Defunctionalized Interpreters for Programming Languages

Olivier Danvy
Department of Computer Science, University of Aarhus *
danvy@brics.dk

Abstract

This document illustrates how functional implementations of formal semantics (structural operational semantics, reduction semantics, small-step and big-step abstract machines, natural semantics, and denotational semantics) can be transformed into each other. These transformations were foreshadowed by Reynolds in "Definitional Interpreters for Higher-Order Programming Languages" for functional implementations of denotational semantics, natural semantics, and big-step abstract machines using closure conversion, CPS transformation, and defunctionalization. Over the last few years, the author and his students have further observed that functional implementations of small-step and of big-step abstract machines are related using fusion by fixed-point promotion and that functional implementations of reduction semantics and of small-step abstract machines are related using refocusing and transition compression. It furthermore appears that functional implementations of structural operational semantics and of reduction semantics are related as well, also using CPS transformation and defunctionalization. This further relation provides an element of answer to Felleisen's conjecture that any structural operational semantics can be expressed as a reduction semantics: for deterministic languages, a reduction semantics is a structural operational semantics in continuation style, where the reduction context is a defunctionalized continuation. As the defunctionalized counterpart of the continuation of a one-step reduction function, a reduction context represents the rest of the reduction, just as an evaluation context represents the rest of the evaluation since it is the defunctionalized counterpart of the continuation of an evaluation function.

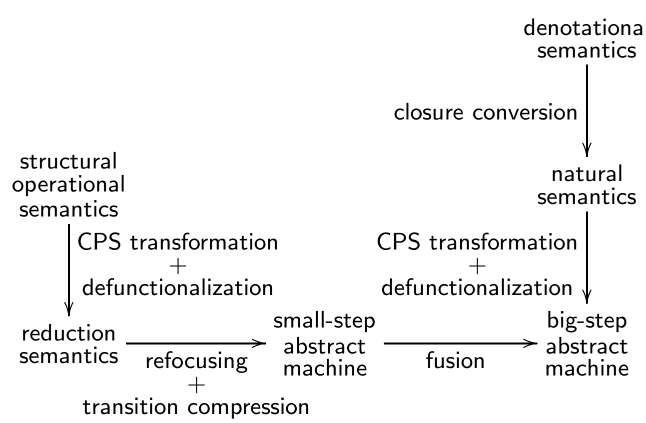
Categories and Subject Descriptors D.1.1 [*Software*]: Programming Techniques—Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.1.1 [*Theory of Computation*]: Computation by Abstract Devices—Models of Computation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational Semantics

General Terms relations between models

Keywords big-step abstract machines, context-sensitive reduction semantics, continuations, CPS transformation, defunctionalization, interruptions, natural semantics, reduction semantics, refocusing, small-step abstract machines, structural operational semantics

1. Introduction

Witness interpreters, compilers, and partial evaluators, and going all the way back to Alan Turing's point of simulating a virtual machine with another one [58] as well as all the way forth to the POPLmark Challenge today [6], there is simply nothing like computation to describe computation. And indeed convenient and expressive meta-languages (e.g., the lambda-calculus and propositional logic) tend to migrate into the realm of programming languages (e.g., for functional programming and for logic programming). Consequently, formalisms used to specify the semantics of programming languages—structural operational semantics (Plotkin [52]), reduction semantics (Felleisen [29]), small-step and big-step abstract machines (Winskel [65]), natural semantics (Kahn [40]), and denotational semantics (Scott and Strachey [57])—can directly be represented as programs. It is the thesis of the author and his students [1, 8, 12, 21, 38, 44–46, 48] that functional implementations of formal semantics are inter-derivable using elementary program transformations. Diagrammatically:



- In "Definitional Interpreters for Higher-Order Programming Languages" [54], John Reynolds initiated the vertical part of this diagram, on the right, structurally relating a higher-order, compositional evaluation function characteristic of a denotational semantics, a first-order, closure-based evaluation function typical of a natural semantics, and first-order mutually recursive transition functions specific to big-step abstract machines. Mads Sig Ager, Małgorzata Biernacka, Dariusz Biernacki, Jan Midtgaard and the author have shown how this functional correspondence between evaluators and abstract machines scales for various evaluation orders and monadic effects [3–5], including, e.g., the language-security technique of properly tail-recursive stack inspection [14] as well as delimited continuations [9].

* IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
http://www.brics.dk/~danvy

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

- In “Refocusing in Reduction Semantics” [27], Lasse Nielsen and the author initiated the horizontal part of the diagram, structurally relating functional representations of reduction semantics and abstract machines. Biernacka and the author have shown how this syntactic correspondence between reduction semantics and abstract machines scales to calculi with explicit substitutions [10]. For example, context-sensitive reduction semantics for Curien’s original calculus of closures [17] give rise to a variety of known and new abstract machines with environments and effects [11]. Recently [25], Kevin Millikin and the author have characterized the structural relation between small-step abstract machines and big-step abstract machines using Atsushi Ohori and Isao Sasano’s lightweight fusion by fixed-point promotion [50].
- In this document, we initiate the vertical left part of the diagram, structurally relating functional representations of structural operational semantics and of reduction semantics. We characterize a reduction semantics as the continuation-semantics analogue of a structural operational semantics. For brevity, we stay away from lambda-languages and binding issues, that with few exceptions [13, 23, 24] have been our favorite domain of discourse so far. Instead, we consider arithmetic expressions with effects (namely interruptions) and stuck terms (namely errors), as in Graham Hutton and Joel Wright’s recent work [37]. In doing so, we aim at the same sort of telling, minimalistic elegance as can be found in Hutton’s publications.

The title “Defunctionalized Interpreters for Programming Languages” is meant as an homage to John Reynolds for his prescient article “Definitional Interpreters for Higher-Order Programming Languages” [54] as well as an appreciative emphasis on the defunctionalization technique he introduced there. (The words ‘higher-order’ did not make it in our title because, even with interruptions and errors, arithmetic expressions are first-order entities.)

Prerequisites: We expect a minimal familiarity with Standard ML, which we use as a pure meta-language, and with the format of formal semantics as can be gathered in an undergraduate textbook [49]. We also gamble on the reader’s patience and good will to see small-step semantics spelled out in complete detail.

Overview: Section 2 is dedicated to arithmetic expressions with interruptions and errors. We successively present a computational basis for interruptions (Section 2.1), the abstract syntax of the arithmetic expressions and its associated syntactic values (Section 2.2), a notion of contraction (Section 2.3), and finally the functional implementation of a structural operational semantics (Figure 1). We then transform this functional implementation into continuation-passing style (Section 3 and Figure 2), and split its sum-accepting continuation into a pair of continuations (Section 4 and Figure 3), where we identify the second continuation as the usual plug function that maps a context and a contractum into a term in a reduction semantics. We then defunctionalize this pair of continuations (Figure 4) and refactor the resulting artifact into the functional implementation of a traditional reduction semantics (Figure 5) where the reduction contexts are the data type of the defunctionalized continuations. This functional implementation of a reduction semantics can be refocused into the functional implementation of a small-step abstract machine that in turn can be fused into essentially the functional implementation of the same big-step abstract machine as in Hutton and Wright’s work [37]. Alternatively (Section 6), one can make the contraction function context-sensitive in the functional implementation of the reduction semantics, which dramatically simplifies the corresponding abstract machine (Section 7). We then present some perspectives (Section 8).

2. A structural operational semantics for arithmetic expressions with interruptions and errors

This section builds on Hutton and Wright’s recent article about the meaning of interruptions [37].

2.1 A basis for interruptions

We first abstractly specify a stream of signals that can be polled:

```
signature SIGNALS = sig
  type signals
  val poll : signals -> bool * signals
end

structure Signals : SIGNALS = struct
  (* deliberately omitted *)
end

type interrupts = Signals.signals
```

Polling the stream of signals yields a boolean value and the remaining stream. The boolean value reflects whether the polled signal should be interpreted as an interruption or not.

Whether to poll for interruptions is determined by a global status value. In a blocked state, the stream of signals is not polled, whereas it is polled in an unblocked state:

```
datatype status = B | U
```

2.2 Abstract syntax (terms and values)

2.2.1 Terms

Following Hutton and Wright, arithmetic expressions are equipped with a sequencing operator, a catch operator to intercept interruptions, a throw operator to syntactically represent the effect of an interruption as an exception, a block (resp. unblock) operator to evaluate a term in a blocked (resp. unblocked) state, and an error operator:

```
datatype term = LIT of int
              | ADD of term * term
              | SEQ of term * term
              | CATCH of term * term
              | THROW
              | BLOCK of term
              | UNBLOCK of term
              | ERROR
```

2.2.2 Notion of value

A value stands for an irreducible term. Either it is an integer, as can be expected from having reduced an arithmetic expression, or it is an exception:

```
datatype value = EXPECT of int | EXCEPT
```

The usual embedding from value to term reads as follows:

```
fun v2t (EXPECT n) = LIT n
  | v2t (EXCEPT)   = THROW
```

2.3 Notion of contraction

A potential redex is either an actual redex or it is stuck. Each of the operators in the syntax gives rise to a potential redex: for additions, for sequencing, for intercepting interruptions, etc.:

```
datatype potential_redex = PRSUM of value * value
                         | PRSEQ of value * term
                         | PRCATCH of value * term
                         | PRBLOCK of term
                         | PRUNBLOCK of term
                         | PRERROR
```

```
fun reduce (t, s, is)          (* : term * status * interrupts -> (term * status * interrupts) option *)
= let datatype intermediate_result = STUCK | VALUE of value | TERM of term * status * interrupts
      fun visit (LIT n)                                  (* : term -> (term * status * interrupts) option *)
          = VALUE (EXPECT n)
        | visit (ADD (t1, t2))
          = (case visit t1
               of STUCK               => STUCK
                | VALUE v1            => (case visit t2
                                            of STUCK               => STUCK
                                             | VALUE v2            => (case contract (PRSUM (v1, v2),
                                                                                     s, is)
                                                                          of NONE
                                                                             => STUCK
                                                                           | SOME (t', s', is')
                                                                             => TERM (t', s', is'))
                                             | TERM (t2', s', is') => TERM (ADD (v2t v1, t2'), s', is'))
                | TERM (t1', s', is') => TERM (ADD (t1', t2), s', is'))
        | visit (SEQ (t1, t2))
          = (case visit t1
               of STUCK               => STUCK
                | VALUE v1            => (case contract (PRSEQ (v1, t2), s, is)
                                            of NONE => STUCK
                                             | SOME (t1', s', is') => TERM (t1', s', is'))
                | TERM (t1', s', is') => TERM (SEQ (t1', t2), s', is'))
        | visit (CATCH (t1, t2))
          = (case visit t1
               of STUCK => STUCK
                | VALUE v1            => (case contract (PRCATCH (v1, t2), s, is)
                                            of NONE => STUCK
                                             | SOME (t', s', is') => TERM (t', s', is'))
                | TERM (t1', s', is') => TERM (CATCH (t1', t2), s', is'))
        | visit THROW
          = VALUE EXCEPT
        | visit (BLOCK t)
          = (case contract (PRBLOCK t, s, is)
               of NONE               => STUCK
                | SOME (t', s', is') => TERM (t', s', is'))
        | visit (UNBLOCK t)
          = (case contract (PRUNBLOCK t, s, is)
               of NONE               => STUCK
                | SOME (t', s', is') => TERM (t', s', is'))
        | visit ERROR
          = (case contract (PRERROR, s, is)
               of NONE               => STUCK
                | SOME (t', s', is') => TERM (t', s', is'))
  in case visit t
       of STUCK              => NONE
        | VALUE v            => SOME (v2t v, s, is)
        | TERM (t, s', is') => SOME (t, s', is')
  end
```

Figure 1. Functional implementation of a structural operational semantics: a one-step reduction function in direct style

The individual contractions are performed within a state, and may change the status, but not the current stream of signals:

- an addition maps two expected numbers into their sum; otherwise, either or both of the arguments are unexpected, and the result is a throw operator; the status remains the same;
- sequencing from an expected number to a term yields this term; otherwise, the first argument is unexpected and the result is a throw operator; the status remains the same;
- intercepting an expected number yields this number, whereas intercepting an exception yields the second argument of the catch operator; the status remains the same;
- the blocking (resp. unblocking) operator yields a blocked (resp. unblocked) status, irrespective of the previous status;
- the error operator is stuck and does not modify the status.

All potential redexes but the last one are thus actual redexes and yield a contractum. Performing a contraction therefore optionally maps a potential redex, a status, and a stream of signals into a term, a status, and a stream of signals:

```
fun perform (PRSUM (EXPECT n1, EXPECT n2), s, is)
    = SOME (LIT (n1 + n2), s, is)
  | perform (PRSUM (_, _), s, is)
    = SOME (THROW, s, is)
  | perform (PRSEQ (EXPECT n, t), s, is)
    = SOME (t, s, is)
  | perform (PRSEQ (EXCEPT, t), s, is)
    = SOME (THROW, s, is)
  | perform (PRCATCH (EXPECT n, t), s, is)
    = SOME (LIT n, s, is)
  | perform (PRCATCH (EXCEPT, t), s, is)
    = SOME (t, s, is)
```

```
fun reduce_c (t, s, is)                          (* : term * status * interrupts -> answer *)
= let datatype intermediate_result = VALUE of value | TERM of term * status * interrupts
      fun visit (LIT n, k)                      (* : term * (intermediate_result -> answer) -> answer *)
          = k (VALUE (EXPECT n))                (* where answer = (term * status * interrupts) option *)
        | visit (ADD (t1, t2), k)
          = visit (t1, fn VALUE v1
                          => visit (t2, fn (VALUE v2)
                                            => (case contract (PRSUM (v1, v2), s, is)
                                                  of NONE               => NONE
                                                   | SOME (t', s', is') => k (TERM (t', s', is')))
                                         | (TERM (t2', s', is'))
                                            => k (TERM (ADD (v2t v1, t2'), s', is')))
                        | TERM (t1', s', is')
                          => k (TERM (ADD (t1', t2), s', is')))
        | visit (SEQ (t1, t2), k)
          = visit (t1, fn VALUE v1
                          => (case contract (PRSEQ (v1, t2), s, is)
                                of NONE               => NONE
                                 | SOME (t', s', is') => k (TERM (t', s', is')))
                        | TERM (t1', s', is')
                          => k (TERM (SEQ (t1', t2), s', is')))
        | visit (CATCH (t1, t2), k)
          = visit (t1, fn VALUE v1
                          => (case contract (PRCATCH (v1, t2), s, is)
                                of NONE               => NONE
                                 | SOME (t', s', is') => k (TERM (t', s', is')))
                        | TERM (t1', s', is')
                          => k (TERM (CATCH (t1', t2), s', is')))
        | visit (THROW, k)
          = k (VALUE EXCEPT)
        | visit (BLOCK t, k)
          = (case contract (PRBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => k (TERM (t', s', is')))
        | visit (UNBLOCK t, k)
          = (case contract (PRUNBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => k (TERM (t', s', is')))
        | visit (ERROR, k)
          = (case contract (PRERROR, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => k (TERM (t', s', is')))
  in visit (t, fn VALUE v
                 => SOME (v2t v, s, is)
                | TERM (t, s', is')
                 => SOME (t, s', is'))
  end
```

Figure 2. Continuation-passing counterpart of Figure 1 where the continuation is only applied to a value or a term

```
  | perform (PRBLOCK t, s, is)
    = SOME (t, B, is)
  | perform (PRUNBLOCK t, s, is)
    = SOME (t, U, is)
  | perform (PRERROR, s, is)
    = NONE
```

Overall, the following contraction function optionally maps a potential redex, the current status, and the current stream of signals to a term, a new status, and a new stream of signals, unless the potential redex is stuck. An interruption raises an exception.

```
fun contract (pr, B, is)
    = perform (pr, B, is)
  | contract (pr, U, is)
    = (case Signals.poll is      (* <- polling *)
         of (false, is') (* <- no interruption *)
            => perform (pr, U, is')
          | (true, is')  (* <- an interruption *)
            => SOME (THROW, U, is'))
```

2.4 One-step reduction

We are now in position to write a total one-step reduction function in direct style that implements a structural operational semantics, as displayed in Figure 1. Computationally, a potential redex is recursively searched depth-first and from left to right, and given a term, a status, and a stream of signals, the one-step reduction function produces the following term in the reduction sequence, if there is any, together with a new status and a new stream of signals.

3. CPS transformation

In the one-step reduction function displayed in Figure 1, let us transform `visit` into Continuation-Passing Style [22, 51, 56]. As such, all its intermediate results are named, their computation is sequentialized, and continuations are introduced, yielding a definition where all recursive calls are tail calls. In addition, rather than propagating the stuck intermediate result all the way through the continuation, we directly map it to the final result `NONE`. The resulting continuation-passing reduction function is displayed in Figure 2.

```
fun reduce_c2 (t, s, is)                              (* : term * status * interrupts -> answer *)
= let fun visit (LIT n, kv, kt) (* : term * (value -> answer) * (term * status * interrupts -> answer)*)
          = kv (EXPECT n)       (*                                                        -> answer *)
        | visit (ADD (t1, t2), kv, kt)          (* where answer = (term * status * interrupts) option *)
          = visit (t1,
                   fn v1 => visit (t2,
                                   fn v2 => (case contract (PRSUM (v1, v2), s, is)
                                               of NONE               => NONE
                                                | SOME (t', s', is') => kt (t', s', is')),
                                   fn (t2', s', is') => kt (ADD (v2t v1, t2'), s', is')),
                   fn (t1', s', is') => kt (ADD (t1', t2), s', is'))
        | visit (SEQ (t1, t2), kv, kt)
          = visit (t1,
                   fn v1 => (case contract (PRSEQ (v1, t2), s, is)
                               of NONE               => NONE
                                | SOME (t', s', is') => kt (t', s', is')),
                   fn (t1', s', is') => kt (SEQ (t1', t2), s', is'))
        | visit (CATCH (t1, t2), kv, kt)
          = visit (t1,
                   fn v1 => (case contract (PRCATCH (v1, t2), s, is)
                               of NONE               => NONE
                                | SOME (t', s', is') => kt (t', s', is')),
                   fn (t1', s', is') => kt (CATCH (t1', t2), s', is'))
        | visit (THROW, kv, kt)
          = kv EXCEPT
        | visit (BLOCK t, kv, kt)
          = (case contract (PRBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => kt (t', s', is'))
        | visit (UNBLOCK t, kv, kt)
          = (case contract (PRUNBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => kt (t', s', is'))
        | visit (ERROR, kv, kt)
          = (case (contract (PRERROR, s, is))
               of NONE               => NONE
                | SOME (t', s', is') => kt (t', s', is'))
  in visit (t,
            fn v => SOME (v2t v, s, is),
            fn (t, s', is') => SOME (t, s', is'))
  end
```

Figure 3. Version of Figure 2 where the continuation is split into two

Since the CPS transformation is fully correct [51], `reduce` in Figure 1 and `reduce_c` in Figure 2 implement the same one-step reduction function.

4. Splitting the continuation into two

In Figure 2, the continuation maps an intermediate result to an answer. The intermediate result is a sum, and two radically distinct things happen to each summand: in one case, the input term continues to be traversed in search for a redex; in the other, the output term continues to be constructed. This distinctness prompts us to make use of the type isomorphism between a sum-accepting function and a pair of functions and split the continuation into two:

$$(A_1 + A_2) \rightarrow B \;\;\cong\;\; (A_1 \rightarrow B) \times (A_2 \rightarrow B)$$

The result is displayed in Figure 3. The first continuation is used to continue the search for a redex and the second one is used to map a contractum into the next term in the reduction sequence.

Since splitting the continuation is obviously correct, `reduce_c` in Figure 2 and `reduce_c2` in Figure 3 implement the same one-step reduction function.

5. Defunctionalization

Let us defunctionalize the continuations of Figure 3 [26, 54]. To this end, we partition their function space into a sum. This sum is indexed by each of the functional abstractions **`fn`** `... => ...` that give rise to an inhabitant of that function space. We implement this sum in the data type `reduction_context` in Figure 4. This data type is shared by the two continuations. It is interpreted, for the first one, by the dispatch function `apply_kt`, and for the second one, by the function `apply_kv`. Modulo renaming, each clause of these apply functions is the body of a function abstraction in Figure 3. For the rest, each function abstraction in Figure 3 is replaced by a constructor of the data type `reduction_context`, and each application of a continuation in Figure 3 is replaced by a call to the corresponding apply function. The resulting program is first-order.

Since defunctionalization is fully correct [7,47,53], `reduce_c2` in Figure 3 and `reduce_c2d` in Figure 4 implement the same one-step reduction function.

```
fun reduce_c2d (t, s, is)                              (* : term * status * interrupts -> answer *)
= let datatype reduction_context = C_EMPTY       (* where answer = (term * status * interrupts) option *)
                                 | C_ADD1 of reduction_context * term
                                 | C_ADD2 of value * reduction_context
                                 | C_SEQ of reduction_context * term
                                 | C_CATCH of reduction_context * term
      fun apply_kt (C_EMPTY, (t', s', is'))     (* : context * (term * status * interrupts) -> answer *)
          = SOME (t', s', is')
        | apply_kt (C_ADD1 (c, t2), (t1', s', is'))
          = apply_kt (c, (ADD (t1', t2), s', is'))
        | apply_kt (C_ADD2 (v1, c), (t2', s', is'))
          = apply_kt (c, (ADD (v2t v1, t2'), s', is'))
        | apply_kt (C_SEQ (c, t2), (t1', s', is'))
          = apply_kt (c, (SEQ (t1', t2), s', is'))
        | apply_kt (C_CATCH (c, t2), (t1', s', is'))
          = apply_kt (c, (CATCH (t1', t2), s', is'))

      fun apply_kv (C_EMPTY, v)                                   (* : context * value -> answer *)
          = SOME (v2t v, s, is)
        | apply_kv (C_ADD1 (c, t2), v1)
          = visit (t2, C_ADD2 (v1, c))
        | apply_kv (C_ADD2 (v1, c), v2)
          = (case contract (PRSUM (v1, v2), s, is)
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))
        | apply_kv (C_SEQ (c, t2), v1)
          = (case contract (PRSEQ (v1, t2), s, is)
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))
        | apply_kv (C_CATCH (c, t2), v1)
          = (case contract (PRCATCH (v1, t2), s, is)
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))

      and visit (LIT n, c)                                         (* : term * context -> answer *)
          = apply_kv (c, EXPECT n)
        | visit (ADD (t1, t2), c)
          = visit (t1, C_ADD1 (c, t2))
        | visit (SEQ (t1, t2), c)
          = visit (t1, C_SEQ (c, t2))
        | visit (CATCH (t1, t2), c)
          = visit (t1, C_CATCH (c, t2))
        | visit (THROW, c)
          = apply_kv (c, EXCEPT)
        | visit (BLOCK t, c)
          = (case contract (PRBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))
        | visit (UNBLOCK t, c)
          = (case contract (PRUNBLOCK t, s, is)
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))
        | visit (ERROR, c)
          = (case (contract (PRERROR, s, is))
               of NONE               => NONE
                | SOME (t', s', is') => apply_kt (c, (t', s', is')))
  in visit (t, C_EMPTY)
  end
```

Figure 4. Defunctionalized counterpart of Figure 3, where `kv` and `kt` share the same data type of reduction contexts

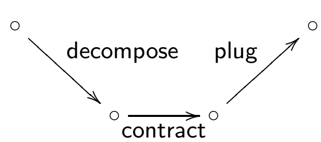
The name `reduction_context` already says it: this data type is that of the reduction contexts. Also, `apply_kt` can be identified as the traditional 'plug' function of a reduction semantics that fills a reduction context with a contractum and yields the next term in the reduction sequence. In fact, Figure 4 displays an implementation (a big-step one, by Reynolds's book and according to the diagram of Section 1) of a reduction semantics.

We refactor this implementation in Figure 5, renaming, e.g., `apply_kt` into `plug` to make that property even more manifest.

```
datatype reduction_context = C_EMPTY
                           | C_ADD1 of reduction_context * term
                           | C_ADD2 of value * reduction_context
                           | C_SEQ of reduction_context * term
                           | C_CATCH of reduction_context * term

fun plug (C_EMPTY, t')                                    (* : context * term -> term *)
    = t'
  | plug (C_ADD1 (c, t2), t1')
    = plug (c, ADD (t1', t2))
  | plug (C_ADD2 (v1, c), t2')
    = plug (c, ADD (v2t v1, t2'))
  | plug (C_SEQ (c, t2), t1')
    = plug (c, SEQ (t1', t2))
  | plug (C_CATCH (c, t2), t1')
    = plug (c, CATCH (t1', t2))

datatype value_or_decomposition = VAL of value | DEC of potential_redex * reduction_context

fun decompose' (LIT n, c)                    (* : term * context -> value_or_decomposition *)
    = decompose'_aux (c, EXPECT n)
  | decompose' (ADD (t1, t2), c)
    = decompose' (t1, C_ADD1 (c, t2))
  | decompose' (SEQ (t1, t2), c)
    = decompose' (t1, C_SEQ (c, t2))
  | decompose' (CATCH (t1, t2), c)
    = decompose' (t1, C_CATCH (c, t2))
  | decompose' (THROW, c)
    = decompose'_aux (c, EXCEPT)
  | decompose' (BLOCK t, c)
    = DEC (PRBLOCK t, c)
  | decompose' (UNBLOCK t, c)
    = DEC (PRUNBLOCK t, c)
  | decompose' (ERROR, c)
    = DEC (PRERROR, c)

and decompose'_aux (C_EMPTY, v)             (* : context * value -> value_or_decomposition *)
    = VAL v
  | decompose'_aux (C_ADD1 (c, t2), v1)
    = decompose' (t2, C_ADD2 (v1, c))
  | decompose'_aux (C_ADD2 (v1, c), v2)
    = DEC (PRSUM (v1, v2), c)
  | decompose'_aux (C_SEQ (c, t2), v1)
    = DEC (PRSEQ (v1, t2), c)
  | decompose'_aux (C_CATCH (c, t2), v1)
    = DEC (PRCATCH (v1, t2), c)

fun decompose t                                    (* : term -> value_or_decomposition *)
    = decompose' (t, C_EMPTY)

fun reduce (t, s, is)          (* : term * status * interrupts -> (term * status * interrupts) option *)
    = (case decompose t
         of VAL v       => SOME (v2t v, s, is)
          | DEC (pr, c) => (case contract (pr, s, is)
                              of NONE                => NONE
                               | SOME (t', s', is') => SOME (plug (c, t'), s', is')))
```

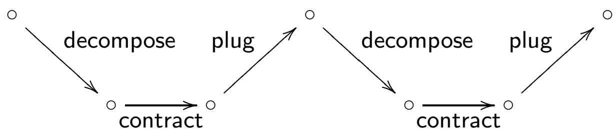
Figure 5. Refactored version of Figure 4, where `apply_kt` is lambda-dropped [28] and renamed into `plug`, an intermediate data type of values or decompositions is introduced, and `visit` and `apply_kv` are renamed into `decompose'` and `decompose'_aux` and made to yield a value or a decomposition. The result is the functional implementation of a reduction semantics [29, 30]

Visually, Figure 5 implements the following diagram:

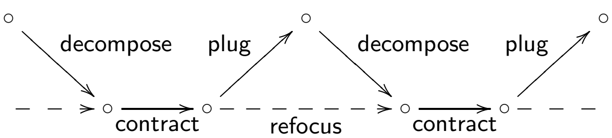


- A value is mapped into itself.
- A non-value term is decomposed into a potential redex and its reduction context. If the potential redex is an actual one, it is contracted and the contractum is plugged into the reduction context, yielding the next term in the reduction sequence. Otherwise, the term is stuck and the reduction sequence stops there.

Evaluation is then traditionally defined as iterated reduction:



As pointed out by Nielsen and the author [27], the intermediate terms in the reduction sequence can be deforested away by refocusing from the site of a redex directly to the site of the next redex in the reduction sequence:



As generously illustrated elsewhere [10, 11, 20, 27], such a refocused evaluation function implements a small-step abstract machine. As shown by Millikin and the author [25], fusing the iteration function and the move function of this yields the functional implementation of a big-step abstract machine. In the present case, this big-step abstract machine essentially coincides with Hutton and Wright's, and one is then back in known territory [3]: the functional implementation of the big-step abstract machine is in defunctionalized form and can thus be refunctionalized [24]; the refunctionalized version is in CPS and can thus be mapped back to direct style [18]; and the result is essentially a functional implementation of Hutton and Wright's natural semantics. We are then in position to answer Hutton and Wright's question about the meaning of interruptions with a variety of inter-derivable semantic artifacts (i.e., man-made-constructs).

6. A context-sensitive reduction semantics for arithmetic expressions with interruptions and errors

In the previous sections, we have derived the functional implementation of a reduction semantics out of the functional implementation of a structural operational semantics by CPS transformation and defunctionalization. Except for the usual device, in Figure 2, of not applying the current continuation when getting stuck in a contraction, we have not made much use of continuations. One could, however, and as pioneered by Felleisen in his PhD thesis [29], make the contraction function *context sensitive* by passing it the current reduction context and acting on it when raising an exception, e.g., to short-cut the syntactic propagation of the exceptions (not just of the errors) towards the root of the term.

In this section, we outline such a context-sensitive reduction semantics. In doing so, we leave the range of CPS-transformed and defunctionalized structural operational semantics; and symmetrically, on the other side of refocusing and fusion by fixed-point promotion, we distance ourselves from a big-step abstract machine in defunctionalized form.

6.1 Abstract syntax (terms and values)

6.1.1 Terms

The terms are the same as in Section 2.2.1.

6.1.2 Notion of value

In contrast to Section 2.2.2, a value is simply an integer:

```
type value = int
```

Embedding a value into a term therefore amounts to quoting it, as it were:

```
fun v2t n = LIT n
```

6.2 Notion of context-sensitive contraction

We now interpret the throw operator as a redex:

```
datatype potential_redex = PRSUM of value * value
                         | PRSEQ of value * term
                         | PRCATCH of value * term
                         | PRTHROW
                         | PRBLOCK of term
                         | PRUNBLOCK of term
                         | PRERROR
```

The simpler values of Section 6.1.2 makes for simpler individual contractions than in Section 2.3. The status may be changed, but not the current stream of signals:

- an addition maps two numbers into their sum; the status remains the same;
- sequencing from a number to a term yields this term; the status remains the same;
- a catch operator is only contracted if there has been no exception; the status remains the same;
- a throw operator unwinds the current context to emptiness or to the first catch handler:

```
fun unwind C_EMPTY
    = NONE
  | unwind (C_ADD1 (c, t2))
    = unwind c
  | unwind (C_ADD2 (v1, c))
    = unwind c
  | unwind (C_SEQ (c, t2))
    = unwind c
  | unwind (C_CATCH (c, t2))
    = SOME (t2, c)
```

  the status remains the same;
- as before, the blocking (resp. unblocking) operator yields a blocked (resp. unblocked) status, irrespective of the previous status;
- the error operator is stuck and does not modify the status.

All potential redexes but the last one are thus actual redexes and yield a contractum. Performing a contraction therefore optionally maps a potential redex, a status, a stream of signals, and a reduction context into a term, a status, a stream of signals, and a reduction context:

```
fun perform (PRSUM (n1, n2), s, is, c)
    = SOME (LIT (n1 + n2), s, is, c)
  | perform (PRSEQ (n, t), s, is, c)
    = SOME (t, s, is, c)
  | perform (PRCATCH (n, t), s, is, c)
    = SOME (LIT n, s, is, c)
  | perform (PRTHROW, s, is, c)
    = (case unwind c
         of NONE
            => SOME (THROW, s, is, C_EMPTY)
          | SOME (t', c')
            => SOME (t', s, is, c'))
  | perform (PRBLOCK t, s, is, c)
    = SOME (t, B, is, c)
  | perform (PRUNBLOCK t, s, is, c)
    = SOME (t, U, is, c)
  | perform (PRERROR, s, is, c)
    = NONE
```

Overall, the following context-sensitive contraction function[1] optionally maps a potential redex, the current status, the current stream of signals, and the current reduction context to a term, a new status, a new stream of signals, and a new reduction context, unless the potential redex is stuck:

```
fun contract (pr, B, is, c)
    = perform (pr, B, is, c)
  | contract (pr, U, is, c)
    = (case Signals.poll is       (* <- polling *)
         of (false, is') (* <- no interruption *)
            => perform (pr, U, is', c)
          | (true, is')  (* <- an interruption *)
            =>
            (case unwind c
              of NONE
                 => SOME (THROW, U, is', C_EMPTY)
               | SOME (t', c')
                 => SOME (t', U, is', c')))
```

The key new point is that if an interruption is detected, it is still treated by raising an exception but this treatment is carried out on the spot by unwinding the context in search of a catch handler or until the context is exhausted.

In fine, given the corresponding decomposition and plugging functions, we can implement the one-step reduction function as follows:

```
fun reduce (t, s, is)
    = (case decompose t
         of VAL v
            => SOME (v2t v, s, is)
          | DEC (pr, c)
            => (case contract (pr, s, is, c)
                  of NONE
                     => NONE
                   | SOME (t', s', is', c')
                     => SOME (plug (c', t'),
                              s', is')))
```

Proving the equivalence of this context-sensitive reduction semantics and of the previous context-insensitive one takes a degree of going. Specifically, one needs to relate the propagation of exceptions in the two semantics.

7. From reduction semantics to abstract machine

Based on the context-sensitive reduction semantics of Section 6, we define evaluation as iterated reduction (Section 7.1). To exemplify that this evaluation function always composes the decomposition function and the plug function, we replace this composition by a call to a dedicated 'refocus' function (Section 7.2). We then define a more efficient version of the refocus function (Section 7.3).

7.1 Reduction-based evaluation

Evaluation can yield an integer, as expected from an arithmetic expression, or an uncaught exception:

```
datatype result = EXPECT of int | EXCEPT
```

It can also become stuck.

We define evaluation by iterating over the result of decomposition. The following function optionally maps a value or a decomposition, a status, and a stream of signals to a result, a status, and a stream of signals:

```
fun iterate (VAL n, s, is)
    = SOME (EXPECT n, s, is)
  | iterate (DEC (pr, c), s, is)
    = (case contract (pr, s, is, c)
         of NONE
            => NONE
          | SOME (THROW, s', is', C_EMPTY)
            => SOME (EXCEPT, s', is')
          | SOME (t', s', is', c')
            => iterate (decompose
                          (plug (c', t')),
                        s', is'))

fun evaluate (t, s, is)
    = iterate (decompose t, s, is)
```

[1] Reminder: This contraction function is context-sensitive because it is passed the reduction context and acts on it.

7.2 Towards refocusing

We exemplify that the evaluation function of Section 7.1 always composes `decompose` and `plug` by defining a dedicated function implementing this composition:

```
fun refocus (t, c)
    = decompose (plug (c, t))
```

We then adjust the evaluation function to use `refocus`:

```
fun iterate (VAL n, s, is)
    = SOME (EXPECT n, s, is)
  | iterate (DEC (pr, c), s, is)
    = (case contract (pr, s, is, c)
         of NONE
            => NONE
          | SOME (THROW, s', is', C_EMPTY)
            => SOME (EXCEPT, s', is')
          | SOME (t', s', is', c')
            => iterate (refocus (t', c'),
                        s', is'))

fun evaluate (t, s, is)
    = iterate (refocus (t, C_EMPTY),
               s, is)
```

Morally, the initial call to `iterate`, in Section 7.1 did compose `decompose` and `plug` since plugging a term in an empty context yields this term.

7.3 Reduction-free evaluation

With the purpose of refocusing, the decomposition function is most conveniently defined as in Figure 5, i.e., as a pair of functions `decompose'` that iteratively decomposes a term and accumulates a context until it reaches a value, and `decompose'_aux` that dispatches on the context. Indeed, *optimal refocusing consists in continuing the decomposition in the current context* [27], and therefore `refocus` can be defined as `decompose'`:

```
fun refocus (t, c)
    = decompose' (t, c)
```

The result is a small-step abstract machine that alternatively refocuses and contracts. This abstract machine is reduction free because it does not construct the intermediate terms in the reduction sequence. It also naturally embodies the optimization enabled by context sensitivity and described in Section 6.2.

8. Conclusion and perspectives

Over the last years, we have observed the following facts and drawn the following lessons:

Abstract machines: Abstract machines form a natural meeting ground between theoretically minded and practically motivated language designers and developers. They are both 'practical enough' to make theoretical results flow into practice and 'theoretical enough' to direct that flow.

Abstract vs. virtual machines: Earlier on [2], we candidly pointed at the difference between abstract machines, that directly operate on terms (e.g., the CEK machine), and virtual machines, that operate on byte code resulting from compiling a term (e.g., the JVM):

$$\begin{tikzcd} \text{source term} \arrow[r, "\text{compile}"] \arrow[dr, "\text{interpret}"'] & \text{byte code} \arrow[d, "\text{run}"] \\ & \text{result} \end{tikzcd}$$

We furthermore observed that in several cases (e.g., William Burge and Peter Henderson's compiler for the SECD machine and Xavier Leroy's compiler for the Krivine machine), the byte code could be deforested and the original abstract machine could be recovered. We applied this deforestation idea to Guy Cousineau, Pierre-Louis Curien, and Michel Mauny's Categorical Abstract Machine as well as to David Schmidt's compiler for the VEC machine. In both cases, we obtained an abstract machine that, on one hand, was in the range of refocusing and transition compression, and on the other hand, was in the range of defunctionalization.

In our experience, designing the instruction set of a virtual machine is powerfully helped by (1) identifying common sequences of contractions in reduction semantics and of transitions in abstract machines, and (2) factoring combinators out of compositional evaluation functions, following Mitchell Wand's path-breaking work on combinator-based compilers in the early 1980's [61–64].

From big-step semantics to abstract machine: a functional correspondence. As initiated by Reynolds, closure conversion, CPS transformation, and defunctionalization make it possible to map a recursive program into the functional implementation of an abstract machine. (If the initial program is block-structured, just lambda-lift it [39].) This combination of transformations can also be used for deriving or relating programs [24,26,60] and is used today, e.g., for web programming [32] and for type inference [43].

From small-step semantics to abstract machine: a syntactic correspondence. Refocusing a reduction semantics and compressing transitions mechanically yield practical abstract machines [20].

Explicit substitutions: For weak-head normalization, Curien's original calculus of closures [17] gives rise to a variety of practical abstract machines with environments [10].

Computational effects: On one hand, parameterizing evaluation functions with monads, and on the other hand, making contraction functions context sensitive, and using the two correspondence mentioned just above yield the same abstract machines. This coincidence scales to classical effects [5, 9, 46] as well as to unusual ones, such as delimited continuations, properly tail-recursive stack inspection, compound monadic effects, and call by need [4,5,9,11].

Applicability: Both the functional correspondence and the syntactic correspondence apply to a host of known machines: SECD, CEK, KAM, CLS, CAM, ZINC, etc. as well as to new machines. It also scales to full normalization (as in 'normalization by evaluation'), objects [23, 38], and the stochastic π-calculus.

In general, the two correspondences provide guidelines in the jungle of semantic artifacts. As Biernacka and the author facetiously put it [11]:

> Call/cc was introduced in Scheme [15] as a Church encoding of Reynolds's escape operator [54]. A typed version of it is available in Standard ML of New Jersey [34] and Griffin has identified its logical content [33]. It is endowed with a variety of specifications: a CPS transformation [22], a CPS interpreter [35, 54], a denotational semantics [41], a computational monad [59], a big-step operational semantics [34], the CEK machine [31], calculi in the form of reduction semantics [30], and a number of implementation techniques [16, 19, 36]—not to mention its call-by-name variant in the archival version of Krivine's machine [42].
>
> Question: How do we know that all the artifacts in this semantic jungle define the same call/cc?

Our answer here: We know for sure when the representation of these semantic artifacts are inter-derivable.

Contexts: Contexts, like zipper data structures, are defunctionalized continuations: of an evaluation function for evaluation contexts, and of a one-step reduction function for reduction contexts. They are also in 1-to-1 correspondence with the compatibility rules in a calculus. In fact, the coincidence between the data types of reduction contexts and of evaluation contexts is pivotal in the correspondence between reduction orders (e.g., normal order, applicative order) and evaluation orders (e.g., call by name, call by value) that Gordon Plotkin discovered in "Call-by-name, call-by-value, and the λ-calculus" [51] and that the author and his students have materialized with the inter-derivations illustrated here.

Defining contexts and proving unique decomposition: As defunctionalized continuations, contexts can be mechanically defined out of a compositional function (e.g., one that searches for the first potential redex in a term, or an ordinary evaluation function). The unique decomposition property then holds as a corollary.

Open problem: We observe that (1) abstract machines can be obtained by CPS-transforming and defunctionalizing a compositional evaluation function, (2) reasoning about compositional evaluation functions is usually done by structural induction, possibly with an additional relation, and (3) reasoning about abstract machines is usually done using a well-founded order. Since finding such a well-founded order requires ingenuity, to which extent could one be induced by defunctionalizing a compositional evaluation function?

Parting thought: To close, we would like to underline the remarkable effectiveness of explicit substitutions (Curien, Lévy, Hardin, Abadi, Cardelli, etc.), refocusing, CPS (Strachey, Wadsworth, Reynolds, Plotkin, Steele, Friedman, Wand, Shivers, etc.), and defunctionalization (Landin, Reynolds) when considering programs as data objects.

Acknowledgments

The author is grateful to the members and chair of the ICFP'08 program committee for their invitation to present the present material. Thanks are also due to Mads Sig Ager, Małgorzata Biernacka, Dariusz Biernacki, Jan Midtgaard, Kevin Millikin, and Lasse Nielsen for the joint ride, and to Kenichi Asai, Jacob Johannsen, Johan Munk, and Ian Zerny for precious and timely comments. The running example of arithmetic expressions with interrupts and errors was developed after hours while attending AFP 2008. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Research Report BRICS RS-03-14, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[3] 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.

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

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, number 3603 in Lecture Notes in Computer Science, pages 50–65, Oxford, UK, August 2005. Springer.

[7] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag.

[8] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[9] 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.

[10] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.

[11] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.

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

[13] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[14] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[15] 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.

[16] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999. A preliminary version was presented at the 1988 ACM Conference on Lisp and Functional Programming.

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

[18] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at ESOP 1992.

[19] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[20] 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.

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

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

[23] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In Wilfrid Hodges and Ruy de Queiroz, editors, *Proceedings of the 15th Workshop on Logic, Language, Information and Computation (WoLLIC 2008)*, number 5110 in Lecture Notes in Artificial Intelligence, pages 1–16, Edinburgh, Scotland, July 2008. Springer-Verlag. Invited talk.

[24] Olivier Danvy and Kevin Millikin. Refunctionalization at work. Research Report BRICS RS-08-4, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2007. To appear in Science of Computer Programming, extended version.

[25] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

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

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

[28] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000. A preliminary version was presented at the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997).

[29] 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.

[30] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at `<http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>` and last accessed in April 2008, 1989-2001.

[31] 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.

[32] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 211–222, Coronado Island, San Diego, California, USA, November 2001. IEEE Computer Society.

[33] 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.

[34] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[35] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press.

[36] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[37] Graham Hutton and Joel Wright. What is the meaning of these constant interruptions? *Journal of Functional Programming*, 17(6):777–792, 2007.

[38] Jacob Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2008. BRICS research report RS-08-6.

[39] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[40] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, February 1987. Springer-Verlag.

[41] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[42] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[43] George Kuan and David MacQueen. Efficient type inference using ranked type variables. In Claudio Russo and Derek Dreyer, editors, *Record of the 1998 ACM SIGPLAN Workshop on ML*, pages 3–14, Freiburg, Germany, October 2007.

[44] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007.

[45] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, May 2007.

[46] Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2007. BRICS research report RS-08-3.

[47] Lasse R. Nielsen. A denotational investigation of defunctionalization. Research Report BRICS RS-00-47, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.

[48] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[49] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992.

[50] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, New York, NY, USA, January 2007. ACM Press.

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

[52] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[53] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, 2006. A preliminary version was presented at the Thirty-First Annual ACM Symposium on Principles of Programming Languages (POPL 2004).

[54] 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 [55].

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

[56] 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.

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

[58] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936-37. Corrections in Volume 43, pages 544-546, 1937.

[59] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[60] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

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

[62] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[63] Mitchell Wand. A semantic prototyping system. In Susan L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 213–221, Montréal, Canada, June 1984. ACM Press.

[64] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.

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