Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 124 (2005) 79–100
www.elsevier.com/locate/entcs

From Reduction-based to Reduction-free Normalization

Olivier Danvy[1]

BRICS[2]

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Abstract

We present a systematic construction of a reduction-free normalization function. Starting from a reduction-based normalization function, i.e., the transitive closure of a one-step reduction function, we successively subject it to refocusing (i.e., deforestation of the intermediate reduced terms), simplification (i.e., fusing auxiliary functions), refunctionalization (i.e., Church encoding), and direct-style transformation (i.e., the converse of the CPS transformation). We consider two simple examples and treat them in detail: for the first one, arithmetic expressions, we construct an evaluation function; for the second one, terms in the free monoid, we construct an accumulator-based flatten function. The resulting two functions are traditional reduction-free normalization functions.
The construction builds on previous work on refocusing and on a functional correspondence between evaluators and abstract machines. It is also reversible.

Keywords: normalization by evaluation, refocusing, defunctionalization, continuation-passing style (CPS).

1 Introduction

Normalization by evaluation is a 'reduction-free' approach to normalizing terms. Instead of repeatedly reducing a term towards its normal form, as

[1] Email: danvy@brics.dk
Home page: http://www.brics.dk/~danvy

[2] Basic Research in Computer Science (http://www.brics.dk),
funded by the Danish National Research Foundation.

doi:10.1016/j.entcs.2005.01.007

in the traditional reduction-based approach, one uses an *extensional normalization function* that does not construct any intermediate term and directly yields a normal form, if there is any [22]. Normalization by evaluation has been developed in intuitionistic type theory [14, 37, 44], proof theory [9, 10], category theory [5,16,41], λ-definability [32], partial evaluation [18,19,26], and formal semantics [1, 29, 30]. The more complicated the terms and the notions of reduction, the more complicated the normalization functions.

Normalization by evaluation therefore requires one to extensionally define a reduction-free normalization function, which is non-trivial [6, 7]. Nevertheless, it is our contention that the computational content of a reduction-based normalization function—i.e., a function intensionally defined as the transitive closure of one-step reduction—can pave the way to constructing a reduction-free normalization function:

Our starting point: We start from a reduction semantics for a language of terms [28], i.e., an abstract syntax, a notion of reduction in the form of a collection of redexes and the corresponding contraction function, and a reduction strategy. The reduction strategy takes the form of a grammar of reduction contexts, its associated plug function, and a decomposition function mapping a term to a value or to a reduction context and a redex (we assume this decomposition to be unique). Thus equipped, we define a one-step reduction function as a function whose fixed points are values, and which otherwise decomposes a non-value term into a reduction context and a redex, contracts this redex, and plugs the contractum into the context:

$$\begin{tikzcd}
\begin{array}{c}\text{non-value}\\ \text{term}\end{array} \arrow[r, "\text{decompose}"] \arrow[d, dashed, "\begin{array}{c}\text{one-step}\\ \text{reduction}\end{array}"'] & \text{context} \times \text{redex} \arrow[d, "\text{identity} \times \text{contraction}"] \\
\text{term} & \text{context} \times \text{contractum} \arrow[l, "\text{plug}"']
\end{tikzcd}$$

A reduction-based normalization function is defined as the reflexive and transitive closure of this reduction function.

Refocusing: On the way to reaching a normal form, the reduction-based normalization function repeatedly decomposes, contracts, and plugs. Observing that most of the time, the decomposition function is applied to the result of the plug function [25], Nielsen and the author have suggested to deforest the intermediate term by replacing the composition of the decomposition function and of the plug function by a *refocus* function that directly maps a reduction context and a contractum to the next reduction context and redex, if there are any. Such a refocused normalization function (i.e., a normalization function using a refocus function instead of a decomposition function and a plug function) can be viewed as an abstract machine.

The functional correspondence: An abstract machine is often a defunctionalized continuation-passing program [2, 3, 4, 13, 21]. When this is the case, such abstract machines can be refunctionalized [24] and transformed into direct style [17].

It is our experience that starting from a reduction semantics for a language of terms, we can refocus the corresponding reduction-based normalization function into an abstract machine, and refunctionalize this abstract machine into a reduction-free normalization function. We have successfully tried this construction on the lambda-calculus, both for weak-head normalization and for full normalization. The goal of this article is to illustrate it with the simple examples of arithmetic expressions and terms of the free monoid.

Overview:

In Section 2, we implement a reduction semantics for arithmetic expressions in complete detail and in Standard ML, and we define the corresponding reduction-based normalization function. In Section 3, we refocus the reduction-based normalization function of Section 2 into an abstract machine, and we present the corresponding reduction-free normalization function. In Sections 4 and 5, we go through the same motions for terms in the free monoid.

Sections 2 and 4 might appear as intimidating; however, except that they are expressed in ML, they describe straightforward reduction semantics as have been developed by Felleisen and his co-workers for the last two decades [27, 28, 45]. For this reason, these two sections have a parallel structure. Similarly, to emphasize that the construction of a reduction-free normalization function out of a reduction-based normalization function is systematic, we have also given Sections 3 and 5 a parallel structure.

Prerequisites:

The reader is expected to have some familiarity with the programming language Standard ML [39], reduction semantics [25, 28], the CPS transformation [23, 43], and defunctionalization [24, 42]. In particular, we build on the relation between continuations and evaluation contexts [20].

2 A reduction semantics for arithmetic expressions

To define a reduction semantics for simplified arithmetic expressions (integer literals and additions), we specify their abstract syntax, their notion of reduction (computing the sum of two integers), their reduction contexts and the corresponding plug function, and how to decompose them into a reduction context and the left-most inner-most redex, if there is one. We then define a one-step reduction function that decomposes a non-value term into a reduction context and a redex, contracts the redex, and plugs the contractum into the context. We can finally define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value, i.e., a normal form, is reached.

2.1 Abstract syntax

An arithmetic expression is either a literal or the addition of two terms:

```
datatype term = LIT of int
              | ADD of term * term
```

2.2 Notion of reduction

A redex is the sum of two literals, and we implement contraction as computing this sum:

```
datatype redex = SUM of int * int

(*  contract : redex -> term  *)
fun contract (SUM (n1, n2))
    = LIT (n1 + n2)
```

The left-most inner-most reduction strategy converges and yields a literal.

2.3 Reduction contexts

We seek the left-most inner-most redex in a term. The grammar of reduction contexts and the corresponding plug function are as follows:

```
datatype context = C0
                 | C1 of term * context
                 | C2 of int * context

(*  plug : context * term -> term  *)
fun plug (C0, t)
    = t
  | plug (C1 (t', c), t)
    = plug (c, ADD (t, t'))
  | plug (C2 (n, c), t)
    = plug (c, ADD (LIT n, t))
```

2.4 Decomposition

A term is a value (i.e., it does not contain any redex) or it can be decomposed into a reduction context and a redex:

```
datatype value_or_decomposition = VAL of term
                                | DEC of context * redex
```

(No term is stuck.)

The decomposition function recursively searches for the left-most innermost redex in a term. It is usually left unspecified in the literature [28]. We define it here it in a form we have found convenient in our previous study of reduction semantics [25], namely with two auxiliary functions, `decompose'` and `decompose'_aux`: `decompose'` traverses a given term and accumulates the reduction context until it finds a value, and `decompose'_aux` dispatches on the accumulated context to decide whether the given term is a value, a redex has been found, or the search must continue:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (LIT n, c)
    = decompose'_aux (c, n)
  | decompose' (ADD (t1, t2), c)
    = decompose' (t1, C1 (t2, c))
(*  decompose'_aux : context * int -> value_or_decomposition  *)
and decompose'_aux (C0, n)
    = VAL (LIT n)
  | decompose'_aux (C1 (t2, c), n)
    = decompose' (t2, C2 (n, c))
  | decompose'_aux (C2 (n', c), n)
    = DEC (c, SUM (n', n))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

Lemma 2.1 *A term* `t` *is either a value or there exists a unique context* `c` *such that* `decompose t` *evaluates to* `DEC (c, r)`*, where* `r` *a redex.*

Proof. Immediate. □

2.5 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a non-value term into a reduction context and a redex, (2) contracts the redex, and (3) plugs the contractum in the reduction context:

```
(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL t')
            => t'
          | (DEC (c, r))
            => plug (c, contract r))
```

2.6 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value (i.e., a fixed point):

```
(*  normalize : term -> term  *)
fun normalize t
    = (case reduce t
         of (LIT n)
            => LIT n
          | t'
            => normalize t')
```

In the following definition, we inline `reduce` in order to directly check whether `decompose` yields a value or a decomposition:

```
(*  iterate0 : value_or_decomposition -> term  *)
fun iterate0 (VAL t)
    = t
  | iterate0 (DEC (c, r))
    = iterate0 (decompose (plug (c, contract r)))

(*  normalize0 : term -> term  *)
fun normalize0 t
    = iterate0 (decompose t)
```

2.7 Reduction-based normalization, typefully

The type of `normalize0` is not informative. To make it appear more clearly that the normalization function yields normal forms, i.e., integers, we can refine the type of values to be that of integers, and adjust the first clause of `decompose'_aux` and the reduction function:

```
datatype value_or_decomposition = VAL of int (* was: term *)
                                | DEC of context * redex

    ...
and decompose'_aux (C0, n)
    = VAL n
  | ...

(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL n)
            => LIT n
          | (DEC (c, r))
            => plug (c, contract r))
```

The reduction-based normalization function can then return an integer rather than a literal:

```
(*  iterate1 : value_or_decomposition -> int  *)
fun iterate1 (VAL n)
    = n
  | iterate1 (DEC (c, r))
    = iterate1 (decompose (plug (c, contract r)))

(*  normalize1 : term -> int  *)
fun normalize1 t
    = iterate1 (decompose t)
```

The type of `normalize1` is more informative than that of `normalize0` since it makes it clear that applying `normalize1` to a term yields a value.

2.8 Summary and conclusion

We have implemented in ML, in complete detail, a reduction semantics for arithmetic expressions. Using this reduction semantics, we have implemented a reduction-based normalization function.

3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 2.7 into a reduction-free normalization function, i.e., one where no intermediate term is ever constructed. We first refocus the reduction-based normalization function [25] to deforest the intermediate terms, and we obtain a 'pre-abstract machine' implementing the transitive closure of the refocus function. We then simplify this pre-abstract machine into an abstract machine, i.e., a state-transition system. This abstract machine is in defunctionalized form [24], and we refunctionalize it. The result is in continuation-passing style and we re-express it in direct style [17]. The resulting direct-style function is a traditional evaluator for arithmetic expressions; in particular, it is reduction-free.

3.1 Plugging and decomposition

In the reduction-based normalization function of Section 2.7, `decompose` is always applied to the result of `plug` after the first decomposition. Let us add a vacuous initial call to `plug` so that in all cases, `decompose` is applied to the result of `plug`:

```
(*  normalize2 : term -> int  *)
fun normalize2 t
    = iterate1 (decompose (plug (C0, t)))
```

3.2 Refocusing

As investigated earlier by Nielsen and the author [25], the composition of `decompose` and `plug` can be deforested into one `refocus` function to avoid the construction of intermediate terms. In addition, this `refocus` function can be expressed very simply in terms of the decomposition functions of Section 2.4 (and this is the reason why we chose to specify them precisely like that):

```
(*  refocus : context * term -> value_or_decomposition  *)
fun refocus (c, t)
    = decompose' (t, c)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate3 : value_or_decomposition -> int  *)
fun iterate3 (VAL v)
    = v

  | iterate3 (DEC (c, r))
    = iterate3 (refocus (c, contract r))

(*  normalize3 : term -> int  *)
fun normalize3 t
    = iterate3 (refocus (C0, t))
```

The refocused normalization function is reduction-free because it is no longer based on a (one-step) reduction function. Instead, the refocus function directly maps a reduction context and a contractum to the next reduction context and redex, if there are any.

3.3 From refocused normalization function to abstract machine

The refocused normalization function is what we call a 'pre-abstract machine' [25] in the sense that `decompose'` and `decompose'_aux` form a transition function and `iterate3` is a 'trampoline' [31], i.e., another transition function that keeps activating the two others until a value is obtained. Let us fuse `iterate3` and `refocus` (i.e., `decompose'` and `decompose'_aux`, which we rename `refocus4` and `refocus4_aux` for the occasion) so that `iterate3` is directly applied to the result of `decompose'` and `decompose'_aux`. The result is a (tail-recursive) state-transition function, i.e., an abstract machine [40]:

```
(*  iterate4 : value_or_decomposition -> int  *)
fun iterate4 (VAL v)
    = v
  | iterate4 (DEC (c, r))
    = refocus4 (contract r, c)
(*  refocus4 : term * context -> int  *)
and refocus4 (LIT n, c)
    = refocus4_aux (c, n)
  | refocus4 (ADD (t1, t2), c)
    = refocus4 (t1, C1 (t2, c))
(*  refocus4_aux : context * int -> int  *)
and refocus4_aux (C0, n)
    = iterate4 (VAL n)
  | refocus4_aux (C1 (t2, c), n)
    = refocus4 (t2, C2 (n, c))
  | refocus4_aux (C2 (n', c), n)
    = iterate4 (DEC (c, SUM (n', n)))

(*  normalize4 : term -> int  *)
fun normalize4 t
    = refocus4 (t, C0)
```

The form of this machine is remarkable because `iterate4` implements the reduction rules of the reduction semantics and `refocus4` and `refocus4_aux` implement its congruence rules—a distinction that usually requires a non-trivial analysis to establish for existing abstract machines [34].

3.4 Inlining and simplification

Since `iterate4` and `contract` are only pedagogical devices, let us inline them to streamline the abstract machine. Inlining `contract`, in the last clause of `refocus4_aux`, yields the following clause:

```
    | refocus4_aux (C2 (n', c), n)
      = refocus4 (LIT (n' + n), c)
```

Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows:

```
    | refocus4_aux (C2 (n', c), n)
      = refocus4_aux (c, n' + n)
```

The resulting simplified machine is an 'eval/apply' abstract machine [36]

3.5 Refunctionalization

Like many other abstract machines [2, 3, 4, 13, 21], the abstract machine of Section 3.4 is in defunctionalized form [24]: the reduction contexts, together with `refocus4_aux`, are the first-order counterpart of a function. The higher-order counterpart of the abstract machine reads as follows:

```
  (*  refocus5 : term * (int -> int) -> int  *)
  fun refocus5 (LIT n, c)
      = c n
    | refocus5 (ADD (t1, t2), c)
      = refocus5 (t1,
                  fn n1 => refocus5 (t2,
                                     fn n2 => c (n1 + n2)))

  (*  normalize5 : term -> int  *)
  fun normalize5 t
      = refocus5 (t, fn n => n)
```

3.6 Back to direct style

The refunctionalized definition of Section 3.5 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [23, 17]. Its direct-style counterpart reads as follows:

```
  (*  refocus6 : term -> int  *)
  fun refocus6 (LIT n)
      = n
    | refocus6 (ADD (t1, t2))
      = (refocus6 t1) + (refocus6 t2)

  (*  normalize6 : term -> int  *)
  fun normalize6 t
      = refocus6 t
```

The resulting definition is that of the usual evaluation function for arithmetic expressions, i.e., a traditional reduction-free normalization function.

3.7 Summary and conclusion

We have refocused the reduction-based normalization function of Section 2 into an abstract machine, and we have exhibited the corresponding reduction-free normalization function.

4 A reduction semantics for terms in the free monoid

To define a reduction semantics for terms in the free monoid over a given carrier set, we specify their abstract syntax (a distinguished unit element, the other elements of the carrier set, and products of terms), their notion of reduction (oriented conversion rules), their reduction contexts and the corresponding plug function, and how to decompose them into a reduction context and the right-most inner-most redex, if there is one. We then define a one-step reduction function that decomposes a non-value term into a reduction context and a redex, contracts the redex, and plugs the contractum into the context. We can finally define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value, i.e., a normal form, is reached.

4.1 Abstract syntax

Given a type `elem` of carrier-set elements, a term in the free monoid is either the unit element, an element of type `elem`, or the product of two terms:

```
datatype term = UNIT
              | ELEM of elem
              | PROD of term * term
```

Terms in the free monoid obey conversion rules: the unit element is neutral for the product (both on the left and on the right), and the product is associative.

4.2 Notion of reduction

We introduce a notion of reduction by orienting the conversion rules into reduction rules:

$$\texttt{PROD (UNIT, t)} \longrightarrow \texttt{t}$$

$$\texttt{ELEM e} \longrightarrow \texttt{PROD (ELEM e, UNIT)}$$

$$\texttt{PROD (PROD (t11, t12), t2)} \longrightarrow \texttt{PROD (t11, PROD (t12, t2))}$$

We represent redexes as a data type and implement their contraction with the corresponding reduction rules:

```
datatype redex = LEFT_UNIT of term
               | RIGHTMOST of elem
```

```
                | ASSOC of (term * term) * term

(*  contract : redex -> term  *)
fun contract (LEFT_UNIT t)
    = t
  | contract (RIGHTMOST e)
    = PROD (ELEM e, UNIT)
  | contract (ASSOC ((t11, t12), t2))
    = PROD (t11, PROD (t12, t2))
```

The right-most inner-most reduction strategy converges and yields a flat, list-like term in normal form.

4.3 Reduction contexts

We seek the right-most inner-most redex in a term. The grammar of reduction contexts and the corresponding plug function are as follows:

```
datatype context = C0
                 | C1 of term * context

(*  plug : context * term -> term  *)
fun plug (C0, t)
    = t
  | plug (C1 (t1, c), t2)
    = plug (c, PROD (t1, t2))
```

4.4 Decomposition

A term is a value (i.e., it does not contain any redex) or it can be decomposed into a reduction context and a redex:

```
datatype value_or_decomposition = VAL of term
                                | DEC of context * redex
```

(No term is stuck.)

The decomposition function recursively searches for the right-most inner-most redex in a term. As in Section 2.4, we define it with two auxiliary functions, `decompose'` and `decompose'_aux`: `decompose'` traverses a given term and accumulates the reduction context until it finds a redex or a value, and `decompose'_aux` dispatches on the accumulated context to decide whether the given term is a value, a redex has been found, or the search must continue:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (UNIT, c)
    = decompose'_aux (c, UNIT)
  | decompose' (ELEM e, c)
    = DEC (c, RIGHTMOST e)
  | decompose' (PROD (t1, t2), c)
    = decompose' (t2, C1 (t1, c))
```

```
(*  decompose'_aux : context * term -> value_or_decomposition  *)
and decompose'_aux (C0, t)
    = VAL t
  | decompose'_aux (C1 (UNIT, c), t2)
    = DEC (c, LEFT_UNIT t2)
  | decompose'_aux (C1 (ELEM e, c), t2)
    = decompose'_aux (c, PROD (ELEM e, t2))
  | decompose'_aux (C1 (PROD (t11, t12), c), t2)
    = DEC (c, ASSOC ((t11, t12), t2))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

Lemma 4.1 *A term* `t` *is either a value or there exists a unique context* `c` *such that* `decompose t` *evaluates to* `DEC (c, r)`*, where* `r` *a redex.*

Proof. Immediate. □

4.5 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a non-value term into a reduction context and a redex, (2) contracts the redex, and (3) plugs the contractum in the reduction context:

```
(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL t')
            => t'
          | (DEC (c, r))
            => plug (c, contract r))
```

4.6 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value. In the following definition, and as in Section 2.6, we inline `reduce` and directly check whether `decompose` yields a value or a decomposition:

```
(*  iterate0 : value_or_decomposition -> term  *)
fun iterate0 (VAL t)
    = t
  | iterate0 (DEC (c, r))
    = iterate0 (decompose (plug (c, contract r)))

(*  normalize0 : term -> term  *)
fun normalize0 t
    = iterate0 (decompose t)
```

4.7 Reduction-based normalization, typefully

As in Section 2.7, the type of `normalize0` is not informative. To make it appear more clearly that the normalization function yields normal forms, let us introduce a data type of terms in normal form:

```
datatype term_nf = UNIT_nf
                 | PROD_nf of elem * term_nf
```

We can then refine the type of values to make it more manifest that a value is in normal form:

```
datatype value_or_decomposition = VAL of term * term_nf
                                | DEC of context * redex
```

We must then adjust `decompose'_aux` to construct values both as regular terms and as terms in normal form:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (UNIT, c)
    = decompose'_aux (c, UNIT, UNIT_nf)
  | decompose' (ELEM e, c)
    = DEC (c, RIGHTMOST e)
  | decompose' (PROD (t1, t2), c)
    = decompose' (t2, C1 (t1, c))
(*  decompose'_aux : context * term * term_nf
                     -> value_or_decomposition  *)
and decompose'_aux (C0, t, t_nf)
    = VAL (t, t_nf)
  | decompose'_aux (C1 (UNIT, c), t2, t2_nf)
    = DEC (c, LEFT_UNIT t2)
  | decompose'_aux (C1 (ELEM e, c), t2, t2_nf)
    = decompose'_aux (c, PROD (ELEM e, t2), PROD_nf (e, t2_nf))
  | decompose'_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
    = DEC (c, ASSOC ((t11, t12), t2))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

The reduction-based normalization function can then return the representation of the term in normal form:

```
(*  iterate1 : value_or_decomposition -> term_nf  *)
fun iterate1 (VAL (t, t_nf))
    = t_nf
  | iterate1 (DEC (c, r))
    = iterate1 (decompose (plug (c, contract r)))

(*  normalize1 : term -> term_nf  *)
fun normalize1 t
    = iterate1 (decompose t)
```

The type of `normalize1` is more informative than that of `normalize0` since it makes it clear that applying `normalize1` to a term yields a term in normal form.

4.8 Summary and conclusion

We have implemented in ML a reduction semantics for terms in the free monoid, given its carrier set. Using this reduction semantics, we have implemented a reduction-based normalization function.

5 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 4.7 into a reduction-free normalization function, i.e., one where no intermediate term is ever constructed. We first refocus the reduction-based normalization function and we obtain a pre-abstract machine. We then simplify this pre-abstract machine into an abstract machine. This abstract machine is in defunctionalized form, and we refunctionalize it. The result is in continuation-passing style and we re-express it in direct style. The resulting direct-style function is a traditional flatten function with an accumulator; in particular, it is reduction-free.

5.1 *Plugging and decomposition*

In the reduction-based normalization function of Section 4.7, `decompose` is always applied to the result of `plug` after the first decomposition. Let us add a vacuous initial call to `plug` so that in all cases, `decompose` is applied to the result of `plug`:

```
(*  normalize2 : term -> term_nf  *)
fun normalize2 t
    = iterate1 (decompose (plug (C0, t)))
```

5.2 *Refocusing*

As in Section 3.2, we now deforest the composition of `decompose` and `plug` into one `refocus` function:

```
(*  refocus : context * term -> value_or_decomposition  *)
fun refocus (c, t)
    = decompose' (t, c)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate3 : value_or_decomposition -> term_nf  *)
fun iterate3 (VAL (t, t_nf))
    = t_nf
  | iterate3 (DEC (c, r))
    = iterate3 (refocus (c, contract r))

(*  normalize3 : term -> term_nf  *)
fun normalize3 t
    = iterate3 (refocus (C0, t))
```

The refocused normalization function is reduction-free because it is no longer based on a reduction function and it no longer constructs intermediate terms.

5.3 From refocused evaluation function to abstract machine

Again, the refocused evaluation function is a 'pre-abstract machine' in the sense that `decompose'` and `decompose'_aux` form a transition function and `iterate3` is a 'trampoline'. Let us fuse `iterate3` and `refocus` (i.e., `decompose'` and `decompose'_aux`, which we rename `refocus4` and `refocus4_aux` as in Section 3.3), so that `iterate3` is directly applied to the result of `decompose'` and `decompose'_aux`. The result is the following abstract machine:

```
(*  iterate4 : value_or_decomposition -> term_nf  *)
fun iterate4 (VAL (t, t_nf))
    = t_nf
  | iterate4 (DEC (c, r))
    = refocus4 (contract r, c)
(*  refocus4 : term * context -> term_nf  *)
and refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT, UNIT_nf)
  | refocus4 (ELEM e, c)
    = iterate4 (DEC (c, RIGHTMOST e))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))
(*  refocus4_aux : context * term * term_nf -> term_nf  *)
and refocus4_aux (C0, t, t_nf)
    = iterate4 (VAL (t, t_nf))
  | refocus4_aux (C1 (UNIT, c), t2, t2_nf)
    = iterate4 (DEC (c, LEFT_UNIT t2))
  | refocus4_aux (C1 (ELEM e, c), t2, t2_nf)
    = refocus4_aux (c, PROD (ELEM e, t2), PROD_nf (e, t2_nf))
  | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
    = iterate4 (DEC (c, ASSOC ((t11, t12), t2)))

(*  normalize4 : term -> term_nf  *)
fun normalize4 t
    = refocus4 (t, C0)
```

5.4 Inlining and simplification

As in Section 3.4, we inline `iterate4` and `contract` to streamline the abstract machine. Three cases occur:

(i) The clause

```
| refocus4 (ELEM e, c)
  = iterate4 (DEC (c, RIGHTMOST e))
```

after inlining `iterate4` and `contract`, reads as follows:

```
| refocus4 (ELEM e, c)
  = refocus4 (PROD (ELEM e, UNIT), c)
```

Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows (skipping two steps):

```
| refocus4 (ELEM e, c)
  = refocus4_aux (c, PROD (ELEM e, UNIT), PROD_nf (e, UNIT_nf))
```

(ii) The clause

```
| refocus4_aux (C1 (UNIT, c), t2, t2_nf)
   = iterate4 (DEC (c, LEFT_UNIT t2))
```

after inlining `iterate4` and `contract`, reads as follows:

```
| refocus4_aux (C1 (UNIT, c), t2, t2_nf)
  = refocus4 (t2, c)
```

We know, however, that `t2` is in normal form, and therefore we can directly call `refocus4_aux` instead:

```
| refocus4_aux (C1 (UNIT, c), t2, t2_nf)
  = refocus4_aux (c, t2, t2_nf)
```

(iii) The clause

```
| refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
  = iterate4 (DEC (c, ASSOC ((t11, t12), t2)))
```

after inlining `iterate4` and `contract`, reads as follows:

```
| refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
  = refocus4 (PROD (t11, PROD (t12, t2)), c)
```

Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows (skipping two steps):

```
| refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
  = refocus4 (t2, C1 (t12, C1 (t11, c)))
```

We know, however, that `t2` is in normal form, and therefore we can directly call `refocus4_aux` instead:

```
| refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
  = refocus4_aux (C1 (t12, C1 (t11, c)), t2, t2_nf)
```

In the resulting definition of `refocus4_aux`, we observe that the second parameter is dead, i.e., that it is never used. Eliminating it (and renaming the last parameter to `a`) yields the following definition:

```
(*  refocus4 : term * context -> term_nf  *)
fun refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT_nf)
  | refocus4 (ELEM e, c)
    = refocus4_aux (c, PROD_nf (e, UNIT_nf))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))

(*  refocus4_aux : context * term_nf -> term_nf  *)
and refocus4_aux (C0, a)
    = a
  | refocus4_aux (C1 (UNIT, c), a)
    = refocus4_aux (c, a)
  | refocus4_aux (C1 (ELEM e, c), a)
    = refocus4_aux (c, PROD_nf (e, a))
  | refocus4_aux (C1 (PROD (t11, t12), c), a)
    = refocus4_aux (C1 (t12, C1 (t11, c)), a)
```

5.5 Refunctionalization

The above definitions of `refocus4` and `refocus4_aux` are not in defunctionalized form because of the last clause of `refocus4_aux` [24]. To put them in defunctionalized form (eureka), we need to introduce one more auxiliary function:

```
(*  refocus4 : term * context -> term_nf  *)
fun refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT_nf)
  | refocus4 (ELEM e, c)
    = refocus4_aux (c, PROD_nf (e, UNIT_nf))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))
(*  refocus4_aux : context * term_nf -> term_nf  *)
and refocus4_aux (C0, a)
    = a
  | refocus4_aux (C1 (t', c), a)
    = refocus4_aux' (t', c, a)
(*  refocus4_aux' : term * context * term_nf -> term_nf  *)
and refocus4_aux' (UNIT, c, a)
    = refocus4_aux (c, a)
  | refocus4_aux' (ELEM e, c, a)
    = refocus4_aux (c, PROD_nf (e, a))
  | refocus4_aux' (PROD (t11, t12), c, a)
    = refocus4_aux' (t12, C1 (t11, c), a)
```

Now the reduction contexts, together with `refocus4_aux`, are the first-order counterpart of a function. The higher-order counterpart of the normalization function reads as follows:

```
(*  refocus5 : term * (term_nf -> term_nf) -> term_nf  *)
fun refocus5 (UNIT, c)
    = c UNIT_nf
  | refocus5 (ELEM e, c)
    = c (PROD_nf (e, UNIT_nf))
  | refocus5 (PROD (t1, t2), c)
    = refocus5 (t2, fn t2'_nf => refocus5_aux' (t1, c, t2'_nf))

(*  refocus5_aux' : term * (term_nf -> term_nf) * term_nf -> term_nf  *)
and refocus5_aux' (UNIT, c, a)
    = c a
  | refocus5_aux' (ELEM e, c, a)
    = c (PROD_nf (e, a))
  | refocus5_aux' (PROD (t11, t12), c, a)
    = refocus5_aux' (t12, fn a' =>
        refocus5_aux' (t11, c, a'), a)

(*  normalize5 : term -> term_nf  *)
fun normalize5 t
    = refocus5 (t, fn a => a)
```

5.6 *Back to direct style*

The refunctionalized definition of Section 5.5 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls. Its direct-style counterpart reads as follows:

```
(*  refocus6 : term -> term_nf  *)
fun refocus6 UNIT
    = UNIT_nf
  | refocus6 (ELEM e)
    = PROD_nf (e, UNIT_nf)
  | refocus6 (PROD (t1, t2))
    = refocus6_aux' (t1, refocus6 t2)
(*  refocus6_aux : term * term_nf -> term_nf  *)
and refocus6_aux' (UNIT, a)
    = a
  | refocus6_aux' (ELEM e, a)
```

```
      = PROD_nf (e, a)
    | refocus6_aux' (PROD (t11, t12), a)
      = refocus6_aux' (t11, refocus6_aux' (t12, a))

  (*  normalize6 : term -> term_nf  *)
  fun normalize6 t
      = refocus6 t
```

The resulting definition is that of a flatten function with an accumulator, i.e., an uncurried version of the usual reduction-free normalization function for the free monoid [11,8,12,35].

5.7 Summary and conclusion

We have refocused the reduction-based normalization function of Section 4 into an abstract machine, and we have exhibited the corresponding reduction-free normalization function.

The resulting reduction-free normalization function could be streamlined by skipping `refocus6` as follows:

```
  (*  normalize7 : term -> term_nf  *)
  fun normalize7 t
      = refocus6_aux' (t, UNIT_nf)
```

This simplified reduction-free normalization function is the traditional flatten function with an accumulator. It, however, corresponds to another reduction-based normalization function and a slightly different reduction strategy—though one that yields the same normal forms.

6 Conclusion

There is a general consensus that normalization by evaluation is an art because one must invent a non-standard, extensional evaluation function and its left inverse [1,6,7,10,12,14,16,26,32,35,37,44].

In this article, we have built on the computational content of a reduction-based normalization function as provided by a reduction semantics, and we have presented a simple, derivational way to construct a reduction-free normalization function. We have illustrated the construction on two examples, arithmetic expressions and terms in a free monoid. Elsewhere, we have successfully constructed weak-head normalization functions for the lambda-calculus (a.k.a. evaluation functions) and normalization functions for the lambda-calculus (yielding long beta-eta-normal forms, when they exist), thereby establishing a link between normalization by evaluation and abstract machines for strong reduction [15,33,38]. We have also constructed one-pass CPS transformations, which provide an early example of normalization by evaluation.

We are currently continuing to experiment with the construction, and the extent to which it is invertible.

Acknowledgement

The author is grateful to Sergio Antoy and Yoshihito Toyama for their invitation to speak at WRS 2004 and to write the present article.

Thanks are also due to Lasse R. Nielsen for our initial joint work on defunctionalization and refocusing [24, 25], to Mads Sig Ager, Małgorzata Biernacka, Dariusz Biernacki, and Jan Midtgaard for our subsequent joint exploration of the functional correspondence between evaluators and abstract machines [2, 3, 4, 12, 13], and to Mayer Goldberg, Julia Lawall, and Kristian Støvring Sørensen for their lucid and timely comments on a preliminary version of this article.

Special thanks go to Sergio Antoy for his editorship.

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

References

[1] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14:587–611, 2004.

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

[3] 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.

[4] 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*, 2005. Accepted for publication. Extended version available as the technical report BRICS RS-04-28.

[5] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[6] Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004*, number 2998 in Lecture Notes in Computer Science, pages 260–275, Nara, Japan, April 2004. Springer-Verlag.

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.

[8] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

[10] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[11] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.

[12] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. Technical Report BRICS RS-04-29, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2004. A preliminary version was presented at the the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[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] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[15] Pierre Crégut. An abstract machine for lambda-terms normalization. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[16] Djordje Čubrić, Peter Dybjer, and Philip J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

[17] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[18] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[19] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[20] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[21] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Lecture Notes in Computer Science, Lübeck, Germany, September 2004. Springer-Verlag. To appear. Extended version available as the technical report BRICS-RS-03-33.

[22] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, University of Aarhus.

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

[24] 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.

[25] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical 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.

[26] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[27] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[28] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html, 1989-2003.

[29] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.

[30] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, number 2987 in Lecture Notes in Computer Science, pages 167–181, Barcelona, Spain, April 2002. Springer-Verlag.

[31] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 18–27, Paris, France, September 1999. ACM Press.

[32] Mayer Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[33] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, September 2002. ACM Press.

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

[35] Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.

[36] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Kathleen Fischer, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

[37] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

[38] 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.

[39] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

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

[41] John C. Reynolds. Using functor categories to generate intermediate code. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 25–36, San Francisco, California, January 1995. ACM Press.

[42] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[43] 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.

[44] René Vestergaard. The simple type theory of normalisation by evaluation. In Bernhard Gramlich and Salvador Lucas, editors, *Proceedings of the First International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, number 57 in Electronic Notes in Theoretical Computer Science, Utrecht, The Netherlands, May 2001. Elsevier Science.

[45] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.