A Walk in the Semantic Park

Olivier Danvy
Department of Computer Science
Aarhus University
Aabogade 34, 8200 Aarhus N
Denmark
danvy@cs.au.dk

Jacob Johannsen
School of Computing
University of Kent
Canterbury, Kent CT2 7NF
United Kingdom
jj80@kent.ac.uk

Ian Zerny
Department of Computer Science
Aarhus University
Aabogade 34, 8200 Aarhus N
Denmark
zerny@cs.au.dk

Abstract

To celebrate the 20th anniversary of PEPM, we are inviting you to a walk in the semantic park and to inter-derive reduction-based and reduction-free negational normalization functions.

Categories and Subject Descriptors D.1.1 [*Software*]: Programming Techniques—applicative (functional) programming; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems.

General Terms Algorithms, Languages, Theory

Keywords reduction-based normalization, reduction-free normalization, negational normal forms, De Morgan laws, reduction semantics, abstract machines, reduction contexts, evaluation contexts, continuations, continuation-passing style (CPS), CPS transformation, defunctionalization, refunctionalization, refocusing

1. Introduction

The De Morgan laws provide conversion rules between Boolean formulas, where negations float up or down an abstract syntax tree:

$$\begin{aligned}\neg(\neg t) &\leftrightarrow t\\ \neg(t_1 \wedge t_2) &\leftrightarrow (\neg t_1) \vee (\neg t_2)\\ \neg(t_1 \vee t_2) &\leftrightarrow (\neg t_1) \wedge (\neg t_2)\end{aligned}$$

where $t ::= x \mid \neg t \mid t \wedge t \mid t \vee t$. These conversion rules can be oriented into reduction rules. For example, the following reduction rules make negations float down the abstract syntax tree of a given formula:

$$\begin{aligned}\neg(\neg t) &\rightarrow t\\ \neg(t_1 \wedge t_2) &\rightarrow (\neg t_1) \vee (\neg t_2)\\ \neg(t_1 \vee t_2) &\rightarrow (\neg t_1) \wedge (\neg t_2)\end{aligned}$$

Any Boolean formula can be reduced into a negational normal form, where only variables are negated:

$$t_{nf} ::= x \mid \neg x \mid t_{nf} \wedge t_{nf} \mid t_{nf} \vee t_{nf}$$

PEPM'11, January 24–25, 2011, Austin, Texas, USA.

Negational normalization can be equivalently viewed as a *small-step process*, where the De Morgan reduction rules are repeatedly applied until a normal form is obtained, and as a *big-step process*, where a given Boolean formula is recursively traversed in one fell swoop. On the occasion it is also specified with an abstract machine, which can itself be equally viewed as a small-step process and as a big-step one [6].

The goal of this article is to inter-derive these normalization processes using the program transformations used in Reynolds's functional correspondence between evaluators and big-step abstract machines [1, 16] and in the syntactic correspondence between calculi and small-step abstract machines [2], to which we add a new prelude. In the rest of this introduction, we specify the abstract syntax of Boolean formulas and of negational normal forms. We then successively consider two reduction strategies: leftmost outermost (Section 2) and leftmost innermost (Section 3). For emphasis, the presentations of Sections 2 and 3 are deliberately parallel, so that the reader can easily identify what is generic to the methodology and what is specific to each example. Throughout, we use pure ML as a functional meta-language. We have tried to make this article self-contained, but in case of doubt, the reader should consult the first author's lecture notes at the Sixth International School on Advanced Functional Programming [3].

Terms: A Boolean formula is either a variable, a negated formula, a conjunction of two formulas, or a disjunction of two formulas. We implement Boolean formulas with the following ML data type:

```
datatype term = VAR of ide
              | NEG of term
              | CONJ of term × term
              | DISJ of term × term
```

The fold functional associated to this data type abstracts its recursive descent by parameterizing what to do in each case:

```
fun term_foldr (var, neg, conj, disj) t
    = let fun visit (VAR x)
              = var x
            | visit (NEG t)
              = neg (visit t)
            | visit (CONJ (t1, t2))
              = conj (visit t1, visit t2)
            | visit (DISJ (t1, t2))
              = disj (visit t1, visit t2)
      in visit t
      end
```

Normal forms: A normal form is a formula where only variables are negated. Since ML does not support subtyping, we implement normal forms with the following specialized data type:

```
datatype term_nf = POSVAR_nf of ide
                 | NEGVAR_nf of ide
                 | CONJ_nf of term_nf × term_nf
                 | DISJ_nf of term_nf × term_nf
```

The fold functional associated to this data type abstracts its recursive descent by parameterizing what to do in each case:

```
fun term_nf_foldr (posvar, negvar, conj, disj) t
    = let fun visit (POSVAR_nf x)
              = posvar x
            | visit (NEGVAR_nf x)
              = negvar x
            | visit (CONJ_nf (t1_nf, t2_nf))
              = conj (visit t1_nf, visit t2_nf)
            | visit (DISJ_nf (t1_nf, t2_nf))
              = disj (visit t1_nf, visit t2_nf)
      in visit t
      end
```

For example, a normal form is dualized by recursively mapping positive occurrences of variables to negative ones, negative occurrences of variables to positive ones, conjunctions to disjunctions, and disjunctions to conjunctions:

```
val dualize = term_nf_foldr (NEGVAR_nf,
                             POSVAR_nf,
                             DISJ_nf,
                             CONJ_nf)
```

A normal form is embedded into a Boolean formula by mapping every specialized constructor into the corresponding original constructor(s):

```
val embed = term_nf_foldr (VAR,
                           fn x ⇒ NEG (VAR x),
                           CONJ,
                           DISJ)
```

2. Leftmost outermost negational normalization

In this section, we go from a leftmost-outermost *reduction* strategy to the corresponding leftmost-outermost *evaluation* strategy. We first implement the reduction strategy (Section 2.1) as a prelude to implementing the corresponding reduction semantics (Section 2.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 2.3) and to the functional correspondence between abstract machines and normalization functions (Section 2.4).

2.1 Prelude to a reduction semantics

The reduction strategy induces a notion of value and of potential redex (i.e., of a term that is an actual redex or that is stuck); we are then in position to state a compositional search function that implements the reduction strategy and maps a given term either to the corresponding value, if it is in normal form, or to a potential redex (Section 2.1.1). From this search function we derive a decomposition function mapping a given term either to the corresponding value, if it is in normal form, or to a potential redex and its reduction context (Section 2.1.2). As a corollary we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 2.1.3).

2.1.1 The reduction strategy

The reduction strategy consists in locating the leftmost-outermost negation of a term which is not a variable. A value therefore is a term where only variables are negated, i.e., a normal form:

```
type value = term_nf
```

A potential redex is the negation of a term that is not a variable:

```
datatype potential_redex = PR_NEG of term
                         | PR_CONJ of term × term
                         | PR_DISJ of term × term
```

The following compositional search function implements the reduction strategy. It searches a potential redex depth-first and from left to right:

```
datatype found = VAL of value
               | POTRED of potential_redex

(*  term → found  *)
fun search_term_neg (VAR x)
    = VAL (NEGVAR_nf x)
  | search_term_neg (NEG t)
    = POTRED (PR_NEG t)
  | search_term_neg (CONJ (t1, t2))
    = POTRED (PR_CONJ (t1, t2))
  | search_term_neg (DISJ (t1, t2))
    = POTRED (PR_DISJ (t1, t2))


(*  term → found  *)
fun search_term (VAR x)
    = VAL (POSVAR_nf x)
  | search_term (NEG t)
    = search_term_neg t
  | search_term (CONJ (t1, t2))
    = (case search_term t1
         of (VAL v1)
            ⇒ (case search_term t2
                 of (VAL v2)
                    ⇒ VAL (CONJ_nf (v1, v2))
                  | (POTRED pr)
                    ⇒ POTRED pr)
          | (POTRED pr)
            ⇒ POTRED pr)
  | search_term (DISJ (t1, t2))
    = (case search_term t1
         of (VAL v1)
            ⇒ (case search_term t2
                 of (VAL v2)
                    ⇒ VAL (DISJ_nf (v1, v2))
                  | (POTRED pr)
                    ⇒ POTRED pr)
          | (POTRED pr)
            ⇒ POTRED pr)

(*  term → found  *)
fun search t
    = search_term t
```

When a negation is encountered, the auxiliary function `search_term_neg` is called to decide whether this negation is a value or a potential redex.

2.1.2 From searching to decomposing

Let us transform the search function of Section 2.1.1 into a decomposition function for the reduction semantics of Section 2.2. The only difference between searching and decomposing is that given a non-value term, searching yields a potential redex whereas decomposing yields a potential redex *and its reduction context*. This reduction context is the defunctionalized continuation of the search function, and we construct it as such, by (1) CPS-transforming the search function (and simplifying it one bit) and (2) defunctionalizing its continuation.

CPS transformation: The search function is CPS-transformed by naming its intermediate results, sequentializing their computation, and introducing an extra functional argument, the continuation, that maps an intermediate result to a final answer:

```
(*  term × (found → α) → α  *)
fun search_term_neg (VAR x, k)
    = k (VAL (NEGVAR_nf x))
  | search_term_neg (NEG t, k)
    = k (POTRED (PR_NEG t))
  | search_term_neg (CONJ (t1, t2), k)
    = k (POTRED (PR_CONJ (t1, t2)))
  | search_term_neg (DISJ (t1, t2), k)
    = k (POTRED (PR_DISJ (t1, t2)))
```

```
(*  term × (found → α) → α  *)
fun search_term (VAR x, k)
    = k (VAL (POSVAR_nf x))
  | search_term (NEG t, k)
    = search_term_neg (t, k)
  | search_term (CONJ (t1, t2), k)
    = search_term (t1,
        fn (VAL v1)
           ⇒ search_term (t2,
                fn (VAL v2)
                   ⇒ k (VAL (CONJ_nf (v1, v2)))
                 | (POTRED pr)
                   ⇒ k (POTRED pr))
         | (POTRED pr)
           ⇒ k (POTRED pr))
  | search_term (DISJ (t1, t2), k)
    = search_term (t1,
        fn (VAL v1)
           ⇒ search_term (t2,
                fn (VAL v2)
                   ⇒ k (VAL (DISJ_nf (v1, v2)))
                 | (POTRED pr)
                   ⇒ k (POTRED pr))
         | (POTRED pr)
           ⇒ k (POTRED pr))

(*  term → found  *)
fun search t
    = search_term (t, fn f ⇒ f)
```

Simplifying the CPS-transformed search: The search is completed as soon as a potential redex is found. It can thus be simplified by only applying the continuation when a value is found:

```
(*  term × (value → found) → found  *)
fun search_term_neg (VAR x, k)
    = k (NEGVAR_nf x)
  | search_term_neg (NEG t, k)
    = POTRED (PR_NEG t)
  | search_term_neg (CONJ (t1, t2), k)
    = POTRED (PR_CONJ (t1, t2))
  | search_term_neg (DISJ (t1, t2), k)
    = POTRED (PR_DISJ (t1, t2))

(*  term × (value → found) → found  *)
fun search_term (VAR x, k)
    = k (POSVAR_nf x)
  | search_term (NEG t, k)
    = search_term_neg (t, k)
  | search_term (CONJ (t1, t2), k)
    = search_term (t1, fn v1 ⇒
        search_term (t2, fn v2 ⇒
          k (CONJ_nf (v1, v2))))
  | search_term (DISJ (t1, t2), k)
    = search_term (t1, fn v1 ⇒
        search_term (t2, fn v2 ⇒
          k (DISJ_nf (v1, v2))))

(*  term → found  *)
fun search t
    = search_term (t, fn v ⇒ VAL v)
```

Potential redexes are now returned directly and the `VAL` constructor is relegated to the initial continuation.

Defunctionalization: To defunctionalize the continuation, we first enumerate the inhabitants of its function space. These inhabitants arise from the initial continuation in the definition of `search` and in the 4 intermediate continuations in the definition of `search_term`. We therefore represent the continuation as a data type with 5 constructors, together with a function `apply_cont` dispatching upon these 5 summands:

```
datatype cont = C0
              | C1 of value × cont
              | C2 of cont × term
              | C3 of value × cont
              | C4 of cont × term

(*  cont → value → found  *)
fun apply_cont C0
    = (fn v ⇒ VAL v)
  | apply_cont (C1 (v1, k))
    = (fn v2 ⇒ apply_cont k (CONJ_nf (v1, v2)))
  | apply_cont (C2 (k, t2))
    = (fn v1 ⇒ search_term (t2, C1 (v1, k)))
  | apply_cont (C3 (v1, k))
    = (fn v2 ⇒ apply_cont k (DISJ_nf (v1, v2)))
  | apply_cont (C4 (k, t2))
    = (fn v1 ⇒ search_term (t2, C3 (v1, k)))

(*  term × cont → found  *)
and search_term_neg (VAR x, k)
    = apply_cont k (NEGVAR_nf x)
  | search_term_neg (NEG t, k)
    = POTRED (PR_NEG t)
  | search_term_neg (CONJ (t1, t2), k)
    = POTRED (PR_CONJ (t1, t2))
  | search_term_neg (DISJ (t1, t2), k)
    = POTRED (PR_DISJ (t1, t2))

(*  term × cont → found  *)
and search_term (VAR x, k)
    = apply_cont k (POSVAR_nf x)
  | search_term (NEG t, k)
    = search_term_neg (t, k)
  | search_term (CONJ (t1, t2), k)
    = search_term (t1, C2 (k, t2))
  | search_term (DISJ (t1, t2), k)
    = search_term (t1, C4 (k, t2))

(*  term → found  *)
fun search t
    = search_term (t, C0)
```

This data type of defunctionalized continuations is that of reduction contexts.

We have defined `apply_cont` in curried form to emphasize that it maps each summand to a continuation. In the following, we consider its uncurried definition.

Decomposition: We are now in position to extend the search function to not only return a potential redex (if one exists) *but also its reduction context*. The result is the decomposition function of a reduction semantics, where `value_or_decomposition`, `decompose`, `decompose_term`, `decompose_term_neg`, and `decompose_cont` are the respective clones of `found`, `search`, `search_term`, `search_term_neg`, and `apply_cont`:

```
datatype value_or_decomposition
  = VAL of value
  | DEC of potential_redex × cont

(*  cont × value → value_or_decomposition  *)
fun decompose_cont (C0, v)
    = VAL v
  | decompose_cont (C1 (v1, k), v2)
    = decompose_cont (k, CONJ_nf (v1, v2))
  | decompose_cont (C2 (k, t2), v1)
    = decompose_term (t2, C1 (v1, k))
  | decompose_cont (C3 (v1, k), v2)
    = decompose_cont (k, DISJ_nf (v1, v2))
  | decompose_cont (C4 (k, t2), v1)
    = decompose_term (t2, C3 (v1, k))

(*  term × cont → value_or_decomposition  *)
and decompose_term_neg (VAR x, k)
    = decompose_cont (k, NEGVAR_nf x)
  | decompose_term_neg (NEG t, k)
    = DEC (PR_NEG t, k)
  | decompose_term_neg (CONJ (t1, t2), k)
    = DEC (PR_CONJ (t1, t2), k)
  | decompose_term_neg (DISJ (t1, t2), k)
    = DEC (PR_DISJ (t1, t2), k)

(*  term × cont → value_or_decomposition  *)
and decompose_term (VAR x, k)
    = decompose_cont (k, POSVAR_nf x)
```

```
  | decompose_term (NEG t, k)
    = decompose_term_neg (t, k)
  | decompose_term (CONJ (t1, t2), k)
    = decompose_term (t1, C2 (k, t2))
  | decompose_term (DISJ (t1, t2), k)
    = decompose_term (t1, C4 (k, t2))
(*  term → value_or_decomposition  *)
fun decompose t
    = decompose_term (t, C0)
```

2.1.3 Recomposing

A reduction context is recomposed around a term with a left fold over this context:

```
(*  cont × term → term  *)
fun recompose (C0, t)
    = t
  | recompose (C1 (v1, k), t2)
    = recompose (k, CONJ (embed v1, t2))
  | recompose (C2 (k, t2), t1)
    = recompose (k, CONJ (t1, t2))
  | recompose (C3 (v1, k), t2)
    = recompose (k, DISJ (embed v1, t2))
  | recompose (C4 (k, t2), t1)
    = recompose (k, DISJ (t1, t2))
```

2.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for negational normalization.

2.2.1 Notion of contraction

The contraction rules implement the De Morgan laws:

```
datatype contractum_or_error = CONTRACTUM of term
                             | ERROR of string

(*  potential_redex → contractum_or_error  *)
fun contract (PR_NEG t)
    = CONTRACTUM t
  | contract (PR_CONJ (t1, t2))
    = CONTRACTUM (DISJ (NEG t1, NEG t2))
  | contract (PR_DISJ (t1, t2))
    = CONTRACTUM (CONJ (NEG t1, NEG t2))
```

In the present case, all potential redexes are actual ones, i.e., no terms are stuck.

2.2.2 One-step reduction

Given a non-value term, a one-step reduction function (1) decomposes this non-value term into a potential redex and a reduction context, (2) contracts the potential redex if it is an actual one, and (3) recomposes the reduction context with the contractum. If the potential redex is not an actual one, reduction is stuck. Given a value term, reduction is also stuck:

```
datatype reduct_or_stuck = REDUCT of term
                         | STUCK of string


(*  term → reduct_or_stuck  *)
fun reduce t
    = (case decompose t
         of (VAL v)
            ⇒ STUCK "irreducible term"
          | (DEC (pr, k))
            ⇒ (case contract pr
                 of (CONTRACTUM t')
                    ⇒ REDUCT (recompose (k, t'))
                  | (ERROR s)
                    ⇒ STUCK s))
```

This one-step reduction function is the hallmark of a reduction semantics [10].

2.2.3 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value or becomes stuck. If it yields a value, this value is the result of evaluation, and if it becomes stuck, evaluation goes wrong:

```
datatype result_or_wrong = RESULT of value
                         | WRONG of string
```

The following definition uses `decompose` to distinguish between value and non-value terms:

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, k))
    = (case contract pr
         of (CONTRACTUM t')
            ⇒ iterate
                 (decompose
                    (recompose (k, t')))
          | (ERROR s)
            ⇒ WRONG s)

(*  term → result_or_wrong  *)
fun normalize t
    = iterate (decompose t)
```

2.3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 2.2.3 into a family of reduction-free normalization functions, i.e., functions that do not enumerate the reduction sequence and where no intermediate reduct is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate reducts [9], and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 2.3.1). After inlining the contraction function (Section 2.3.2), we transform this small-step abstract machine into a big-step one [6] (Section 2.3.3). This machine exhibits a number of corridor transitions, and we compress them (Section 2.3.4). We also opportunistically specialize its contexts (Section 2.3.5). The resulting abstract machine is in defunctionalized form [8], and we refunctionalize it [7] (Section 2.4.1). The result is in continuation-passing style and we re-express it in direct style [4] (Section 2.4.2). The resulting direct-style function is a traditional conversion function for Boolean formulas; in particular, it is compositional. We express it with one recursive descent using `term_foldr` (Section 2.4.3).

Modus operandi: In each of the following subsections, we derive successive versions of the normalization function, indexing its components with the number of the subsection.

2.3.1 Refocusing

The normalization function of Section 2.2.3 is reduction-based because it constructs every intermediate term in the reduction sequence. In its definition, `decompose` is always applied to the result of `recompose` after the first decomposition. In fact, a vacuous initial call to `recompose` ensures that in all cases, `decompose` is applied to the result of `recompose`:

```
fun normalize t
    = iterate (decompose (recompose (C0, t)))
```

We can factor out these composite calls in a function, `refocus0`, that maps a contractum and its reduction context to the next potential redex and the next reduction context, if such a pair exists, in the reduction sequence:

```
(*  term × cont → value_or_decomposition  *)
fun refocus0 (t, k)
    = decompose (recompose (k, t))
```

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate0 (VAL v)
    = RESULT v
  | iterate0 (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM t')
           ⇒ iterate0 (refocus0 (t', k))
         | (ERROR s)
           ⇒ WRONG s)

(*  term → result_or_wrong  *)
fun normalize0 t
    = iterate0 (refocus0 (t, C0))
```

Refocusing, extensionally: The refocus function goes from a redex site to the next redex site, if there is one.

Refocusing, intensionally: As investigated by Nielsen and the first author [9], the refocus function can be deforested to avoid constructing any intermediate reduct. Such a deforestation makes the normalization function reduction-free. The deforested version of `refocus` is optimally defined as continuing the decomposition of the contractum in the current context, i.e., as `decompose_term`:

```
(*  term × cont → value_or_decomposition  *)
fun refocus1 (t, k)
    = decompose_term (t, k)
```

The refocused evaluation function therefore reads as follows:

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM t')
           ⇒ iterate1 (refocus1 (t', k))
         | (ERROR s)
           ⇒ WRONG s)

(*  term → result_or_wrong  *)
fun normalize1 t
    = iterate1 (refocus1 (t, C0))
```

This refocused normalization function is reduction-free because it is no longer based on a (one-step) reduction function and it no longer enumerates the successive reducts in the reduction sequence.

In the rest of this section, we mechanically transform this reduction-free normalization function into an abstract machine.

2.3.2 Inlining the contraction function

We first unfold the call to `contract` in the definition of `iterate1`, and name the resulting function `iterate2`. Reasoning by inversion, there are three potential redexes and therefore the `DEC` clause in the definition of `iterate1` is replaced by three `DEC` clauses in the definition of `iterate2`:

```
(*  term × cont → value_or_decomposition  *)
fun refocus2 (t, k)
    = decompose_term (t, k)

(*  value_or_decomposition → result_or_wrong  *)
fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_NEG t, k))
    = iterate2
        (refocus2 (t, k))
  | iterate2 (DEC (PR_CONJ (t1, t2), k))
    = iterate2
        (refocus2 (DISJ (NEG t1, NEG t2), k))
  | iterate2 (DEC (PR_DISJ (t1, t2), k))
    = iterate2
        (refocus2 (CONJ (NEG t1, NEG t2), k))

(*  term → result_or_wrong  *)
fun normalize2 t
    = iterate2 (refocus2 (t, C0))
```

2.3.3 Lightweight fusion: from small-step abstract machine to big-step abstract machine

The refocused normalization function is a small-step abstract machine in the sense that `refocus2` (i.e., `decompose_term`, `decompose_term_neg` and `decompose_cont`) acts as an inner transition function and `iterate2` as an outer transition function. The outer transition function (also known as a 'driver loop' and as a 'trampoline' [11]) keeps activating the inner transition function until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [6, 12], we fuse `iterate2` and `refocus2` (i.e., `decompose_term`, `decompose_term_neg` and `decompose_cont`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_term`, `decompose_term_neg` and `decompose_cont`. The result is a big-step abstract machine [15] consisting of four (mutually tail-recursive) state-transition functions:

- `normalize3_term` is the composition of `iterate2` and `decompose_term` and a clone of `decompose_term`;
- `normalize3_term_neg` is the composition of `iterate2` and `decompose_term_neg` and a clone of `decompose_term_neg`;
- `normalize3_cont` is the composition of `iterate2` and `decompose_cont` that directly calls `iterate3` over a value or a decomposition instead of returning it to `iterate2` as `decompose_cont` did;
- `iterate3` is a clone of `iterate2` that calls the fused function `normalize3_term`.

```
(*  cont × value → result_or_wrong  *)
fun normalize3_cont (C0, v)
    = iterate3 (VAL v)
  | normalize3_cont (C1 (v1, k), v2)
    = normalize3_cont (k, CONJ_nf (v1, v2))
  | normalize3_cont (C2 (k, t2), v1)
    = normalize3_term (t2, C1 (v1, k))
  | normalize3_cont (C3 (v1, k), v2)
    = normalize3_cont (k, DISJ_nf (v1, v2))
  | normalize3_cont (C4 (k, t2), v1)
    = normalize3_term (t2, C3 (v1, k))

(*  term × cont → result_or_wrong  *)
and normalize3_term_neg (VAR x, k)
    = normalize3_cont (k, NEGVAR_nf x)
  | normalize3_term_neg (NEG t, k)
    = iterate3 (DEC (PR_NEG t, k))
  | normalize3_term_neg (CONJ (t1, t2), k)
    = iterate3 (DEC (PR_CONJ (t1, t2), k))
  | normalize3_term_neg (DISJ (t1, t2), k)
    = iterate3 (DEC (PR_DISJ (t1, t2), k))

(*  term × cont → result_or_wrong  *)
and normalize3_term (VAR x, k)
    = normalize3_cont (k, POSVAR_nf x)
  | normalize3_term (NEG t, k)
    = normalize3_term_neg (t, k)
  | normalize3_term (CONJ (t1, t2), k)
    = normalize3_term (t1, C2 (k, t2))
  | normalize3_term (DISJ (t1, t2), k)
    = normalize3_term (t1, C4 (k, t2))

(*  value_or_decomposition → result_or_wrong  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_NEG t, k))
    = normalize3_term (t, k)
  | iterate3 (DEC (PR_CONJ (t1, t2), k))
    = normalize3_term (DISJ (NEG t1, NEG t2), k)
  | iterate3 (DEC (PR_DISJ (t1, t2), k))
    = normalize3_term (CONJ (NEG t1, NEG t2), k)
```

```
(*  term → result_or_wrong  *)
fun normalize3 t
    = normalize3_term (t, C0)
```

2.3.4 Hereditary transition compression

In the abstract machine of Section 2.3.3, many of the transitions are 'corridor' ones in that they yield configurations for which there is a unique further transition. Let us hereditarily compress these transitions. To this end, we cut-and-paste the transition functions above, renaming their indices from `3` to `4`. We consider each of their clauses in turn:

Clause `normalize4_cont (C0, v)`:

```
normalize4_cont (C0, v)
= (* by inlining normalize4_cont *)
iterate4 (VAL v)
= (* by inlining iterate4 *)
RESULT v
```

Clause `normalize4_term_neg (NEG t, k)`:

```
normalize4_term_neg (NEG t, k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_NEG t, k))
= (* by inlining iterate4 *)
normalize3_term (t, k)
```

Clause `refocus4_term_neg (CONJ (t1, t2), k)`:

```
normalize4_term_neg (CONJ (t1, t2), k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_CONJ (t1, t2), k))
= (* by inlining iterate4 *)
normalize4_term (DISJ (NEG t1, NEG t2), k)
= (* by inlining normalize4_term *)
normalize4_term (NEG t1, C4 (k, NEG t2))
= (* by inlining normalize4_term *)
normalize4_term_neg (t1, C4 (k, NEG t2))
```

Clause `refocus4_term_neg (DISJ (t1, t2), k)`:

```
normalize4_term_neg (DISJ (t1, t2), k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_DISJ (t1, t2), k))
= (* by inlining iterate4 *)
normalize4_term (CONJ (NEG t1, NEG t2), k)
= (* by inlining normalize4_term *)
normalize4_term (NEG t1, C2 (k, NEG t2))
= (* by inlining normalize4_term *)
normalize4_term_neg (t1, C2 (k, NEG t2))
```

As a corollary of the compressions, the definition of `iterate3` is now unused and can be omitted. The resulting abstract machine reads as follows:

```
(*  cont × value → result_or_wrong  *)
fun normalize4_cont (C0, v)
    = RESULT v
  | normalize4_cont (C1 (v1, k), v2)
    = normalize4_cont (k, CONJ_nf (v1, v2))
  | normalize4_cont (C2 (k, t2), v1)
    = normalize4_term (t2, C1 (v1, k))
  | normalize4_cont (C3 (v1, k), v2)
    = normalize4_cont (k, DISJ_nf (v1, v2))
  | normalize4_cont (C4 (k, t2), v1)
    = normalize4_term (t2, C3 (v1, k))

(*  term × cont → result_or_wrong  *)
and normalize4_term_neg (VAR x, k)
    = normalize4_cont (k, NEGVAR_nf x)
  | normalize4_term_neg (NEG t, k)
    = normalize4_term (t, k)
  | normalize4_term_neg (CONJ (t1, t2), k)
    = normalize4_term_neg (t1, C4 (k, NEG t2))
  | normalize4_term_neg (DISJ (t1, t2), k)
    = normalize4_term_neg (t1, C2 (k, NEG t2))

(*  term × cont → result_or_wrong  *)
and normalize4_term (VAR x, k)
    = normalize4_cont (k, POSVAR_nf x)
  | normalize4_term (NEG t, k)
    = normalize4_term_neg (t, k)
  | normalize4_term (CONJ (t1, t2), k)
    = normalize4_term (t1, C2 (k, t2))
  | normalize4_term (DISJ (t1, t2), k)
    = normalize4_term (t1, C4 (k, t2))

(*  term → result_or_wrong  *)
fun normalize4 t
    = normalize4_term (t, C0)
```

2.3.5 Context specialization

To symmetrize the definitions of `refocus4_term` and `refocus4_term_neg`, we introduce two specialized contexts for `C2` and `C4`, and we specialize `normalize4_cont` to directly call `normalize5_term_neg` for the new contexts `C2NEG` and `C4NEG`:

```
datatype cont = C0
              | C1 of value × cont
              | C2 of cont × term
              | C2NEG of cont × term
              | C3 of value × cont
              | C4 of cont × term
              | C4NEG of cont × term

(*  cont × value → result_or_wrong  *)
fun normalize5_cont (C0, v)
    = RESULT v
  | normalize5_cont (C1 (v1, k), v2)
    = normalize5_cont (k, CONJ_nf (v1, v2))
  | normalize5_cont (C2 (k, t2), v1)
    = normalize5_term (t2, C1 (v1, k))
  | normalize5_cont (C2NEG (k, t2), v1)
    = normalize5_term_neg (t2, C1 (v1, k))
  | normalize5_cont (C3 (v1, k), v2)
    = normalize5_cont (k, DISJ_nf (v1, v2))
  | normalize5_cont (C4 (k, t2), v1)
    = normalize5_term (t2, C3 (v1, k))
  | normalize5_cont (C4NEG (k, t2), v1)
    = normalize5_term_neg (t2, C3 (v1, k))

(*  term × cont → result_or_wrong  *)
and normalize5_term_neg (VAR x, k)
    = normalize5_cont (k, NEGVAR_nf x)
  | normalize5_term_neg (NEG t, k)
    = normalize5_term (t, k)
  | normalize5_term_neg (CONJ (t1, t2), k)
    = normalize5_term_neg (t1, C4NEG (k, t2))
  | normalize5_term_neg (DISJ (t1, t2), k)
    = normalize5_term_neg (t1, C2NEG (k, t2))

(*  term × cont → result_or_wrong  *)
and normalize5_term (VAR x, k)
    = normalize5_cont (k, POSVAR_nf x)
  | normalize5_term (NEG t, k)
    = normalize5_term_neg (t, k)
  | normalize5_term (CONJ (t1, t2), k)
    = normalize5_term (t1, C2 (k, t2))
  | normalize5_term (DISJ (t1, t2), k)
    = normalize5_term (t1, C4 (k, t2))

(*  term → result_or_wrong  *)
fun normalize5 t
    = normalize5_term (t, C0)
```

2.4 From abstract machines to normalization functions

In this section, we transform the compressed abstract machine of Section 2.3.4 into two compositional normalization functions, one in continuation-passing style (Section 2.4.1) and one in direct style (Section 2.4.2).

2.4.1 Refunctionalization

Like many other big-step abstract machines [1, 3], the abstract machine of Section 2.3.5 is in defunctionalized form [8]: the reduction contexts, together with `normalize5_cont`, are the first-order counterpart of a function. This function is introduced with the data-type constructors `C0`, etc., and eliminated with calls to the dispatching function `normalize5_cont`. The higher-order counterpart of this abstract machine reads as follows:

```
(*  term × (value → α) → α  *)
fun normalize6_term_neg (VAR x, k)
    = k (NEGVAR_nf x)
  | normalize6_term_neg (NEG t, k)
    = normalize6_term (t, k)
  | normalize6_term_neg (CONJ (t1, t2), k)
    = normalize6_term_neg (t1, fn v1 ⇒
        normalize6_term_neg (t2, fn v2 ⇒
          k (DISJ_nf (v1, v2))))
  | normalize6_term_neg (DISJ (t1, t2), k)
    = normalize6_term_neg (t1, fn v1 ⇒
        normalize6_term_neg (t2, fn v2 ⇒
          k (CONJ_nf (v1, v2))))

(*  term × (value → α) → α  *)
and normalize6_term (VAR x, k)
    = k (POSVAR_nf x)
  | normalize6_term (NEG t, k)
    = normalize6_term_neg (t, k)
  | normalize6_term (CONJ (t1, t2), k)
    = normalize6_term (t1, fn v1 ⇒
        normalize6_term (t2, fn v2 ⇒
          k (CONJ_nf (v1, v2))))
  | normalize6_term (DISJ (t1, t2), k)
    = normalize6_term (t1, fn v1 ⇒
        normalize6_term (t2, fn v2 ⇒
          k (DISJ_nf (v1, v2))))

(*  term → result_or_wrong  *)
fun normalize6 t
    = normalize6_term (t, fn v ⇒ RESULT v)
```

This normalization function is compositional over source terms.

2.4.2 Back to direct style

The refunctionalized definition of Section 2.4.1 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [4]. Its direct-style counterpart reads as follows:

```
(*  term → value  *)
fun normalize7_term_neg (VAR x)
    = NEGVAR_nf x
  | normalize7_term_neg (NEG t)
    = normalize7_term t
  | normalize7_term_neg (CONJ (t1, t2))
    = DISJ_nf (normalize7_term_neg t1,
               normalize7_term_neg t2)
  | normalize7_term_neg (DISJ (t1, t2))
    = CONJ_nf (normalize7_term_neg t1,
               normalize7_term_neg t2)

(*  term → value  *)
and normalize7_term (VAR x)
    = POSVAR_nf x
  | normalize7_term (NEG t)
    = normalize7_term_neg t
  | normalize7_term (CONJ (t1, t2))
    = CONJ_nf (normalize7_term t1,
               normalize7_term t2)
  | normalize7_term (DISJ (t1, t2))
    = DISJ_nf (normalize7_term t1,
               normalize7_term t2)

(*  term → result_or_wrong  *)
fun normalize7 t
    = RESULT (normalize7_term t)
```

This normalization function is compositional over source terms.

2.4.3 Catamorphic normalizers

The compositional normalizer of Section 2.4.2 features two mutually recursive functions from terms to values. These two functions can be expressed as one, using the following type isomorphism:

$$(A \to B) \times (A \to B) \;\simeq\; A \to B^2$$

Representationally, this isomorphism can be exploited in two ways: by representing B^2 as $2 \to B$ and by representing B^2 as $B \times B$. Let us review each of these representations.

Representing B^2 as $2 \to B$: We first need a two-element type to account for the "polarity" of the current sub-term, i.e., whether the number of negations between the root of the given term and the current sub-term is even (in which case the polarity is positive) or it is odd (in which case the polarity is negative):

```
datatype polarity = P  (* P like Plus  *)
                  | M  (* M like Minus *)
```

We are now in position to express the normalizer with one recursive descent over the given term, threading the current polarity in an inherited fashion, and returning a term in normal form:

```
(*  term → (polarity → value)  *)
fun normalize8_term (VAR x)
    = (fn P ⇒ POSVAR_nf x
        | M ⇒ NEGVAR_nf x)
  | normalize8_term (NEG t)
    = let val c = normalize8_term t
      in fn P ⇒ c M
          | M ⇒ c P
      end
  | normalize8_term (CONJ (t1, t2))
    = let val c1 = normalize8_term t1
          val c2 = normalize8_term t2
      in fn P ⇒ CONJ_nf (c1 P, c2 P)
          | M ⇒ DISJ_nf (c1 M, c2 M)
      end
  | normalize8_term (DISJ (t1, t2))
    = let val c1 = normalize8_term t1
          val c2 = normalize8_term t2
      in fn P ⇒ DISJ_nf (c1 P, c2 P)
          | M ⇒ CONJ_nf (c1 M, c2 M)
      end

(*  term → result_or_wrong  *)
fun normalize8 t
    = RESULT (normalize8_term t P)
```

Initially, the given term has a positive polarity.

To make it manifest that this normalizer is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `term_foldr`:

```
(*  term → (polarity → value)  *)
val normalize9_term
    = term_foldr
        (fn x ⇒
          (fn P ⇒ POSVAR_nf x
            | M ⇒ NEGVAR_nf x),
         fn c ⇒
           (fn P ⇒ c M
             | M ⇒ c P),
         fn (c1, c2) ⇒
           (fn P ⇒ CONJ_nf (c1 P, c2 P)
             | M ⇒ DISJ_nf (c1 M, c2 M)),
         fn (c1, c2) ⇒
           (fn P ⇒ DISJ_nf (c1 P, c2 P)
             | M ⇒ CONJ_nf (c1 M, c2 M)))

(*  term → result_or_wrong  *)
fun normalize9 t
    = RESULT (normalize9_term t P)
```

Representing B^2 as $B \times B$: We use a pair holding a term in normal form and its dual. This pair puts us in position to express

the normalizer with one recursive descent over the given term, returning a pair of terms in normal form in a synthesized fashion:

```
(*  term → value × value *)
fun normalize10_term (VAR x)
    = (POSVAR_nf x, NEGVAR_nf x)
  | normalize10_term (NEG t)
    = let val (tp, tm) = normalize10_term t
      in (tm, tp)
      end
  | normalize10_term (CONJ (t1, t2))
    = let val (t1p, t1m) = normalize10_term t1
          val (t2p, t2m) = normalize10_term t2
      in (CONJ_nf (t1p, t2p), DISJ_nf (t1m, t2m))
      end
  | normalize10_term (DISJ (t1, t2))
    = let val (t1p, t1m) = normalize10_term t1
          val (t2p, t2m) = normalize10_term t2
      in (DISJ_nf (t1p, t2p), CONJ_nf (t1m, t2m))
      end

(*  term → result_or_wrong  *)
fun normalize10 t
    = let val (tp, tm) = normalize10_term t
      in RESULT tp
      end
```

The final result is the positive component of the resulting pair.

To make it manifest that this normalization function is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `term_foldr`:

```
(*  term → value × value  *)
val normalize11_term
    = term_foldr
        (fn x ⇒
          (POSVAR_nf x, NEGVAR_nf x),
         fn (tp, tm) ⇒
          (tm, tp),
         fn ((t1p, t1m), (t2p, t2m)) ⇒
          (CONJ_nf (t1p, t2p), DISJ_nf (t1m, t2m)),
         fn ((t1p, t1m), (t2p, t2m)) ⇒
          (DISJ_nf (t1p, t2p), CONJ_nf (t1m, t2m)))

(*  term → result_or_wrong  *)
fun normalize11 t
    = let val (tp, tm) = normalize11_term t
      in RESULT tp
      end
```

2.5 Summary and conclusion

We have refocused the reduction-based normalization function of Section 2.2.3 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions that all are inter-derivable.

3. Leftmost innermost negational normalization

In this section, we go from a leftmost-innermost *reduction* strategy to the corresponding leftmost-innermost *evaluation* strategy. We first implement the reduction strategy (Section 3.1) as a prelude to implementing the corresponding reduction semantics (Section 3.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 3.3) and to the functional correspondence between abstract machines and normalization functions (Section 3.4).

3.1 Prelude to a reduction semantics

The reduction strategy induces a notion of value and of potential redex (i.e., of a term that is an actual redex or that is stuck); we are then in position to state a compositional search function that implements the reduction strategy and maps a given term either to the corresponding value, if it is in normal form, or to a potential redex (Section 3.1.1). From this search function we derive a decomposition function mapping a given term either to the corresponding value, if it is in normal form, or to a potential redex and its reduction context (Section 3.1.2). As a corollary we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 3.1.3).

3.1.1 The reduction strategy

The reduction strategy consists in locating the leftmost-innermost negation of a term which is not a variable. A value therefore is a term where only variables are negated, i.e., a normal form:

```
type value = term_nf
```

A potential redex is the negation of a term in negational normal form which is not a variable:

```
datatype potential_redex = PR_NEG of value
                         | PR_CONJ of value × value
                         | PR_DISJ of value × value
```

The following compositional search function implements the reduction strategy: it searches a potential redex depth-first and from left to right:

```
datatype found = VAL of value
               | POTRED of potential_redex

(*  term → found  *)
fun search_term_neg t
    = (case search_term t
        of (VAL v)
           ⇒ (case v
                of (POSVAR_nf x)
                   ⇒ VAL (NEGVAR_nf x)
                 | (NEGVAR_nf x)
                   ⇒ POTRED (PR_NEG (POSVAR_nf x))
                 | (CONJ_nf (v1, v2))
                   ⇒ POTRED (PR_CONJ (v1, v2))
                 | (DISJ_nf (v1, v2))
                   ⇒ POTRED (PR_DISJ (v1, v2)))
         | (POTRED pr)
           ⇒ POTRED pr)

(*  term → found  *)
and search_term (VAR x)
    = VAL (POSVAR_nf x)
  | search_term (NEG t)
    = search_term_neg t
  | search_term (CONJ (t1, t2))
    = (case search_term t1
        of (VAL v1)
           ⇒ (case search_term t2
                of (VAL v2)
                   ⇒ VAL (CONJ_nf (v1, v2))
                 | (POTRED pr)
                   ⇒ POTRED pr)
         | (POTRED pr)
           ⇒ POTRED pr)
  | search_term (DISJ (t1, t2))
    = (case search_term t1
        of (VAL v1)
           ⇒ (case search_term t2
                of (VAL v2)
                   ⇒ VAL (DISJ_nf (v1, v2))
                 | (POTRED pr)
                   ⇒ POTRED pr)
         | (POTRED pr)
           ⇒ POTRED pr)

(*  term → found  *)
fun search t
    = search_term t
```

When a negation is encountered, the auxiliary function `search_term_neg` is called to decide whether this negation is a value or a potential redex.

3.1.2 From searching to decomposing

As in Section 2.1.2, we transform the search function of Section 3.1.1 into a decomposition function for the reduction semantics of Section 3.2. We do so by (1) CPS-transforming the search function, (2) defunctionalizing its continuation,

```
datatype cont = C0
              | C1 of value × cont
              | C2 of cont × term
              | C3 of value × cont
              | C4 of cont × term
              | C5 of cont
```

and (3) returning a potential redex (if one exists) and its reduction context:

```
datatype value_or_decomposition
  = VAL of value
  | DEC of potential_redex × cont

(*  cont × value → value_or_decomposition  *)
fun decompose_cont (C0, v)
    = VAL v
  | decompose_cont (C1 (v1, k), v2)
    = decompose_cont (k, CONJ_nf (v1, v2))
  | decompose_cont (C2 (k, t2), v1)
    = decompose_term (t2, C1 (v1, k))
  | decompose_cont (C3 (v1, k), v2)
    = decompose_cont (k, DISJ_nf (v1, v2))
  | decompose_cont (C4 (k, t2), v1)
    = decompose_term (t2, C3 (v1, k))
  | decompose_cont (C5 k, v)
    = (case v
        of (POSVAR_nf x)
           ⇒ decompose_cont (k, NEGVAR_nf x)
         | (NEGVAR_nf x)
           ⇒ DEC (PR_NEG (POSVAR_nf x), k)
         | (CONJ_nf (v1, v2))
           ⇒ DEC (PR_CONJ (v1, v2), k)
         | (DISJ_nf (v1, v2))
           ⇒ DEC (PR_DISJ (v1, v2), k))

(*  term × cont → value_or_decomposition  *)
and decompose_term_neg (t, k)
    = decompose_term (t, C5 k)

(*  term × cont → value_or_decomposition  *)
and decompose_term (VAR x, k)
    = decompose_cont (k, POSVAR_nf x)
  | decompose_term (NEG t, k)
    = decompose_term_neg (t, k)
  | decompose_term (CONJ (t1, t2), k)
    = decompose_term (t1, C2 (k, t2))
  | decompose_term (DISJ (t1, t2), k)
    = decompose_term (t1, C4 (k, t2))

(*  term → value_or_decomposition  *)
fun decompose t
    = decompose_term (t, C0)
```

3.1.3 Recomposing

Recomposing a reduction context around a term is simply done with a left fold over the reduction context:

```
(*  cont × term → term  *)
fun recompose (C0, t)
    = t
  | recompose (C1 (v1, k), t2)
    = recompose (k, CONJ (embed v1, t2))
  | recompose (C2 (k, t2), t1)
    = recompose (k, CONJ (t1, t2))
  | recompose (C3 (v1, k), t2)
    = recompose (k, DISJ (embed v1, t2))
  | recompose (C4 (k, t2), t1)
    = recompose (k, DISJ (t1, t2))
  | recompose (C5 k, t)
    = recompose (k, NEG t)
```

3.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for negational normalization.

3.2.1 Notion of contraction

The contraction rules implement the De Morgan laws:

```
datatype contractum_or_error = CONTRACTUM of term
                             | ERROR of string

(*  potential_redex → contractum_or_error  *)
fun contract (PR_NEG v)
    = CONTRACTUM (embed v)
  | contract (PR_CONJ (v1, v2))
    = CONTRACTUM (DISJ (NEG (embed v1),
                        NEG (embed v2)))
  | contract (PR_DISJ (v1, v2))
    = CONTRACTUM (CONJ (NEG (embed v1),
                        NEG (embed v2)))
```

In the present case, all potential redexes are actual ones, i.e., no terms are stuck.

3.2.2 One-step reduction

Given a non-value term, a one-step reduction function (1) decomposes this non-value term into a potential redex and a reduction context, (2) contracts the potential redex if it is an actual one, and (3) recomposes the reduction context with the contractum. If the potential redex is not an actual one, reduction is stuck. Given a value term, reduction is also stuck:

```
datatype reduct_or_stuck = REDUCT of term
                         | STUCK of string

(*  term → reduct_or_stuck  *)
fun reduce t
    = (case decompose t
         of (VAL v)
            ⇒ STUCK "irreducible term"
          | (DEC (pr, k))
            ⇒ (case contract pr
                 of (CONTRACTUM t')
                    ⇒ REDUCT (recompose (k, t'))
                  | (ERROR s)
                    ⇒ STUCK s))
```

This one-step reduction function is the hallmark of a reduction semantics [10].

3.2.3 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value or becomes stuck. If it yields a value, this value is the result of evaluation, and if it becomes stuck, evaluation goes wrong:

```
datatype result_or_wrong = RESULT of value
                         | WRONG of string
```

The following definition uses `decompose` to distinguish between value and non-value terms:

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, k))
    = (case contract pr
         of (CONTRACTUM t')
            ⇒ iterate
                 (decompose
                    (recompose (k, t')))
          | (ERROR s)
            ⇒ WRONG s)

(*  term → result_or_wrong  *)
fun normalize t
    = iterate (decompose t)
```

3.3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 3.2.3 into a family of reduction-free normalization functions, i.e., functions that do not enumerate the reduction sequence and where no intermediate reduct is ever constructed. We first refocus the reduction-based normalization function [9] to deforest the intermediate terms, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 3.3.1). After inlining the contraction function (Section 3.3.2), we transform this small-step abstract machine into a big-step one [6] (Section 3.3.3). This machine exhibits a number of corridor transitions, and we compress them (Section 3.3.4). The resulting abstract machine is in defunctionalized form [8], and we refunctionalize it [7] (Section 3.4.1). The result is in continuation-passing style and we re-express it in direct style [4] (Section 3.4.2). The resulting direct-style function is a traditional conversion function for Boolean formulas; in particular, it is compositional. We express it with one recursive descent using `term_foldr` (Section 3.4.3).

Modus operandi: In each of the following subsections, we derive successive versions of the normalization function, indexing its components with the number of the subsection.

3.3.1 Refocusing

As in Section 2.3.1, we isolate the recomposition of a reduction context with a contractum and its subsequent decomposition in one refocus function. In this refocus function, we short-cut the construction of every reduct in the reduction sequence, turning normalization from being reduction-based to being reduction-free.

3.3.2 Inlining the contraction function

As in Section 2.3.2, we inline `contract` in the definition of `iterate1`. The result is a small-step abstract machine.

3.3.3 Lightweight fusion: from small-step abstract machine to big-step abstract machine

As in Section 2.3.3, we fuse the outer and inner transition functions of the small-step abstract machine of Section 3.3.2. The result is a big-step abstract machine.

3.3.4 Hereditary transition compression

As in Section 2.3.4, many of the transitions of the abstract machine of Section 2.3.3 are 'corridor' ones. We compress them hereditarily, and we also exploit the property that decomposing the term representation of a value in a context is the same as continuing the decomposition of this value in this context.

3.3.5 Context specialization

As in Section 2.3.5, for symmetry, we introduce two specialized contexts for `C2` and `C4`, and we specialize `normalize4_cont` to directly call `normalize5_cont_neg` for the new contexts `C2NEG` and `C4NEG`:

```
datatype cont = C0
              | C1 of value × cont
              | C2 of cont × term
              | C2NEG of cont × value
              | C3 of value × cont
              | C4 of cont × term
              | C4NEG of cont × value
              | C5 of cont

(*  cont × value → result_or_wrong  *)
fun normalize5_cont (C0, v)
    = RESULT v
  | normalize5_cont (C1 (v1, k), v2)
    = normalize5_cont (k, CONJ_nf (v1, v2))
  | normalize5_cont (C2 (k, t2), v1)
    = normalize5_term (t2, C1 (v1, k))
  | normalize5_cont (C2NEG (k, v2), v1)
    = normalize5_cont_neg (C1 (v1, k), v2)
  | normalize5_cont (C3 (v1, k), v2)
    = normalize5_cont (k, DISJ_nf (v1, v2))
  | normalize5_cont (C4 (k, t2), v1)
    = normalize5_term (t2, C3 (v1, k))
  | normalize5_cont (C4NEG (k, v2), v1)
    = normalize5_cont_neg (C3 (v1, k), v2)
  | normalize5_cont (C5 k, v)
    = normalize5_cont_neg (k, v)

(*  cont × value → result_or_wrong  *)
and normalize5_cont_neg (k, POSVAR_nf x)
    = normalize5_cont (k, NEGVAR_nf x)
  | normalize5_cont_neg (k, NEGVAR_nf x)
    = normalize5_cont (k, POSVAR_nf x)
  | normalize5_cont_neg (k, CONJ_nf (v1, v2))
    = normalize5_cont_neg (C4NEG (k, v2), v1)
  | normalize5_cont_neg (k, DISJ_nf (v1, v2))
    = normalize5_cont_neg (C2NEG (k, v2), v1)

(*  term × cont → result_or_wrong  *)
and normalize5_term (VAR x, k)
    = normalize5_cont (k, POSVAR_nf x)
  | normalize5_term (NEG t, k)
    = normalize5_term (t, C5 k)
  | normalize5_term (CONJ (t1, t2), k)
    = normalize5_term (t1, C2 (k, t2))
  | normalize5_term (DISJ (t1, t2), k)
    = normalize5_term (t1, C4 (k, t2))

(*  term → result_or_wrong  *)
fun normalize5 t
    = normalize5_term (t, C0)
```

3.4 From abstract machines to normalization functions

In this section, we transform the compressed abstract machine of Section 3.3.4 into two compositional normalization functions, one in continuation-passing style (Section 3.4.1) and one in direct style (Section 3.4.2).

3.4.1 Refunctionalization

Like many other big-step abstract machines [1, 3], the abstract machine of Section 3.3.4 is in defunctionalized form [8]: the reduction contexts, together with `normalize4_cont`, are the first-order counterpart of a function. We refunctionalize this big-step abstract machine into a higher-order normalization function. As in Section 2.4.1, this normalization function is compositional over source terms.

3.4.2 Back to direct style

The refunctionalized definition of Section 3.4.1 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [4]. Its direct-style counterpart reads as follows:

```
(*  value → value  *)
fun normalize7_cont_neg (POSVAR_nf x)
    = NEGVAR_nf x
  | normalize7_cont_neg (NEGVAR_nf x)
    = POSVAR_nf x
  | normalize7_cont_neg (CONJ_nf (v1, v2))
    = DISJ_nf (normalize7_cont_neg v1,
               normalize7_cont_neg v2)
  | normalize7_cont_neg (DISJ_nf (v1, v2))
    = CONJ_nf (normalize7_cont_neg v1,
               normalize7_cont_neg v2)

(*  term → value  *)
and normalize7_term (VAR x)
    = POSVAR_nf x
  | normalize7_term (NEG t)
    = normalize7_cont_neg (normalize7_term t)
```

```
  | normalize7_term (CONJ (t1, t2))
    = CONJ_nf (normalize7_term t1,
               normalize7_term t2)
  | normalize7_term (DISJ (t1, t2))
    = DISJ_nf (normalize7_term t1,
               normalize7_term t2)

(*  term → result_or_wrong  *)
fun normalize7 t
    = RESULT (normalize7_term t)
```

This normalization function is compositional over source terms, and uses an auxiliary function which is compositional over normal forms.

3.4.3 Catamorphic normalizer

To make it manifest that the normalizer of Section 3.4.2 is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `term_foldr`. By the same token, since the auxiliary function is also compositional and singly recursive, we also express it as an instance of `term_nf_foldr`:

```
(*  value → value  *)
val normalize8_cont_neg = term_nf_foldr (NEGVAR_nf,
                                         POSVAR_nf,
                                         DISJ_nf,
                                         CONJ_nf)

(*  term → value  *)
val normalize8_term = term_foldr (POSVAR_nf,
                                  normalize8_cont_neg,
                                     CONJ_nf,
                                     DISJ_nf)

(*  term → result_or_wrong  *)
fun normalize8 t
    = RESULT (normalize8_term t)
```

NB: In effect, `normalize8_cont_neg` dualizes normal forms. This dualization captures the essence of the innermost normalization strategy.

3.5 Summary and conclusion

We have refocused the reduction-based normalization function of Section 3.2.3 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions that all are inter-derivable.

4. Conclusion and perspectives

The inter-derivations illustrated here witness a striking unity of computation, be this for reduction semantics, abstract machines, and normalization function: they all truly define the same elephant. The structural coincidence between reduction contexts and evaluation contexts as defunctionalized continuations, in particular, plays a key rôle to connect reduction strategies and evaluation strategies, a connection that was first established by Plotkin [14]. As for Ohori and Sasano's lightweight fusion [12], it provides the linchpin between the functional representations of small-step and big-step operational semantics [6]. Overall, the inter-derivations illustrate the conceptual value of semantics-based program manipulation, as promoted over the past two decades in PEPM.

Acknowledgments: We are grateful to Jeremy Siek and Siau-Cheng Khoo for the invitation to present this work at the 20th anniversary of PEPM. The example of negational normalization originates in a joint work of the first and second authors [5]. The two preludes to a reduction semantics originate in a joint work of the first and third authors, who benefited from Kenichi Asai's gracious hospitality at Ochanomizu University to complete this article.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003.

[2] M. Biernacka and O. 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.

[3] O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.

[4] O. Danvy. Back to direct style. *Science of Computer Programming*, 22 (3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[5] O. Danvy and J. Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76:302–323, 2010.

[6] O. Danvy and K. 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.

[7] O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04.

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

[9] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, Nov. 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.

[10] M. 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, Aug. 1987.

[11] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In P. Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, Sept. 1999.

[12] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, Jan. 2007.

[13] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

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

[15] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark, Sept. 1981. Reprinted in the Journal of Logic and Algebraic Programming 60-61:17-139, 2004, with a foreword [13].

[16] J. 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 [17].

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