented the first syntactic theory 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.

The SECD machine and the J operator were the first of their kind. Architecturally, the SECD machine has been superseded by abstract machines with a single control component instead of two (namely C and D). Programmatically, 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), even though it is simple to simulate escape and call/cc in terms of J. Yet as we have shown here, both the SECD machine and the J operator fit in the functional correspondence [3,4,6,7,13,16,30,31] as well as in the syntactic correspondence [12, 14,15,29,31,40], which made it possible for us to mechanically characterize them in new and precise ways.

8 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 [68, page 133]

It was Strachey who coined the term "first-class functions" [96, Section 3.5.1].[7] In turn it was Landin who, through the J operator, invented what we know today as first-class continuations [49]. 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.[8]

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 [87]; 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' 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 [87]. 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.

[7] *"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 [72, page 75]

[8] *"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 [71, Section 1]

In an earlier work [30], 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 [86]. 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 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 8 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 and by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

Appendices

These 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 A), a callee-save, stackless evaluator and the corresponding abstract machine (Appendix B), and a caller-save, stack-threading evaluator and the corresponding abstract machine (Appendix C).

A A caller-save, stackless evaluator and the corresponding abstract machine

The following evaluator for the pure call-by-value λ-calculus (i.e., the language of Section 1.2 without constants and the J operator) is standard. As pointed out by Reynolds [86], 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 `t0` is evaluated, the environment is implicitly saved in the context in order to evaluate `t1` later on. In other words, it is solely a synthesized attribute.

As is well known since Reynolds [4, 86], 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 [46], i.e., an eval-apply abstract machine where the evaluation contexts and the apply transition function are the defunctionalized counterparts of the continuations of the evaluator just above:

$$\begin{aligned}
\text{(terms)} \quad & t ::= x \mid \lambda x.t \mid t\ t \\
\text{(values)} \quad & v ::= [x, t, e] \\
\text{(environments)} \quad & s ::= \varnothing \mid (x, v) \cdot s \\
\text{(contexts)} \quad & k ::= \mathsf{END} \mid \mathsf{ARG}(t, e, k) \mid \mathsf{FUN}(v, k)
\end{aligned}$$

$$\begin{aligned}
\langle x,\ e,\ k\rangle &\Rightarrow_{CEK} \langle k,\ v\rangle && \text{if } lookup(x, e) = v \\
\langle \lambda x.t,\ e,\ k\rangle &\Rightarrow_{CEK} \langle k,\ [x, t, e]\rangle \\
\langle t_0\ t_1,\ e,\ k\rangle &\Rightarrow_{CEK} \langle t_0,\ e,\ \mathsf{ARG}(t_1, e, k)\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \mathsf{END},\ v\rangle &\Rightarrow_{CEK} v \\
\langle \mathsf{ARG}(t, e, k),\ v\rangle &\Rightarrow_{CEK} \langle t,\ e,\ \mathsf{FUN}(v, k)\rangle \\
\langle \mathsf{FUN}([x, t, e], k),\ v\rangle &\Rightarrow_{CEK} \langle t,\ e',\ k\rangle && \text{where } e' = extend(x, v, e)
\end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle t, \varnothing, \mathsf{END}\rangle$. It halts with a value v if it reaches a configuration $\langle \mathsf{END}, v\rangle$.

B A callee-save, stackless evaluator and the corresponding abstract machine

The following evaluator is a callee-save version of the evaluator of Appendix A. Whereas the evaluator of Appendix A 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 `t0`. In other words, the environment is not solely an inherited attribute as in the evaluator of Appendix A: 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 A, this evaluator is not properly tail recursive since the evaluation of the body of a function abstraction no longer occurs in tail position [26, 85].

As in Appendix A, 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:

$$\text{(contexts)}\quad \mathrm{k} ::= \mathsf{END} \mid \mathsf{ARG}(\mathrm{t}, \mathrm{k}) \mid \mathsf{FUN}(v, \mathrm{k}) \mid \mathsf{RET}(e, \mathrm{k})$$

$$\begin{aligned}
\langle \mathrm{x},\, e,\, \mathrm{k}\rangle &\Rightarrow_{CEK_E} \langle \mathrm{k},\, v,\, e\rangle && \text{if } \mathit{lookup}(\mathrm{x}, e) = v\\
\langle \lambda \mathrm{x}.\mathrm{t},\, e,\, \mathrm{k}\rangle &\Rightarrow_{CEK_E} \langle \mathrm{k},\, [\mathrm{x}, \mathrm{t}, e],\, e\rangle\\
\langle \mathrm{t}_0\ \mathrm{t}_1,\, e,\, \mathrm{k}\rangle &\Rightarrow_{CEK_E} \langle \mathrm{t}_0,\, e,\, \mathsf{ARG}(\mathrm{t}_1, \mathrm{k})\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \mathsf{END},\, v,\, e\rangle &\Rightarrow_{CEK_E} v\\
\langle \mathsf{ARG}(\mathrm{t}, \mathrm{k}),\, v,\, e\rangle &\Rightarrow_{CEK_E} \langle \mathrm{t},\, e,\, \mathsf{FUN}(v, \mathrm{k})\rangle\\
\langle \mathsf{FUN}([\mathrm{x}, \mathrm{t}, e'], \mathrm{k}),\, v,\, e\rangle &\Rightarrow_{CEK_E} \langle \mathrm{t},\, e'',\, \mathsf{RET}(e, \mathrm{k})\rangle \text{ where } e'' = \mathit{extend}(\mathrm{x}, v, e')\\
\langle \mathsf{RET}(e', \mathrm{k}),\, v,\, e\rangle &\Rightarrow_{CEK_E} \langle \mathrm{k},\, v,\, e'\rangle
\end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle \mathrm{t},\, \varnothing,\, \mathsf{END}\rangle$. It halts with a value v if it reaches a configuration $\langle \mathsf{END},\, v,\, e\rangle$.

Compared to the CEK machine, 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.

C A caller-save, stack-threading evaluator and the corresponding abstract machine

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.[9]

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 or a global data stack, then functional values are passed their argument and a data stack, and return a value and a data stack. One can choose

[9] If 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.

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.

As in Appendix B, 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 A and 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 another variant of the CEK machine with a data stack; its terms, values, and environments remain the same:

$$\text{(contexts)}\quad k ::= \mathsf{END} \mid \mathsf{ARG}(t, e, k) \mid \mathsf{FUN}(k) \mid \mathsf{RET}(s, k)$$

$$\begin{aligned}
\langle x,\ s,\ e,\ k\rangle &\Rightarrow_{CEK_S} \langle k,\ v :: s\rangle && \text{if } \mathit{lookup}(x, e) = v\\
\langle \lambda x.t,\ s,\ e,\ k\rangle &\Rightarrow_{CEK_S} \langle k,\ [x, t, e] :: s\rangle\\
\langle t_0\ t_1,\ s,\ e,\ k\rangle &\Rightarrow_{CEK_S} \langle t_0,\ s,\ e,\ \mathsf{ARG}(t_1, e, k)\rangle\\
\\
\langle \mathsf{END},\ v :: s\rangle &\Rightarrow_{CEK_S} v\\
\langle \mathsf{ARG}(t, e, k),\ s\rangle &\Rightarrow_{CEK_S} \langle t,\ s,\ e,\ \mathsf{FUN}(k)\rangle\\
\langle \mathsf{FUN}(k),\ v :: [x, t, e] :: s\rangle &\Rightarrow_{CEK_S} \langle t,\ \mathsf{nil},\ e',\ \mathsf{RET}(s, k)\rangle && \text{where } e' = \mathit{extend}(x, v, e)\\
\langle \mathsf{RET}(s', k),\ v :: s\rangle &\Rightarrow_{CEK_S} \langle k,\ v :: s'\rangle
\end{aligned}$$

This machine evaluates a closed term t by starting in the configuration $\langle t, \mathsf{nil}, \varnothing, \mathsf{END}\rangle$. It halts with a value v if it reaches a configuration $\langle \mathsf{END}, v :: s\rangle$.

Compared to the CEK machine, 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.

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

[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 technical 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 technical 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 2001 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] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, 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 syntactic correspondence between context-sensitive calculi and abstract machines. Technical Report BRICS RS-05-38, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2005. Accepted for publication in Theoretical Computer Science (March 2006).

[15] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the technical report BRICS RS-06-3.

[16] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[17] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 16(3):269–280, 2006.

[18] 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.

[19] 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.

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

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

[22] Luca Cardelli. The functional abstract machine. *Polymorphism*, 1(1), January 1983.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[28] 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).

[29] 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.

[30] 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.

[31] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006.

[32] Olivier Danvy. Refunctionalization at work. In *Preliminary proceedings of the 8th International Conference on Mathematics of Program Construction (MPC '06)*, Kuressaare, Estonia, July 2006. Invited talk.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number 4015 in Lecture Notes in Computer Science, pages 55–73, Dublin, Ireland, September 2005. Springer-Verlag.

[39] 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.

[40] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[41] 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.

[42] Antony J. T. Davie. *Introduction to Functional Programming Systems Using Haskell*, volume 27 of *Cambridge Computer Science Texts*. Cambridge University Press, 1992.

[43] 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.

[44] 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.

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

[46] 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.

[47] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[48] 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.

[49] 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.

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

[51] 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 [52].

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

[53] Michael Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Transactions on Programming Languages and Systems*, 6(4):603–631, 1984.

[54] Carsten K. Gomard and Peter Sestoft. Globalization and live variables. In Hudak and Jones [61], pages 166–177.

[55] Brian T. Graham. *The SECD microprocessor: a verification case study*. Kluwer Academic Publishers, 1992.

[56] 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.

[57] John Hannan. Staging transformations for abstract machines. In Hudak and Jones [61], pages 130–141.

[58] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[59] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] Werner E. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

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

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

[68] 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 [98].

[69] 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.

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

[71] 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.

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

[73] 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.

[74] Erik Meijer. Generalised expression evaluation. Technical Report 88-5, Department of Informatics, University of Nijmegen, Nijmegen, The Netherlands, 1988.

[75] Kevin Millikin. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2007. Forthcoming.

[76] 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.

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

[78] 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.

[79] Peter Møller Neergaard. *Complexity Aspects of Programming Language Design—From Logspace to Elementary Time via Proofnets and Intersection Types*. PhD thesis, Mitchom School of Computer Science, Brandeis University, Waltham, Massachusetts, October 2004.

[80] 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.

[81] Michel Parigot. λμ-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.

[82] 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.

[83] 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.

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

[85] John D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.

[86] 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 [88].

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

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

[89] Colin Runciman and Ian Toyn. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4(4):339–363, 1986.

[90] 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.

[91] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[92] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 2007. Journal version of [91]. To appear.

[93] 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.

[94] 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.

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

[96] 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 [77].

[97] 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 [101].

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

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

[100] Vasco Thudichum Vasconcelos. Lambda and pi calculi, CAM and SECD machines. *Journal of Functional Programming*, 15(1):101–127, 2005.

[101] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1 / 2):131–133, 2000.

[102] 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.

Recent BRICS Report Series Publications

RS-06-17 **Olivier Danvy and Kevin Millikin.** ***A Rational Deconstruction of Landin's J Operator*****. December 2006. ii+37 pp. Revised version of BRICS RS-06-4. A preliminary version appears in the proceedings of IFL 2005, LNCS 4015:55–73.**

RS-06-16 **Anders Møller.** ***Static Analysis for Event-Based XML Processing*****. October 2006. 16 pp.**

RS-06-15 **Dariusz Biernacki, Olivier Danvy, and Kevin Millikin.** ***A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*****. October 2006. ii+28 pp. Revised version of BRICS RS-05-16.**

RS-06-14 **Giorgio Delzanno, Javier Esparza, and Jiří Srba.** ***Monotonic Set-Extended Prefix Rewriting and Verification of Recursive Ping-Pong Protocols*****. July 2006. 31 pp. To appear in ATVA '06.**

RS-06-13 **Jiří Srba.** ***Visibly Pushdown Automata: From Language Equivalence to Simulation and Bisimulation*****. July 2006. 21 pp. To appear in CSL '06.**

RS-06-12 **Kristian Støvring.** ***Higher-Order Beta Matching with Solutions in Long Beta-Eta Normal Form*****. June 2006. 13 pp. To appear in *Nordic Journal of Computing*, 2006.**

RS-06-11 **Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski.** ***An Interface Theory for Input/Output Automata*****. June 2006. 40 pp. Appears in Misra, Nipkow and Sekerinski, editors, *Formal Methods: 14th International Symposium*, FM '06 Proceedings, LNCS 4085, 2006, pages 82–97.**

RS-06-10 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP*****. June 2006. 23 pp. Full version of paper presented at SAS '06.**

RS-06-9 **Claus Brabrand, Robert Giegerich, and Anders Møller.** ***Analyzing Ambiguity of Context-Free Grammars*****. April 2006. 19 pp.**

RS-06-8 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP*****. April 2006. 22 pp.**

RS-06-7 **Petr Jančar and Jiří Srba.** ***Undecidability Results for Bisimilarity on Prefix Rewrite Systems*****. April 2006. 20 pp. Presented at *FoSSaCS 2006*, LNCS 3921:277–291.**