From Reduction-based to Reduction-free Normalization

Olivier Danvy
Department of Computer Science,
Aarhus University*

May 2009

Abstract

We document an operational method to construct reduction-free normalization functions. Starting from a reduction-based normalization function from a reduction semantics, i.e., the iteration of a one-step reduction function, we successively subject it to refocusing (i.e., deforestation of the intermediate successive terms in the reduction sequence), equational simplification, refunctionalization (i.e., the converse of defunctionalization), and direct-style transformation (i.e., the converse of the CPS transformation), ending with a reduction-free normalization function of the kind usually crafted by hand. We treat in detail four simple examples: calculating arithmetic expressions, recognizing Dyck words, normalizing lambda-terms with explicit substitutions and call/cc, and flattening binary trees.

The overall method builds on previous work by the author and his students on a syntactic correspondence between reduction semantics and abstract machines and on a functional correspondence between evaluators and abstract machines. The measure of success of these two correspondences is that each of the inter-derived semantic artifacts (i.e., man-made constructs) could plausibly have been written by hand, as is the actual case for several ones derived here.

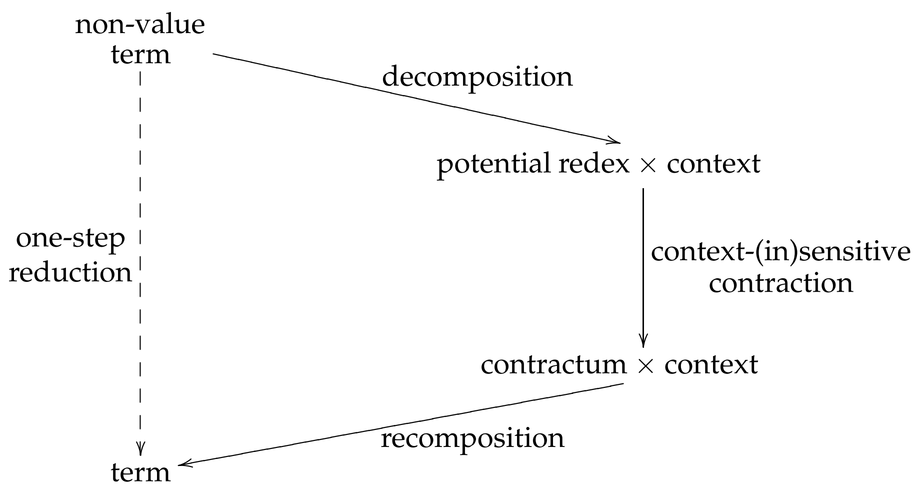
*Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `<danvy@cs.au.dk>`

Contents

1 Introduction

Grosso modo, there are two ways to specify the semantics of a programming language, given a specification of its syntax: one uses small steps and is based on a notion of reduction, and the other uses big steps and is based on a notion of evaluation. Plotkin, 30 years ago [64], has connected the two, most notably by showing how two standard reduction orders (namely normal order and applicative order) respectively correspond to two equally standard evaluation orders (namely call by name and call by value). In these lecture notes, we continue Plotkin's program and illustrate how the computational content of a reduction-based normalization function—i.e., a function intensionally defined as the iteration of a one-step reduction function—can pave the way to intensionally constructing a reduction-free normalization function—i.e., a big-step evaluation function:

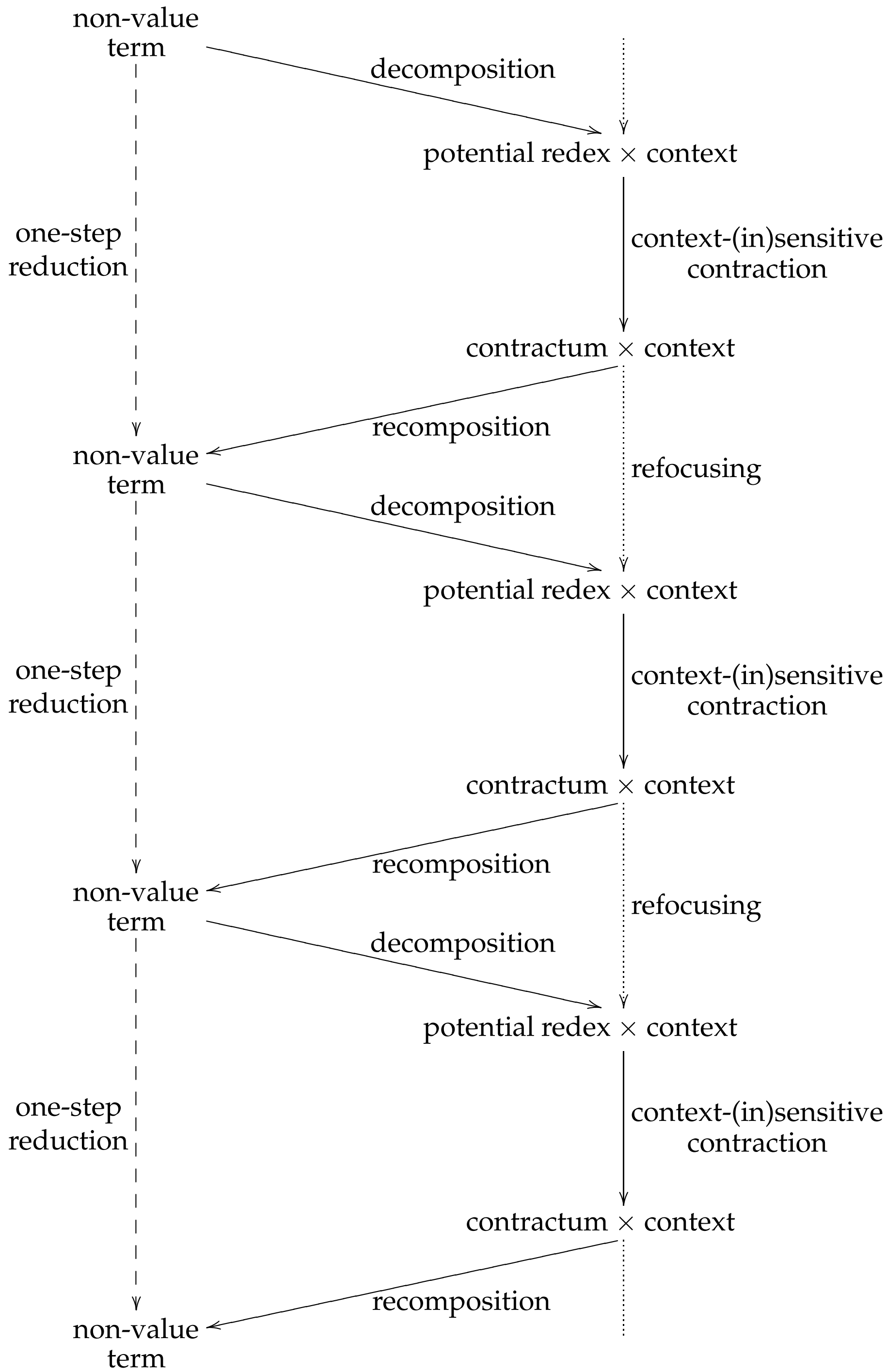
Our starting point: We start from a reduction semantics for a language of terms [40], i.e., an abstract syntax (terms and values), a notion of reduction in the form of a collection of potential redexes and the corresponding contraction function, and a reduction strategy. The reduction strategy takes the form of a grammar of reduction contexts (terms with a hole), its associated recompose function (filling the hole of a given context with a given term), and a decomposition function mapping a term to a value or to a potential redex and a reduction context. Under the assumption that this decomposition is unique, we define a one-step reduction function as a partial function whose fixed points are values and which otherwise decomposes a non-value term into a reduction context and a potential redex, contracts this potential redex if it is an actual one (otherwise the non-value term is stuck), and recomposes the context with the contractum:



The contraction function is context-insensitive if it maps an actual redex to a contractum regardless of its reduction context. Otherwise, it is context-sensitive and maps an actual redex and its reduction context to a contractum and a reduction context (possibly another one).

A *reduction-based* normalization function is defined as the iteration of this one-step reduction function along the reduction sequence.

A syntactic correspondence: On the way towards a normal form, the reduction-based normalization function repeatedly decomposes, contracts, and recomposes. Observing that most of the time, the decomposition function is applied to the result of the recomposition function [38], Nielsen and the author have suggested to deforest the intermediate term by replacing the composition of the decomposition function and of the recomposition function by a *refocus* function that directly maps a contractum and a reduction context to the next potential redex and reduction context, if there are any in the reduction sequence. Such a refocused normalization function (i.e., a normalization function using a refocus function instead of a decomposition function and a recomposition function) takes the form of a small-step abstract machine. This abstract machine is *reduction-free* because it does not construct any of the intermediate terms in the reduction sequence on the way towards a normal form:


non-value term
decomposition
potential redex × context
one-step reduction
context-(in)sensitive contraction
contractum × context
recomposition
non-value term
refocusing
decomposition
potential redex × context
one-step reduction
context-(in)sensitive contraction
contractum × context
recomposition
non-value term
refocusing
decomposition
potential redex × context
one-step reduction
context-(in)sensitive contraction
contractum × context
recomposition
non-value term

A functional correspondence: A big-step abstract machine is often a defunctionalized continuation-passing program [3, 4, 5, 16, 23]. When this is the case, such abstract machines can be refunctionalized [35, 37] and transformed into direct style [20, 32].

It is our consistent 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 of the kind usually crafted by hand. The goal of these lecture notes is to illustrate this method with four simple examples: arithmetic expressions, Dyck words, applicative-order lambda-terms with explicit substitutions, first without and then with call/cc, and binary trees.

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 a small-step abstract machine, and we present the corresponding compositional reduction-free normalization function. In Sections 4 and 5, we go through the same motions for recognizing Dyck words. In Section 6 and 7, we repeat the construction for lambda-terms applied to integers, and in Section 8 and 9 for lambda-terms applied to integers and call/cc. Finally, in Sections 10 to 13, we turn to flattening binary trees. In Sections 10 and 11, we proceed outside in, whereas in Sections 12 and 13, we proceed inside out. Admittedly at the price of repetitiveness, each of these pairs of sections (i.e., 2 and 3, 4 and 5, etc.) can be read independently. All the other ones have the same structure and narrative and they can thus be given a quicker read.

Structure: Sections 2, 4, 6, 8, 10, and 12 might seem intimidating, but they should not: they describe, in ML, straightforward reduction semantics as have been developed by Felleisen and his co-workers for the last two decades [39, 40, 73]. For this reason, these sections both have a parallel structure and as similar a narrative as seemed sensible:

1. Abstract syntax
2. Notion of contraction
3. Reduction strategy
4. One-step reduction
5. Reduction-based normalization

6. Summary
7. Exercises

Similarly, to emphasize that the construction of a reduction-free normalization function out of a reduction-based normalization function is systematic, Sections 3, 5, 7, 9, 11, and 13 have also been given a parallel structure and a similar narrative:

1. Decomposition and recomposition
2. Refocusing: from reduction-based to reduction-free normalization
3. Inlining the contraction function
4. Lightweight fusion: from small-step to big-step abstract machine
5. Compressing corridor transitions
6. Renaming transition functions and flattening configurations
7. Refunctionalization
8. Back to direct style
9. Closure unconversion
10. Summary
11. Exercises

We kindly invite the reader to play along and follow this derivational structure, at least for a start.

Prerequisites: We expect the reader to have a very basic familiarity with the programming language Standard ML [59] and to have read John Reynolds's "Definitional Interpreters" [67] at least once (otherwise the reader should start by reading the appendices of the present lecture notes, page 87 and onwards). For the rest, the lecture notes are self-contained.

Concepts: The readers receptive to suggestions will be entertained with the following concepts: reduction semantics [38, 40], including decomposition and its left inverse, recomposition; small-step and big-step abstract machines [65]; lightweight fusion [33, 36, 63] and its left inverse, lightweight fission; defunctionalization [37, 67] and its left inverse, refunctionalization [35]; the CPS transformation [30, 70] and its left inverse, the direct-style transformation [20, 32];

and closure conversion [53] and its left inverse, closure unconversion. In particular, we regularly build on evaluation contexts being the defunctionalized continuations of an evaluation function [22, 26]. To make these lecture notes self-contained, we have spelled out closure conversion, CPS transformation, defunctionalization, lightweight fission, and lightweight fusion in appendix.

Contribution: These lecture notes build on work that was carried out at Aarhus University over the last decade and that gave rise to a number of doctoral theses [2, 10, 15, 24, 57, 58, 62] and MSc theses [48, 61]. The examples of arithmetic expressions and of binary trees were presented at WRS'04 [21]. The example of lambda-terms originates in a joint work with Lasse R. Nielsen [38], Małgorzata Biernacka [12, 13], and Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard [3, 5]. The term 'lightweight fission' was suggested by Chung-chieh Shan.[1]

Online material: The entire ML code of these lecture notes is available from the home page of the author, at http://www.cs.au.dk/danvy/AFP08/, along with a comprehensive glossary.

2 A reduction semantics for calculating arithmetic expressions

The goal of this section is to define a one-step reduction function for arithmetic expressions and to construct the corresponding reduction-based evaluation function.

To define a reduction semantics for simplified arithmetic expressions (integer literals, additions, and subtractions), we specify their abstract syntax (Section 2.1), their notion of contraction (Section 2.2), and their reduction strategy (Section 2.3). We then define a one-step reduction function that decomposes a non-value term into a potential redex and a reduction context, contracts the potential redex, if it is an actual one, and recomposes the context with the contractum (Section 2.4). 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 (Section 2.5).

2.1 Abstract syntax: terms and values

Terms: An arithmetic expression is either a literal or an operation over two terms. In this section, we only consider two operators: addition and subtraction.

[1] Personal communication to the author, 30 October 2008, Aarhus, Denmark.

```
datatype operator = ADD | SUB

datatype term = LIT of int | OPR of term * operator * term
```

Values: Values are terms without operations. We specify them with a separate data type, along with an embedding function from values to terms:

```
datatype value = INT of int

fun embed_value_in_term (INT n)
    = LIT n
```

2.2 Notion of contraction

A potential redex is an operation over two values:

```
datatype potential_redex = PR_OPR of value * operator * value
```

A potential redex may be an actual one and trigger a contraction, or it may be stuck. Correspondingly, the following data type accounts for a successful or failed contraction:

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

The string accounts for an error message.

We are now in position to define a contraction function:

```
(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_OPR (INT n1, ADD, INT n2))
    = CONTRACTUM (LIT (n1 + n2))
  | contract (PR_OPR (INT n1, SUB, INT n2))
    = CONTRACTUM (LIT (n1 - n2))
```

In the present case, no terms are stuck. Stuck terms would arise if operators were extended to include division, since an integer cannot be divided by 0. (See Exercise 6 in Section 2.7.)

2.3 Reduction strategy

We seek the left-most inner-most potential redex in a term.

Reduction contexts: The grammar of reduction contexts reads as follows:

```
datatype context = CTX_MT
                 | CTX_LEFT of context * operator * term
                 | CTX_RIGHT of value * operator * context
```

Operationally, a context is a term with a hole, represented inside-out in a zipper-like fashion [47]. (And "MT" is read aloud as "empty.")

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

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * context
```

The decomposition function recursively searches for the left-most innermost redex in a term. It is usually left unspecified in the literature [40]. We define it here in a form that time and again we have found convenient [26], namely as a big-step abstract machine with two state-transition functions, `decompose_term` and `decompose_context` between two states: a term and a context, and a context and a value.

- `decompose_term` traverses a given term and accumulates the reduction context until it finds a value;
- `decompose_context` dispatches over the accumulated context to determine whether the given term is a value, the search must continue, or a potential redex has been found.

```
(*  decompose_term : term * context -> value_or_decomposition  *)
fun decompose_term (LIT n, C)
    = decompose_context (C, INT n)
  | decompose_term (OPR (t1, r, t2), C)
    = decompose_term (t1, CTX_LEFT (C, r, t2))

(*  decompose_context : context * value -> value_or_decomposition  *)
and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_LEFT (C, r, t2), v1)
    = decompose_term (t2, CTX_RIGHT (v1, r, C))
  | decompose_context (CTX_RIGHT (v1, r, C), v2)
    = DEC (PR_OPR (v1, r, v2), C)

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose_term (t, CTX_MT)
```

Recomposition: The recomposition function peels off context layers and constructs the resulting term, iteratively:

```
(*  recompose : context * term -> term  *)
fun recompose (CTX_MT, t)
    = t
  | recompose (CTX_LEFT (C, r, t2), t1)
    = recompose (C, OPR (t1, r, t2))
  | recompose (CTX_RIGHT (v1, r, C), t2)
    = recompose (C, OPR (embed_value_in_term v1, r, t2))
```

Lemma 1 *A term* `t` *is either a value or there exists a unique context* `C` *such that* `decompose t` *evaluates to* `DEC (pr, C)`*, where* `pr` *is a potential redex.*

Proof 1 *Straightforward, considering that* `context` *and* `decompose_context` *are a defunctionalized representation. The refunctionalized counterpart of* `decompose` *et al. reads as follows:*

```
(*  decompose'_term : term * context * (value -> value_or_decomposition)
                      -> value_or_decomposition  *)
fun decompose'_term (LIT n, C, k)
    = k (INT n)
  | decompose'_term (OPR (t1, r, t2), C, k)
    = decompose'_term (t1, CTX_LEFT (C, r, t2), fn v1 =>
        decompose'_term (t2, CTX_RIGHT (v1, r, C), fn v2 =>
          DEC (PR_OPR (v1, r, v2), C)))

(*  decompose' : term -> value_or_decomposition  *)
fun decompose' t
    = decompose'_term (t, CTX_MT, fn v => VAL v)
```

Since `decompose'` *(and its auxiliary function* `decompose'_term`*) is well typed, it yields a value or a decomposition. Since* `decompose'_term` *is compositional in its first argument (the term to decompose) and affine in its third (its continuation), it terminates; and since it deterministically traverses its first argument depth first and from left to right, its result is unique.* □

2.4 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) decomposes a 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. The following data type accounts for whether the contraction is successful or the non-value term is stuck:

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

```
(*  reduce : term -> reduct  *)
fun reduce t
    = (case decompose t
         of (VAL v)
            => REDUCT (embed_value_in_term v)
          | (DEC (pr, C))
            => (case contract pr
                  of (CONTRACTUM t')
                     => REDUCT (recompose (C, t'))
                   | (ERROR s)
                     => STUCK s))
```

2.5 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), if any. The following data type accounts for whether evaluation yields a value or goes wrong:

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

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

```
(*  iterate0 : value_or_decomposition -> result  *)
fun iterate0 (VAL v)
    = RESULT v
  | iterate0 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate0 (decompose (recompose (C, t')))
          | (ERROR s)
            => WRONG s)

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

2.6 Summary

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

2.7 Exercises

Exercise 1 *Define a function* `embed_potential_redex_in_term` *that maps a potential redex into a term.*

Exercise 2 *Show that, for any term* `t`*, if evaluating* `decompose t` *yields* `DEC (pr, C)`*, then evaluating* `recompose (C, embed_potential_redex_in_term pr)` *yields* `t`*. (Hint: Reason by structural induction over* `t`*, using inversion at each step.)*

Exercise 3 *Write a handful of test terms and specify the expected outcome of their normalization.*

Exercise 4 *Implement the reduction semantics above in the programming language of your choice (e.g., Haskell or Scheme), and run the tests of Exercise 3.*

Exercise 5 *Write an unparser from terms to the concrete syntax of your choice, and instrument the normalization function of Section 2.5 so that (one way or another) it displays the successive terms in the reduction sequence.*

Exercise 6 *Extend the source language with multiplication and division, and adjust your implementation, including the unparser of Exercise 5:*

```
datatype operator = ADD | SUB | MUL | DIV

(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_OPR (INT n1, ADD, INT n2))
    = CONTRACTUM (LIT (n1 + n2))
  | contract (PR_OPR (INT n1, SUB, INT n2))
    = CONTRACTUM (LIT (n1 - n2))
  | contract (PR_OPR (INT n1, MUL, INT n2))
    = CONTRACTUM (LIT (n1 * n2))
  | contract (PR_OPR (INT n1, DIV, INT 0))
    = ERROR "division by 0"
  | contract (PR_OPR (INT n1, DIV, INT n2))
    = CONTRACTUM (LIT (n1 div n2))
```

In addition to the two changes just above (i.e., the definitions of `operator` *and of* `contract`*), what else needs to be adjusted in your extended implementation?*

Exercise 7 *Write test terms that use multiplications and divisions and specify the expected outcome of their evaluation, and run these tests on your extended implementation.*

Exercise 8 *As a follow-up to Exercise 5, visualize the reduction sequence of a stuck term.*

Exercise 9 *Write a function mapping a natural number* n *to a term that normalizes into* `RESULT (INT n)` *in* n *steps. (In other words, the reduction sequence of this term should have length* n*.)*

Exercise 10 *Write a function mapping a natural number* n *to a term that normalizes into* `RESULT (INT n)` *in* $2 \times n$ *steps.*

Exercise 11 *Write a function mapping an even natural number* n *to a term that normalizes into* `RESULT (INT n)` *in* $n/2$ *steps.*

Exercise 12 *Write a function mapping a natural number* n *to a term that normalizes into* `RESULT (INT n!)` *(i.e., the factorial of* n*) in* 0 *steps.*

Exercise 13 *Write a function mapping a natural number* n *to a term whose normalization becomes stuck after* 2^n *steps.*

Exercise 14 *Extend the data types* `reduct` *and* `result` *with not just an error message but also the problematic potential redex:*

```
datatype reduct = REDUCT of term
                | STUCK of string * term

datatype result = RESULT of value
                | WRONG of string * term
```

(Hint: The function `embed_potential_redex_in_term` *from Exercise 1 will come handy.) Adapt your implementation to this new data type, and test it.*

Exercise 15 *Write the direct-style counterpart of* `decompose'` *and* `decompose'_term` *in the proof of Lemma 1, using* `callcc` *and* `throw` *as found in the* `SMLofNJ.Cont` *library.*

Exercise 16 *The following function allegedly distributes multiplications and divisions over additions and subtractions:*

```
(*  distribute : term -> term  *)
fun distribute t
    = let fun visit (LIT n, k)
              = k (LIT n)
            | visit (OPR (t1, ADD, t2), k)
              = OPR (visit (t1, k), ADD, visit (t2, k))
            | visit (OPR (t1, SUB, t2), k)
              = OPR (visit (t1, k), SUB, visit (t2, k))
            | visit (OPR (t1, MUL, t2), k)
              = visit (t1, fn t1' =>
                  visit (t2, fn t2' =>
                    k (OPR (t1', MUL, t2'))))
            | visit (OPR (t1, DIV, t2), k)
              = visit (t1, fn t1' =>
                  visit (t2, fn t2' =>
                    k (OPR (t1', DIV, t2'))))
      in visit (t, fn t' => t')
      end
```

1. *Verify this allegation on a couple of examples.*

2. *Write a new data type (or more precisely: two) accounting for additions and subtractions of multiplications and divisions, and retarget* `distribute` *so that it constructs elements of your data type. Run your code on the same couple of examples as just above.*

3. *What is the type of* `visit` *now? (To answer this question, you might want to lambda-lift the definition of* `visit` *outside your definition of* `distribute` *so that the two definitions coexist in the same scope, and let ML infer their type.)*

Exercise 17 *It is tempting to see the second parameter of* `visit`*, in Exercise 16, as a continuation. However, the definition of* `visit` *is not in continuation-passing style since in the second and third clause, the calls to* `visit` *are not in tail position. (Technically, the second parameter of* `visit` *is a 'delimited' continuation [29].)*

1. *CPS-transform your definition of* `visit`*, keeping* `distribute` *in direct style for simplicity. For comparison, CPS-transforming the original definition of* `visit` *would yield something like the following template:*

```
(*  distribute' : term -> term  *)
fun distribute' t
    = let fun visit (..., k, mk)
                = ...
      in visit (t, fn (t', mk) => mk t', fn t' => t')
      end
```

   The result is now in CPS: all calls are tail calls, right up to the initial (meta-) continuation.

2. *Defunctionalize the second and third parameters of* `visit` *(i.e., the delimited continuation* `k` *and the meta-continuation* `mk`*). You now have a big-step abstract machine: an iterative state-transition system where each clause specifies a transition.*

3. *Along the lines of Appendix F, write the corresponding small-step abstract machine.*

3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 2.5 into a family of reduction-free normalization functions, i.e., ones where no intermediate term is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate terms, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 3.1). After inlining the contraction function (Section 3.2), we transform this small-step abstract machine into a big-step one (Section 3.3). This machine exhibits a number of corridor transitions, and we compress them (Section 3.4). We then flatten its configurations and rename its transition functions into something more intuitive (Section 3.5). The resulting abstract machine is in defunctionalized form, and we refunctionalize it (Section 3.6). The result is in continuation-passing style and we re-express it in direct style (Section 3.7). The resulting direct-style function is a traditional evaluator for arithmetic expressions; in particular, it is compositional and reduction-free.

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. In practice, the reader should run the tests of Exercise 3 in Section 2.7 at each step of the derivation, for sanity value.

3.1 Refocusing: from reduction-based to reduction-free normalization

The normalization function of Section 2.5 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`:

```
(*  normalize0' : term -> result  *)
fun normalize0' t
    = iterate0 (decompose (recompose (CTX_MT, t)))
```

Refocusing, extensionally: As investigated earlier by Nielsen and the author [38], the composition of `decompose` and `recompose` can be deforested into a '`refocus`' function to avoid constructing the intermediate terms in the reduction sequence. Such a deforestation makes the normalization function reduction-free.

Refocusing, intensionally: It turns out that the `refocus` function can be expressed very simply in terms of the decomposition functions of Section 2.3 (and this is the reason why we chose to specify them precisely like that):

```
(*  refocus : term * context -> value_or_decomposition  *)
fun refocus (t, C)
    = decompose_term (t, C)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate1 : value_or_decomposition -> result  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate1 (refocus (t', C))
          | (ERROR s)
            => WRONG s)

(*  normalize1 : term -> result  *)
fun normalize1 t
    = iterate1 (refocus (t, CTX_MT))
```

This 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 contractum and a reduction context to the next redex and reduction context, if there are any in the reduction sequence.

3.2 Inlining the contraction function

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

```
(*  iterate2 : value_or_decomposition -> result  *)
fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_OPR (INT n1, ADD, INT n2), C))
    = iterate2 (refocus (LIT (n1 + n2), C))
  | iterate2 (DEC (PR_OPR (INT n1, SUB, INT n2), C))
    = iterate2 (refocus (LIT (n1 - n2), C))

(*  normalize2 : term -> result  *)
fun normalize2 t
    = iterate2 (refocus (t, CTX_MT))
```

We are now ready to fuse the composition of `iterate2` with `refocus` (shaded just above).

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

The refocused normalization function is small-step abstract machine in the sense that `refocus` (i.e., `decompose_term` and `decompose_context`) acts as a transition function and `iterate1` as a 'trampoline' [43], i.e., a 'driver loop' or again another transition function that keeps activating `refocus` until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [33, 36, 63], we fuse `iterate2` and `refocus` (i.e., `decompose_term` and `decompose_context`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_term` and `decompose_context`. The result is a big-step abstract machine [65] consisting of three (mutually tail-recursive) state-transition functions:

- `refocus3_term` is the composition of `iterate2` and `decompose_term` and a clone of `decompose_term`;
- `refocus3_context` is the composition of `iterate2` and `decompose_context` that directly calls `iterate3` over a value or a decomposition instead of returning it to `iterate2` as `decompose_context` did;
- `iterate3` is a clone of `iterate2` that calls the fused function `refocus3_term`.

```
(*  refocus3_term : term * context -> result  *)
fun refocus3_term (LIT n, C)
    = refocus3_context (C, INT n)
  | refocus3_term (OPR (t1, r, t2), C)
    = refocus3_term (t1, CTX_LEFT (C, r, t2))

(*  refocus3_context : context * value -> result  *)
and refocus3_context (CTX_MT, v)
    = iterate3 (VAL v)
  | refocus3_context (CTX_LEFT (C, r, t2), v1)
    = refocus3_term (t2, CTX_RIGHT (v1, r, C))
  | refocus3_context (CTX_RIGHT (v1, r, C), v2)
    = iterate3 (DEC (PR_OPR (v1, r, v2), C))

(*  iterate3 : value_or_decomposition -> result  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_OPR (INT n1, ADD, INT n2), C))
    = refocus3_term (LIT (n1 + n2), C)
  | iterate3 (DEC (PR_OPR (INT n1, SUB, INT n2), C))
    = refocus3_term (LIT (n1 - n2), C)

(*  normalize3 : term -> result  *)
fun normalize3 t
    = refocus3_term (t, CTX_MT)
```

In this abstract machine, `iterate3` implements the contraction rules of the reduction semantics separately from its congruence rules, which are implemented by `refocus3_term` and `refocus3_context`. This staged structure is remarkable because obtaining this separation for pre-existing abstract machines is known to require non-trivial analyses [44].

3.4 Compressing corridor transitions

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

Clause `refocus4_context (CTX_MT, v)`:

```
refocus4_context (CTX_MT, v)
= (* by unfolding the call to refocus4_context *)
iterate4 (VAL v)
= (* by unfolding the call to iterate4 *)
RESULT v
```

Clause `iterate4 (DEC (PR_OPR (INT n1, ADD, INT n2), C))`:

```
iterate4 (DEC (PR_OPR (INT n1, ADD, INT n2), C))
= (* by unfolding the call to iterate4 *)
refocus4_term (LIT (n1 + n2), C)
= (* by unfolding the call to refocus4_term *)
refocus4_context (C, INT (n1 + n2))
```

Clause `iterate4 (DEC (PR_OPR (INT n1, SUB, INT n2), C))`:

```
iterate4 (DEC (PR_OPR (INT n1, SUB, INT n2), C))
= (* by unfolding the call to iterate4 *)
refocus4_term (LIT (n1 - n2), C)
= (* by unfolding the call to refocus4_term *)
refocus4_context (C, INT (n1 - n2))
```

There are two corollaries to the compressions above:

Dead clauses: The clause "`iterate4 (VAL v)`" is dead, and therefore can be implemented as raising a "`DEAD_CLAUSE`" exception.

Invariants: All live transitions to `iterate4` are now over `DEC (PR_OPR (v1, r, v2), C)`, for some `v1`, `r`, `v2`, and `C`.

3.5 Renaming transition functions and flattening configurations

The resulting simplified machine is a familiar 'eval/apply/continue' abstract machine [54]. We therefore rename `refocus4_term` to `eval5`, `refocus4_context` to `continue5`, and `iterate4` to `apply5`. We also flatten the configuration `iterate4 (DEC (PR_OPR (v1, r, v2), C))` into `apply5 (v1, r, v2, C)`. The result reads as follows:

```
(*  eval5 : term * context -> result  *)
fun eval5 (LIT n, C)
    = continue5 (C, INT n)
  | eval5 (OPR (t1, r, t2), C)
    = eval5 (t1, CTX_LEFT (C, r, t2))

(*  continue5 : context * value -> result  *)
and continue5 (CTX_MT, v)
    = RESULT v
  | continue5 (CTX_LEFT (C, r, t2), v1)
    = eval5 (t2, CTX_RIGHT (v1, r, C))
  | continue5 (CTX_RIGHT (v1, r, C), v2)
    = apply5 (v1, r, v2, C)

(*  apply5 : value * operator * value * context -> result  *)
and apply5 (INT n1, ADD, INT n2, C)
    = continue5 (C, INT (n1 + n2))
  | apply5 (INT n1, SUB, INT n2, C)
    = continue5 (C, INT (n1 - n2))

(*  normalize5 : term -> result  *)
fun normalize5 t
    = eval5 (t, CTX_MT)
```

3.6 Refunctionalization

Like many other abstract machines [3, 4, 5, 16, 23], the abstract machine of Section 3.5 is in defunctionalized form [37]: the reduction contexts, together with `continue5`, are the first-order counterpart of a function. The higher-order counterpart of this abstract machine reads as follows:

```
(*  eval6 : term * (value -> 'a) -> 'a  *)
fun eval6 (LIT n, k)
    = k (INT n)
  | eval6 (OPR (t1, r, t2), k)
    = eval6 (t1, fn v1 =>
        eval6 (t2, fn v2 =>
          apply6 (v1, r, v2, k)))
```

```
(*  apply6 : value * operator * value * (value -> 'a) -> 'a  *)
and apply6 (INT n1, ADD, INT n2, k)
    = k (INT (n1 + n2))
  | apply6 (INT n1, SUB, INT n2, k)
    = k (INT (n1 - n2))

(*  normalize6 : term -> result  *)
fun normalize6 t
    = eval6 (t, fn v => RESULT v)
```

The resulting refunctionalized program is a familiar eval/apply evaluation function in CPS.

3.7 Back to direct style

The refunctionalized definition of Section 3.6 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [30, 20]. Its direct-style counterpart reads as follows:

```
(*  eval7 : term -> value  *)
fun eval7 (LIT n)
    = INT n
  | eval7 (OPR (t1, r, t2))
    = apply7 (eval7 t1, r, eval7 t2)

(*  apply7 : value * operator * value -> value  *)
and apply7 (INT n1, ADD, INT n2)
    = INT (n1 + n2)
  | apply7 (INT n1, SUB, INT n2)
    = INT (n1 - n2)

(*  normalize7 : term -> result  *)
fun normalize7 t
    = RESULT (eval7 t)
```

The resulting program is a traditional eval/apply evaluation function in direct style, à la McCarthy, i.e., a reduction-free normalization function of the kind usually crafted by hand.

3.8 Closure unconversion

This section is intentionally left blank, since the expressible values in the interpreter of Section 3.7 are first-order.

3.9 Summary

We have refocused the reduction-based normalization function of Section 2 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions. Most of the members of this family are ML implementations of independently known semantic artifacts: abstract machines, big-step operational semantics, and denotational semantics.

3.10 Exercises

Exercise 18 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 4 in Section 2.7. At each step of the derivation, run the tests of Exercise 3 in Section 2.7.*

Exercise 19 *Up to and including the normalization function of Section 3.5, it is simple to visualize the successive terms in the reduction sequence, namely by instrumenting* `iterate1`, `iterate2`, `iterate3`, `iterate4`, *and* `apply5`. *Do you agree? What about from Section 3.6 and onwards?*

Exercise 20 *Would it make sense, in the definition of* `normalize6`, *to take* `fn v => v` *as the initial continuation? If so, what would be the definition of* `normalize7` *and what would be its type?*

Exercise 21 *Refocus the reduction-based normalization function of Exercise 6 in Section 2.7 and move on until the eval/apply evaluation function in CPS. From then on, to write it in direct style, the simplest is to use a dynamically scoped exception handled at the top level:*

```
exception WRONG of string

(*  eval7 : term -> value  *)
fun eval7 (LIT n)
    = INT n
  | eval7 (OPR (t1, r, t2))
    = apply7 (eval7 t1, r, eval7 t2)

(*  apply7 : value * value -> value *)
and apply7 (INT n1, ADD, INT n2)
    = INT (n1 + n2)
  | apply7 (INT n1, SUB, INT n2)
    = INT (n1 - n2)
  | apply7 (INT n1, MUL, INT n2)
    = INT (n1 * n2)
  | apply7 (INT n1, DIV, INT 0)
    = raise (WRONG "division by 0")
```

```
  | apply7 (INT n1, DIV, INT n2)
    = INT (n1 div n2)

(*  normalize7 : term -> result  *)
fun normalize7 t
    = RESULT (eval7 t)
      handle (WRONG s) => STUCK s
```

In a pinch, of course, a lexically scoped first-class continuation (using `callcc` *and* `throw` *as found in the* `SMLofNJ.Cont` *library) would do as well:*

```
(*  normalize7' : term -> result  *)
fun normalize7' t
    = callcc (fn top =>
        let (*  eval7 : term -> value  *)
            fun eval7 (LIT n)
                = INT n
              | eval7 (OPR (t1, r, t2))
                = apply7 (eval7 t1, r, eval7 t2)
            (*  apply7 : value * value -> value *)
            and apply7 (INT n1, ADD, INT n2)
                = INT (n1 + n2)
              | apply7 (INT n1, SUB, INT n2)
                = INT (n1 - n2)
              | apply7 (INT n1, MUL, INT n2)
                = INT (n1 * n2)
              | apply7 (INT n1, DIV, INT 0)
                = throw top (STUCK "division by 0")
              | apply7 (INT n1, DIV, INT n2)
                = INT (n1 div n2)
        in RESULT (eval7 t)
        end)
```

4 A reduction semantics for recognizing Dyck words

The goal of this section is to define a one-step reduction function towards recognizing well-parenthesized words, i.e., Dyck words, and to construct the corresponding reduction-based recognition function.

To define a reduction semantics for recognizing Dyck words, we first specify the abstract syntax of parenthesized words (Section 4.1), the associated notion of contraction (Section 4.2), and the reduction strategy (Section 4.3). We then define a one-step reduction function that decomposes a non-empty word into a redex and a reduction context, contracts the redex, and recomposes the context with the contractum if the contraction has succeeded (Section 4.4). We can finally define a reduction-based recognition function that repeatedly applies the one-step reduction function until an empty word is reached, if each contraction has succeeded (Section 4.5).

4.1 Abstract syntax: terms and values

Pre-terms: We start from a string of characters and parse it into a word, i.e., an ML list of parentheses:

```
datatype parenthesis = L of int | R of int

type word = parenthesis list

(*  smurf : string -> word option  *)
fun smurf s
    = let fun loop (~1, ps)
                = SOME ps
            | loop (i, ps)
                = (case String.sub (s, i)
                     of #"("
                        => loop (i - 1, (L 0) :: ps)
                      | #"["
                        => loop (i - 1, (L 1) :: ps)
                      | #"{"
                        => loop (i - 1, (L 2) :: ps)
                      | #"}"
                        => loop (i - 1, (R 2) :: ps)
                      | #"]"
                        => loop (i - 1, (R 1) :: ps)
                      | #")"
                        => loop (i - 1, (R 0) :: ps)
                      | _
                        => NONE)
      in loop ((String.size s) - 1, nil)
      end
```

Terms: A term is a word.

Values: A value is an empty word, i.e., an empty list of parentheses.

4.2 Notion of contraction

Our notion of contraction consists in removing matching pairs of parentheses in a context. As usual, we represent redexes as a data type and implement their contraction with a function:

```
datatype potential_redex = PR_MATCH of int * int

type contractum_or_error = bool

(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_MATCH (l, r))
    = l = r
```

4.3 Reduction strategy

We seek the left-most pair of matching parentheses in a word.

Reduction contexts: The grammar of reduction contexts reads as follows:

```
type  left_context = int list
type right_context = word

type context = left_context * right_context
```

Decomposition: A term is a value (i.e., it does not contain any potential redex, i.e., here, it is the empty word), it can be decomposed into a potential redex and a reduction context, or it is neither:

```
datatype value_or_decomposition = VAL
                                | DEC of potential_redex * context
                                | NEITHER of string
```

The decomposition function iteratively searches for the left-most potential redex in a word. As in Section 2.3, we define it as a big-step abstract machine with auxiliary functions, `decompose_word`, `decompose_word_paren`, and `decompose_context` between three states: a left and a right context; a left context, a left parenthesis, and a right context; and a left context and an optional right parenthesis and right context.

- `decompose_word` dispatches on the right context and defers to `decompose_word_paren`, and `decompose_context`;
- `decompose_word_paren` dispatches on the current parenthesis, and defers to `decompose_word` or `decompose_context`;
- `decompose_context` determines whether a value has been found, a potential redex has been found, or neither.

```
(*  decompose_word : left_context * right_context
                     -> value_or_decomposition  *)
fun decompose_word (ls, nil)
    = decompose_context (ls, NONE)
  | decompose_word (ls, p :: ps)
    = decompose_word_paren (ls, p, ps)

(*  decompose_word_paren : left_context * parenthesis * right_context
                           -> value_or_decomposition  *)
and decompose_word_paren (ls, L l, ps)
    = decompose_word (l :: ls, ps)
  | decompose_word_paren (ls, R r, ps)
    = decompose_context (ls, SOME (r, ps))

(*  decompose_context : left_context * (parenthesis * right_context) option
                        -> value_or_decomposition  *)
and decompose_context (nil, NONE)
    = VAL
  | decompose_context (nil, SOME (r, ps))
    = NEITHER "unmatched right parenthesis"
  | decompose_context (l :: ls, NONE)
    = NEITHER "unmatched left parenthesis"
  | decompose_context (l :: ls, SOME (r, ps))
    = DEC (PR_MATCH (l, r), (ls, ps))

(*  decompose : word -> value_or_decomposition  *)
fun decompose w
    = decompose_word (nil, w)
```

Recomposition: The recomposition function peels off the layers of the left context and constructs the resulting term, iteratively:

```
(*  recompose_word : context -> word  *)
fun recompose_word (nil, ps)
    = ps
  | recompose_word (l :: ls, ps)
    = recompose_word (ls, (L l) :: ps)

(*  recompose : context * unit -> word  *)
fun recompose ((ls, ps), ())
    = recompose_word (ls, ps)
```

Lemma 2 *A word* `w` *is either a value, or there exists a unique context* `C` *such that* `decompose w` *evaluates to* `DEC (pr, C)`*, where* `pr` *is a potential redex, or it is stuck.*

Proof 2 *Straightforward (see Exercise 25 in Section 4.7).*

4.4 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a non-value, non-stuck 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. The following data type accounts for whether the contraction is successful or the non-value term is stuck:

```
datatype reduct = REDUCT of word
                | STUCK

(*  reduce : word -> reduct  *)
fun reduce w
    = (case decompose w
         of VAL
            => REDUCT nil
          | (DEC (pr, C))
            => if contract pr
               then REDUCT (recompose (C, ()))
               else STUCK
          | (NEITHER s)
            => STUCK)
```

4.5 Reduction-based recognition

A reduction-based recognition function is one that iterates the one-step reduction function until it yields a value or finds a mismatch. In the following definition, and as in Section 2.5, we use `decompose` to distinguish between value terms, decomposable terms, and stuck terms:

```
(*  iterate0 : value_or_decomposition -> bool  *)
fun iterate0 VAL
    = true
  | iterate0 (DEC (pr, C))
    = if contract pr
      then iterate0 (decompose (recompose (C, ())))
      else false
  | iterate0 (NEITHER s)
    = false

(*  normalize0 : word -> bool  *)
fun normalize0 w
    = iterate0 (decompose w)
```

The correctness and termination of this definition is simple to establish: each iteration removes the left-most pair of matching parentheses, and the procedure stops if no parentheses are left or if no left-most pair of parentheses exists or if they do not match.

4.6 Summary

We have implemented a reduction semantics for recognizing well-parenthesized words, in complete detail. Using this reduction semantics, we have presented a reduction-based recognition function.

4.7 Exercises

Exercise 22 *Write a handful of test words and specify the expected outcome of their recognition.*

Exercise 23 *Implement the reduction semantics above in the programming language of your choice, and run the tests of Exercise 22.*

Exercise 24 *Instrument the implementation of Exercise 23 to visualize a reduction sequence.*

Exercise 25 *In the proof of Lemma 2, do as in the proof of Lemma 1 and write the refunctionalized counterpart of* `decompose` *et al.*

Exercise 26 *Let us modify the notion of contraction to match as many left and right parentheses as possible:*

```
(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_MATCH (l, r), C)
    = let fun visit (l :: ls, (R r) :: ps)
              = if r = l
                then visit (ls, ps)
                else NONE
            | visit (ls, ps)
              = SOME (ls, ps)
      in if l = r
         then visit C
         else NONE
      end
```

Use the result of Exercise 24 to visualize a reduction sequence with such a generalized contraction.

5 From reduction-based to reduction-free recognition

In this section, we transform the reduction-based recognition function of Section 4.5 into a family of reduction-free recognition functions, i.e., one where no intermediate word is ever constructed. We first refocus the reduction-based recognition function to deforest the intermediate words, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 5.1). After inlining the contraction function (Section 5.2), we transform this small-step abstract machine into a big-step one (Section 5.3). This abstract machine exhibits a number of corridor transitions, and we compress them (Section 5.4). We then flatten its configurations and rename its transition functions into something more intuitive (Section 5.5). The resulting abstract machine is in defunctionalized form, and we refunctionalize it (Section 5.6). The result is in continuation-passing style and we re-express it in direct style (Section 5.7). The resulting direct-style function is compositional and reduction-free.

Modus operandi: In each of the following subsections, and as in Section 3, we derive successive versions of the recognition function, indexing its components with the number of the subsection. In practice, the reader should run the tests of Exercise 22 in Section 4.7 at each step of the derivation, for sanity value.

5.1 Refocusing: from reduction-based to reduction-free recognition

The recognition function of Section 4.5 is reduction-based because it constructs every intermediate word 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`:

```
(*  normalize0' : word -> bool  *)
fun normalize0' w
    = iterate0 (decompose (recompose ((nil, w), ())))
```

Refocusing, extensionally: The composition of `decompose` and `recompose` can be deforested into a '`refocus`' function to avoid constructing the intermediate words in the reduction sequence. Such a deforestation makes the recognition function reduction-free.

Refocusing, intensionally: As in Section 3.1, the `refocus` function can be expressed very simply in terms of the decomposition functions of Section 4.3:

```
(*  refocus : context * unit -> value_or_decomposition  *)
fun refocus ((ls, ps), ())
    = decompose_word (ls, ps)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate1 : value_or_decomposition -> bool  *)
fun iterate1 VAL
    = true
  | iterate1 (DEC (pr, C))
    = if contract pr
      then iterate1 (refocus (C, ()))
      else false
  | iterate1 (NEITHER s)
    = false

(*  normalize1 : word -> bool  *)
fun normalize1 w
    = iterate1 (refocus ((nil, w), ()))
```

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

5.2 Inlining the contraction function

We first inline the call to `contract` in the definition of `iterate1`, and name the resulting function `iterate2`:

```
(*  iterate2 : value_or_decomposition -> bool  *)
fun iterate2 VAL
    = true
  | iterate2 (DEC (PR_MATCH (l, r), C))
    = if l = r
      then iterate2 (refocus (C, ()))
      else false
  | iterate2 (NEITHER s)
    = false

(*  normalize2 : word -> bool  *)
fun normalize2 w
    = iterate2 (refocus ((nil, w), ()))
```

We are now ready to fuse the composition of `iterate2` with `refocus` (shaded just above).

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

The refocused recognition function is a small-step abstract machine in the sense that `refocus` (i.e., `decompose_word`, `decompose_word_paren`, and `decompose_context`) acts as a transition function and `iterate1` as a driver loop that keeps activating `refocus` until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [33, 36, 63], we fuse `iterate2` and `refocus` (i.e., `decompose_word`, `decompose_word_paren`, and `decompose_context`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_word`, `decompose_word_paren`, and `decompose_context`. The result is a big-step abstract machine [65] consisting of four (mutually tail-recursive) state-transition functions:

- `refocus3_word` is the composition of `iterate2` and `decompose_word` and a clone of `decompose_word`;
- `refocus3_word_paren` is the composition of `iterate2` and `decompose_word_paren` and a clone of `decompose_word_paren`;
- `refocus3_context` is the composition of `iterate2` and `decompose_context` that directly calls `iterate3` instead of returning to `iterate2` as `decompose_context` did;
- `iterate3` is a clone of `iterate2` that calls the fused function `refocus3_word`.

```
(*  refocus3_word : left_context * right_context -> bool  *)
fun refocus3_word (ls, nil)
    = refocus3_context (ls, NONE)
  | refocus3_word (ls, p :: ps)
    = refocus3_word_paren (ls, p, ps)

(*  refocus3_word_paren : left_context * parenthesis * right_context
                          -> bool  *)
and refocus3_word_paren (ls, L l, ps)
    = refocus3_word (l :: ls, ps)
  | refocus3_word_paren (ls, R r, ps)
    = refocus3_context (ls, SOME (r, ps))

(*  refocus3_context : left_context * (parenthesis * right_context) option
                       -> bool  *)
and refocus3_context (nil, NONE)
    = iterate3 VAL
  | refocus3_context (nil, SOME (r, ps))
    = iterate3 (NEITHER "unmatched right parenthesis")
  | refocus3_context (l :: ls, NONE)
    = iterate3 (NEITHER "unmatched left parenthesis")
  | refocus3_context (l :: ls, SOME (r, ps))
    = iterate3 (DEC (PR_MATCH (l, r), (ls, ps)))

(*  iterate3 : value_or_decomposition -> bool  *)
and iterate3 VAL
```

```
    = true
  | iterate3 (DEC (PR_MATCH (l, r), C))
    = if l = r
      then refocus3_word C
      else false
  | iterate3 (NEITHER s)
    = false

(*  normalize3 : word -> bool  *)
fun normalize3 w
    = refocus3_word (nil, w)
```

In this abstract machine, `iterate3` implements the contraction rule of the reduction semantics separately from its congruence rules, which are implemented by `refocus3_word`, `refocus3_word_paren`, and `refocus3_context`. This staged structure is remarkable because obtaining this separation for pre-existing abstract machines is known to require non-trivial analyses [44].

5.4 Compressing corridor transitions

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

Clause `refocus4_context (nil, NONE)`:

```
refocus4_context (nil, NONE)
= (* by unfolding the call to refocus4_context *)
iterate4 VAL
= (* by unfolding the call to iterate4 *)
true
```

Clause `refocus4_context (nil, SOME (r, ps))`:

```
refocus4_context (nil, SOME (r, ps))
= (* by unfolding the call to refocus4_context *)
iterate4 (NEITHER "unmatched right parenthesis")
= (* by unfolding the call to iterate4 *)
false
```

Clause `refocus4_context (l :: ls, NONE)`:

```
refocus4_context (l :: ls, NONE)
= (* by unfolding the call to refocus4_context *)
iterate4 (NEITHER "unmatched left parenthesis")
= (* by unfolding the call to iterate4 *)
false
```

Clause `refocus4_context (l :: ls, SOME (r, ps))`:

```
refocus4_context (l :: ls, SOME (r, ps))
= (* by unfolding the call to refocus4_context *)
iterate4 (DEC (PR_MATCH (l, r), (ls, ps)))
= (* by unfolding the call to iterate4 *)
if l = r
then refocus4_word (ls, ps)
else false
```

There is one corollary to the compressions above:

Dead clauses: All of the calls to `iterate4` have been unfolded, and therefore the definition of `iterate4` is dead.

5.5 Renaming transition functions and flattening configurations

The resulting simplified machine is an 'eval/dispatch/continue' abstract machine. We therefore rename `refocus4_word` to `eval5`, `refocus4_word_paren` to `eval5_paren`, and `refocus4_context` to `continue5`. The result reads as follows:

```
(*  eval5 : left_context * right_context -> bool  *)
fun eval5 (ls, nil)
    = continue5 (ls, NONE)
  | eval5 (ls, p :: ps)
    = eval5_paren (ls, p, ps)

(*  eval5_paren : left_context * parenthesis * right_context -> bool  *)
and eval5_paren (ls, L l, ps)
    = eval5 (l :: ls, ps)
  | eval5_paren (ls, R r, ps)
    = continue5 (ls, SOME (r, ps))

(*  continue5 : left_context * (parenthesis * right_context) option
                -> bool  *)
and continue5 (nil, NONE)
    = true
  | continue5 (nil, SOME (r, ps))
    = false
  | continue5 (l :: ls, NONE)
```

```
    = false
  | continue5 (l :: ls, SOME (r, ps))
    = if l = r
      then eval5 (ls, ps)
      else false

(*  normalize5 : word -> bool  *)
fun normalize5 w
    = eval5 (nil, w)
```

5.6 Refunctionalization

The above definitions of `eval5` and `continue5` are in defunctionalized form. The reduction contexts, together with `continue5`, are the first-order counterpart of a function. The higher-order counterpart of this abstract machine reads as follows:

```
(*  eval6 : ((parenthesis * right_context) option -> bool)
            * right_context
            -> bool  *)
fun eval6 (k, nil)
    = k NONE
  | eval6 (k, p :: ps)
    = eval6_paren (k, p, ps)

(*  eval6_paren : ((parenthesis * right_context) option -> bool)
                  * parenthesis * right_context
                  -> bool  *)
and eval6_paren (k, L l, ps)
    = eval6 (fn NONE
                => false
              | (SOME (r, ps))
                => if l = r
                   then eval6 (k, ps)
                   else false,
             ps)
  | eval6_paren (k, R r, ps)
    = k (SOME (r, ps))

(*  normalize6 : word -> bool  *)
fun normalize6 w
    = eval6 (fn NONE
                => true
              | (SOME (r, ps))
                => false,
             w)
```

5.7 Back to direct style

The refunctionalized definition of Section 5.6 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [30, 20]. Its direct-style counterpart reads as follows:

```
val callcc = SMLofNJ.Cont.callcc
val throw = SMLofNJ.Cont.throw

(*  normalize7 : word -> bool  *)
fun normalize7 w
    = callcc (fn top =>
        let (*  eval7 : right_context
                        -> (int * right_context) option  *)
            fun eval7 nil
                = NONE
              | eval7 (p :: ps)
                = eval7_paren (p, ps)
            (*  eval7_paren : parenthesis * right_context
                              -> (int * right_context) option *)
            and eval7_paren (L l, ps)
                = (case eval7 ps
                     of  NONE
                        => throw top false
                      | (SOME (r, ps))
                        => if l = r
                           then eval7 ps
                           else throw top false)
              | eval7_paren (R r, ps)
                = SOME (r, ps)
        in case eval7 w
             of NONE
                => true
              | (SOME (r, pr))
                => false
        end)
```

The resulting definition is that of a recursive function that makes as many calls as it encounters left parentheses and that returns when encountering a right parenthesis and escapes in case of mismatch.

5.8 Closure unconversion

This section is intentionally left blank, since the expressible values in the interpreter of Section 5.7 are first-order.

5.9 Summary

We have refocused the reduction-based recognition function of Section 4 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free recognition functions. Most of the members of this family correspond to something one could write by hand.

5.10 Exercises

Exercise 27 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 23 in Section 4.7. At each step of the derivation, run the tests of Exercise 22 in Section 4.7.*

Exercise 28 *Continue Exercise 26 and refocus the reduction-based recognition function with generalized contraction. Do you end up with a big-step abstract machine in defunctionalized form?*

6 A reduction semantics for normalizing lambda-terms with integers

The goal of this section is to define a one-step reduction function for lambda-terms and to construct the corresponding reduction-based evaluation function.

To define a reduction semantics for lambda-terms with integers (arbitrary literals and a predefined successor function), we specify their abstract syntax (Section 6.1), their notion of contraction (Section 6.2), and their reduction strategy (Section 6.3). We then define a one-step reduction function that decomposes a non-value closure into a potential redex and a reduction context, contracts the potential redex, if it is an actual one, and recomposes the context with the contractum (Section 6.4). 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 (Section 6.5).

The abstract syntax of lambda-terms with integer literals reads as follows. It is completely standard:

```
structure Syn
= struct
    datatype term = LIT of int
                  | IDE of string
                  | LAM of string * term
                  | APP of term * term
  end
```

The S combinator (i.e., $\lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$), for example, is represented as follows:

```
local open Syn
in val S = LAM ("f", LAM ("g", LAM ("x",
             APP (APP (IDE "f", IDE "x"),
                  APP (IDE "g", IDE "x")))))
end
```

In the course of the development, we will make use of environments to represent the bindings of identifiers to denotable values. Our representation is a canonical association list (i.e., list of pairs associating identifiers and denotable values):

```
structure Env
= struct
    type 'a env = (string * 'a) list

    val empty = []                                    (* : 'a env  *)

    fun extend (x, v, env)   (*  : string * 'a * 'a env -> 'a env  *)
        = (x, v) :: env

    fun lookup (x, env)        (*  : string * 'a env -> 'a option  *)
        = let fun search []
                  = NONE
                | search ((x', v) :: env)
                  = if x = x' then SOME v else search env
          in search env
          end
  end
```

In the initial environment, the identifier `succ` denotes the successor function.

More about explicit substitutions can be found in Delia Kesner's recent overview of the field [50]. In this section, we consider an applicative order of Curien's calculus of closures [12, 19].

6.1 Abstract syntax: closures and values

A closure can either be an integer, a ground closure pairing a term and an environment, a combination of closures, or the successor function. A value can either be an integer, the successor function, or a ground closure pairing a lambda-abstraction and an environment. Environments bind identifiers to values.

```
datatype  closure = CLO_INT of int
                  | CLO_GND of Syn.term * bindings
                  | CLO_APP of closure * closure
                  | CLO_SUCC
and         value = VAL_INT of int
```

```
                    | VAL_SUCC
                    | VAL_FUNC of string * Syn.term * bindings
withtype bindings = value Env.env
```

Values are specified with a separate data type. The corresponding embedding of values in closures reads as follows:

```
fun embed_value_in_closure (VAL_INT n)
    = CLO_INT n
  | embed_value_in_closure (VAL_FUNC (x, t, bs))
    = CLO_GND (Syn.LAM (x, t), bs)
  | embed_value_in_closure VAL_SUCC
    = CLO_SUCC
```

The initial environment binds the identifier `succ` to the value `VAL_SUCC`:

```
val initial_bindings = Env.extend ("succ", VAL_SUCC, Env.empty)
```

6.2 Notion of contraction

A potential redex is a ground closure pairing an identifier and an environment, the application of a value to another value, and a ground closure pairing a term application and an environment:

```
datatype potential_redex = PR_IDE of string * bindings
                         | PR_APP of value * value
                         | PR_PROP of Syn.term * Syn.term * bindings
```

A potential redex may be an actual one and trigger a contraction, or it may be stuck. Correspondingly, the following data type accounts for a successful or failed contraction:

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

The string accounts for an error message.

We are now in position to define a contraction function:

- A potential redex `PR_IDE (x, bs)` is an actual one if the identifier `x` is bound in the environment `bs`. If so, the contractum is the denotation of `x` in `bs`.

- A potential redex `PR_APP (v0, v1)` is an actual one if `v0` stands for the successor function and if `v1` stands for an integer value, or if `v0` stands for a functional value that arose from evaluating a ground closure pairing a lambda-abstraction and an environment.

- A ground closure pairing a term application and an environment is contracted into a combination of ground closures.

```
(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_IDE (x, bs))
    = (case Env.lookup (x, bs)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
            => CONTRACTUM (embed_value_in_closure v))
  | contract (PR_APP (VAL_SUCC, VAL_INT n))
    = CONTRACTUM (embed_value_in_closure (VAL_INT (n + 1)))
  | contract (PR_APP (VAL_SUCC, v))
    = ERROR "non-integer value"
  | contract (PR_APP (VAL_FUNC (x, t, bs), v))
    = CONTRACTUM (CLO_GND (t, Env.extend (x, v, bs)))
  | contract (PR_APP (v0, v1))
    = ERROR "non-applicable value"
  | contract (PR_PROP (t0, t1, bs))
    = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)))
```

A non-value closure is stuck whenever it(s iterated reduction) gives rise to a potential redex which is not an actual one, which happens when an identifier does not occur in the current environment (i.e., an identifier is used but not declared), or for ill-typed applications of one value to another.

6.3 Reduction strategy

We seek the left-most inner-most potential redex in a closure.

Reduction contexts: The grammar of reduction contexts reads as follows:

```
datatype context = CTX_MT
                 | CTX_FUN of context * closure
                 | CTX_ARG of value * context
```

Operationally, a context is a closure with a hole, represented inside-out in a zipper-like fashion [47].

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

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * context
```

The decomposition function recursively searches for the left-most innermost redex in a closure. It is usually left unspecified in the literature [40]. As usual, we define it here as a big-step abstract machine with two state-transition functions, decompose_closure and decompose_context between two states: a closure and a context, and a context and a value.

- decompose_closure traverses a given closure and accumulates the reduction context until it finds a value;
- decompose_context dispatches over the accumulated context to determine whether the given closure is a value, the search must continue, or a potential redex has been found.

```
(*  decompose_closure : closure * context -> value_or_decomposition  *)
fun decompose_closure (CLO_INT n, C)
    = decompose_context (C, VAL_INT n)
  | decompose_closure (CLO_GND (Syn.LIT n, bs), C)
    = decompose_context (C, VAL_INT n)
  | decompose_closure (CLO_GND (Syn.IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (Syn.LAM (x, t), bs), C)
    = decompose_context (C, VAL_FUNC (x, t, bs))
  | decompose_closure (CLO_GND (Syn.APP (t0, t1), bs), C)
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_SUCC, C)
    = decompose_context (C, VAL_SUCC)

(*  decompose_context : context * value -> value_or_decomposition  *)
and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)

(*  decompose : closure -> value_or_decomposition  *)
fun decompose c
    = decompose_closure (c, CTX_MT)
```

Recomposition: The recomposition function peels off context layers and constructs the resulting closure, iteratively:

```
(*  recompose : context * closure -> closure  *)
```

```
fun recompose (CTX_MT, c)
    = c
  | recompose (CTX_FUN (C, c1), c0)
    = recompose (C, CLO_APP (c0, c1))
  | recompose (CTX_ARG (v0, C), c1)
    = recompose (C, CLO_APP (embed_value_in_closure v0, c1))
```

Lemma 3 *A closure* `c` *is either a value or there exists a unique context* `C` *such that* `decompose c` *evaluates to* `DEC (pr, C)`, *where* `pr` *is a potential redex.*

Proof 3 *Straightforward (see Exercise 38 in Section 6.7).*

6.4 One-step reduction

As in Section 2.4, we are now in position to define a one-step reduction function as a function that (1) maps a non-value closure 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. The following data type accounts for whether the contraction is successful or the non-value closure is stuck:

```
datatype reduct = REDUCT of closure
                | STUCK of string

(*  reduce : closure -> reduct  *)
fun reduce c
    = (case decompose c
         of (VAL v)
            => REDUCT (embed_value_in_closure v)
          | (DEC (pr, C))
            => (case contract pr
                  of (CONTRACTUM c')
                     => REDUCT (recompose (C, c'))
                   | (ERROR s)
                     => STUCK s))
```

6.5 Reduction-based normalization

As in Section 2.5, a reduction-based normalization function is one that iterates the one-step reduction function until it yields a value (i.e., a fixed point). The following definition uses `decompose` to distinguish between value and non-value closures:

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

```
(*  iterate0 : value_or_decomposition -> result  *)
fun iterate0 (VAL v)
    = RESULT v
  | iterate0 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM c')
            => iterate0 (decompose (recompose (C, c')))
          | (ERROR s)
            => WRONG s)

(*  normalize0 : term -> result  *)
fun normalize0 t
    = iterate0 (decompose (CLO_GND (t, initial_bindings)))
```

6.6 Summary

We have implemented an applicative-order reduction semantics for lambda-terms with integers and explicit substitutions in complete detail. Using this reduction semantics, we have presented a reduction-based applicative-order normalization function.

6.7 Exercises

Exercise 29 *Implement an alternative representation of environments such as*

```
type 'a env = string -> 'a option
```

and verify that defunctionalizing this representation yields a representation isomorphic to the one that uses association lists.

Exercise 30 *Define a function* `embed_potential_redex_in_closure` *that maps a potential redex into a closure.*

Exercise 31 *Show that, for any closure* `c`*, if evaluating* `decompose c` *yields* `DEC (pr, C)`*, then evaluating* `recompose (C, embed_potential_redex_in_closure pr)` *yields* `c`*.*
(Hint: Reason by structural induction over `c`*, using inversion at each step.)*

Exercise 32 *Write a handful of test terms and specify the expected outcome of their normalization.*
(Hint: Take a look at Appendix A.2.)

Exercise 33 *Implement the reduction semantics above in the programming language of your choice (e.g., Haskell or Scheme), and run the tests of Exercise 32.*

Exercise 34 *Write an unparser from closures to the concrete syntax of your choice, and instrument the normalization function of Section 6.5 so that (one way or another) it displays the successive closures in the reduction sequence.*
(Hint: A ground closure can be unparsed as a let expression.) Visualize the reduction sequences of a non-stuck closure and of a stuck closure.

Exercise 35 *Extend the source language with curried addition, subtraction, multiplication, and division, and adjust your implementation.*

Except for the initial bindings and the contraction function, what else needs to be adjusted in your implementation?

Exercise 36 *As a follow-up to Exercise 35, write test terms that use arithmetic operations and specify the expected outcome of their evaluation, and run these tests on your extended implementation.*

Exercise 37 *Extend the data type* `reduct` *with not just an error message but also the problematic potential redex:*

```
datatype reduct = REDUCT of closure
                | STUCK of string * closure
```

(Hint: A function `embed_potential_redex_in_closure` *will come handy.) Adapt your implementation to this new data type, and test it.*

Exercise 38 *In the proof of Lemma 3, do as in the proof of Lemma 1 and write the refunctionalized counterpart of* `decompose` *et al.*

7 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 6.5 into a family of reduction-free normalization functions, i.e., ones where no intermediate closure is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate closures, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 7.1). After inlining the contraction function (Section 7.2), we transform this small-step abstract machine into a big-step one (Section 7.3). This machine exhibits a number of corridor transitions, and we compress them (Section 7.4). We then flatten its configurations and rename its transition functions into something more intuitive (Section 7.5). The resulting abstract machine is in defunctionalized form, and we refunctionalize it (Section 7.6). The result is in continuation-passing style and we re-express it in direct style (Section 7.7). The resulting direct-style function is in closure-converted form, and we closure-unconvert it (Section 7.8). The result is a traditional call-by-value evaluator for lambda-terms; in particular, it is compositional and reduction-free.

Modus operandi: In each of the following subsections, and as in Section 3, we derive successive versions of the normalization function, indexing its components with the number of the subsection. In practice, the reader should run the tests of Exercise 32 in Section 6.7 at each step of the derivation, for sanity value.

7.1 Refocusing: from reduction-based to reduction-free normalization

The normalization function of Section 6.5 is reduction-based because it constructs every intermediate closure 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`:

```
(*  normalize0' : term -> result  *)
fun normalize0' t
    = iterate0 (decompose (recompose (CTX_MT,
                                      CLO_GND (t, initial_bindings))))
```

Refocusing, extensionally: As in Section 3.1, the composition of `decompose` and `recompose` can be deforested into a '`refocus`' function to avoid constructing the intermediate closures in the reduction sequence. Such a deforestation makes the normalization function reduction-free.

Refocusing, intensionally: As in Section 3.1, the `refocus` function can be expressed very simply in terms of the decomposition functions of Section 6.3:

```
(*  refocus : closure * context -> value_or_decomposition  *)
fun refocus (c, C)
    = decompose_closure (c, C)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate1 : value_or_decomposition -> result  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM c')
            => iterate1 (refocus (c', C))
          | (ERROR s)
            => WRONG s)

(*  normalize1 : term -> result  *)
fun normalize1 t
    = iterate1 (refocus (CLO_GND (t, initial_bindings), CTX_MT))
```

This 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 contractum and a reduction context to the next potential redex and reduction context, if there are any in the reduction sequence.

7.2 Inlining the contraction function

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

```
(*  iterate2 : value_or_decomposition -> result  *)
fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_IDE (x, bs), C))
    = (case Env.lookup (x, bs)
         of NONE
            => WRONG "undeclared identifier"
          | (SOME v)
            => iterate2 (refocus (embed_value_in_closure v, C)))
  | iterate2 (DEC (PR_APP (VAL_SUCC, VAL_INT n), C))
    = iterate2 (refocus (embed_value_in_closure (VAL_INT (n + 1)), C))
  | iterate2 (DEC (PR_APP (VAL_SUCC, v), C))
    = WRONG "non-integer value"
  | iterate2 (DEC (PR_APP (VAL_FUNC (x, t, bs), v), C))
    = iterate2 (refocus (CLO_GND (t, Env.extend (x, v, bs)), C))
  | iterate2 (DEC (PR_APP (v0, v1), C))
    = WRONG "non-applicable value"
  | iterate2 (DEC (PR_PROP (t0, t1, bs), C))
    = iterate2 (refocus (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)), C))

(*  normalize2 : term -> result  *)
fun normalize2 t
    = iterate2 (refocus (CLO_GND (t, initial_bindings), CTX_MT))
```

We are now ready to fuse the composition of `iterate2` with `refocus` (shaded just above).

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

The refocused normalization function is small-step abstract machine in the sense that `refocus` (i.e., `decompose_closure` and `decompose_context`) acts as a transition function and `iterate1` as a driver loop that keeps activating `refocus` until a value is obtained. We fuse `iterate2` and `refocus` (i.e., `decompose_closure` and `decompose_` context) so that the resulting function iterate3 is directly applied to the result of decompose_closure and decompose_context. The result is a big-step abstract machine consisting of three (mutually tail-recursive) state-transition functions:

- refocus3_closure is the composition of iterate2 and decompose_closure and a clone of decompose_closure;
- refocus3_context is the composition of iterate2 and decompose_context that directly calls iterate3 over a value or a decomposition instead of returning it to iterate2 as decompose_context did;
- iterate3 is a clone of iterate2 that calls the fused function refocus3_closure.

```
(*  refocus3_closure : closure * context -> result  *)
fun refocus3_closure (CLO_INT n, C)
    = refocus3_context (C, VAL_INT n)
  | refocus3_closure (CLO_GND (Syn.LIT n, bs), C)
    = refocus3_context (C, VAL_INT n)
  | refocus3_closure (CLO_GND (Syn.IDE x, bs), C)
    = iterate3 (DEC (PR_IDE (x, bs), C))
  | refocus3_closure (CLO_GND (Syn.LAM (x, t), bs), C)
    = refocus3_context (C, VAL_FUNC (x, t, bs))
  | refocus3_closure (CLO_GND (Syn.APP (t0, t1), bs), C)
    = iterate3 (DEC (PR_PROP (t0, t1, bs), C))
  | refocus3_closure (CLO_APP (c0, c1), C)
    = refocus3_closure (c0, CTX_FUN (C, c1))
  | refocus3_closure (CLO_SUCC, C)
    = refocus3_context (C, VAL_SUCC)

(*  refocus3_context : context * value -> result  *)
and refocus3_context (CTX_MT, v)
    = iterate3 (VAL v)
  | refocus3_context (CTX_FUN (C, c1), v0)
    = refocus3_closure (c1, CTX_ARG (v0, C))
  | refocus3_context (CTX_ARG (v0, C), v1)
    = iterate3 (DEC (PR_APP (v0, v1), C))

(*  iterate3 : value_or_decomposition -> result  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_IDE (x, bs), C))
    = (case Env.lookup (x, bs)
         of NONE
            => WRONG "undeclared identifier"
          | (SOME v)
            => refocus3_closure (embed_value_in_closure v, C))
  | iterate3 (DEC (PR_APP (VAL_SUCC, VAL_INT n), C))
    = refocus3_closure (embed_value_in_closure (VAL_INT (n + 1)), C)
```

```
  | iterate3 (DEC (PR_APP (VAL_SUCC, v), C))
    = WRONG "non-integer value"
  | iterate3 (DEC (PR_APP (VAL_FUNC (x, t, bs), v), C))
    = refocus3_closure (CLO_GND (t, Env.extend (x, v, bs)), C)
  | iterate3 (DEC (PR_APP (v0, v1), C))
    = WRONG "non-applicable value"
  | iterate3 (DEC (PR_PROP (t0, t1, bs), C))
    = refocus3_closure (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)), C)

(*  normalize3 : term -> result  *)
fun normalize3 t
    = refocus3_closure (CLO_GND (t, initial_bindings), CTX_MT)
```

In this abstract machine, `iterate3` implements the contraction rules of the reduction semantics separately from its congruence rules, which are implemented by `refocus3_closure` and `refocus3_context`. This staged structure is remarkable because obtaining this separation for pre-existing abstract machines is known to require non-trivial analyses [44].

7.4 Compressing corridor transitions

In the abstract machine above, many of the transitions are 'corridor' ones in that they yield configurations for which there is a unique further transition, and so on. Let us compress these transitions. To this end, we cut-and-paste the transition functions above, renaming their indices from `3` to `4`, and consider each of their clauses in turn, making use of the equivalence between `refocus4_closure (embed_value_in_closure v, C)` and `refocus4_context (C, v)`:

Clause `refocus4_closure (CLO_GND (Syn.IDE x, bs), C)`:

```
refocus4_closure (CLO_GND (Syn.IDE x, bs), C)
= (* by unfolding the call to refocus4_closure *)
iterate4 (DEC (PR_IDE (x, bs), C))
= (* by unfolding the call to iterate4 *)
(case Env.lookup (x, bs)
   of NONE
      => WRONG "undeclared identifier"
    | (SOME v)
      => refocus4_closure (embed_value_in_closure v, C))
= (* eureka *)
(case Env.lookup (x, bs)
   of NONE
      => WRONG "undeclared identifier"
    | (SOME v)
      => refocus4_context (C, v))
```

Clause `refocus4_closure (CLO_GND (Syn.APP (t0, t1), bs), C)`:

```
refocus4_closure (CLO_GND (Syn.APP (t0, t1), bs), C)
= (* by unfolding the call to refocus4_closure *)
iterate4 (DEC (PR_PROP (t0, t1, bs)), C)
= (* by unfolding the call to iterate4 *)
refocus4_closure (CLO_GND (t0, bs), CTX_FUN (C, CLO_GND (t1, bs)))
```

There are two corollaries to the compressions above:

Dead clauses: The clauses for non-ground closures are dead, and so is the clause "`iterate4 (VAL v)`." They can therefore be implemented as raising a "`DEAD_CLAUSE`" exception.

Invariants: All transitions to `refocus_closure` are now over ground closures. All live transitions to `iterate4` are now over `DEC (PR_APP (v0, v1), C)`, for some `v0`, `v1`, and `C`.

7.5 Renaming transition functions and flattening configurations

In Section 7.4, the resulting simplified machine is a familiar 'eval/apply/continue' abstract machine [54] operating over ground closures. We therefore rename `refocus4_closure` to `eval5`, `refocus4_context` to `continue5`, and `iterate4` to `apply5`, and flatten the configuration `refocus4_closure (CLO_GND (t, bs), C)` into `eval5 (t, bs, C)` and the configuration `iterate4 (DEC (PR_APP (v0, v1), C))` into `apply5 (v0, v1, C)`, as well as the definition of values and contexts:

```
datatype    value = VAL_INT of int
                  | VAL_SUCC
                  | VAL_FUNC of string * Syn.term * bindings
withtype bindings = value Env.env

datatype context = CTX_MT
                 | CTX_FUN of context * (Syn.term * bindings)
                 | CTX_ARG of value * context

val initial_bindings = Env.extend ("succ", VAL_SUCC, Env.empty)
```

The result reads as follows:

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

```
(*  eval5 : term * bindings * context -> result  *)
fun eval5 (Syn.LIT n, bs, C)
    = continue5 (C, VAL_INT n)
  | eval5 (Syn.IDE x, bs, C)
    = (case Env.lookup (x, bs)
         of NONE
            => WRONG "undeclared identifier"
          | (SOME v)
            => continue5 (C, v))
  | eval5 (Syn.LAM (x, t), bs, C)
    = continue5 (C, VAL_FUNC (x, t, bs))
  | eval5 (Syn.APP (t0, t1), bs, C)
    = eval5 (t0, bs, CTX_FUN (C, (t1, bs)))

(*  continue5 : context * value -> result  *)
and continue5 (CTX_MT, v)
    = RESULT v
  | continue5 (CTX_FUN (C, (t1, bs)), v0)
    = eval5 (t1, bs, CTX_ARG (v0, C))
  | continue5 (CTX_ARG (v0, C), v1)
    = apply5 (v0, v1, C)

(*  apply5 : value * value * context -> result  *)
and apply5 (VAL_SUCC, VAL_INT n, C)
    = continue5 (C, VAL_INT (n + 1))
  | apply5 (VAL_SUCC, v, C)
    = WRONG "non-integer value"
  | apply5 (VAL_FUNC (x, t, bs), v, C)
    = eval5 (t, Env.extend (x, v, bs), C)
  | apply5 (v0, v1, C)
    = WRONG "non-applicable value"

(*  normalize5 : term -> result  *)
fun normalize5 t
    = eval5 (t, initial_bindings, CTX_MT)
```

The resulting abstract machine is the familiar environment-based CEK machine [41].

7.6 Refunctionalization

Like many other big-step abstract machines [3, 4, 5, 16, 23], the abstract machine of Section 7.5 is in defunctionalized form [37]: the reduction contexts, together with `continue5`, are the first-order counterpart of a function. The higher-order counterpart of this abstract machine reads as follows:

```
datatype    value = VAL_INT of int
```

```
                 | VAL_SUCC
                 | VAL_FUNC of string * Syn.term * bindings
withtype bindings = value Env.env

val initial_bindings = Env.extend ("succ", VAL_SUCC, Env.empty)

datatype result = RESULT of value
               | WRONG of string

(*  eval6 : term * bindings * (value -> result) -> result  *)
fun eval6 (Syn.LIT n, bs, k)
    = k (VAL_INT n)
  | eval6 (Syn.IDE x, bs, k)
    = (case Env.lookup (x, bs)
         of NONE
            => WRONG "undeclared identifier"
          | (SOME v)
            => k v)
  | eval6 (Syn.LAM (x, t), bs, k)
    = k (VAL_FUNC (x, t, bs))
  | eval6 (Syn.APP (t0, t1), bs, k)
    = eval6 (t0, bs, fn v0 =>
        eval6 (t1, bs, fn v1 =>
          apply6 (v0, v1, k)))

(*  apply6 : value * value * (value -> result) -> result  *)
and apply6 (VAL_SUCC, VAL_INT n, k)
    = k (VAL_INT (n + 1))
  | apply6 (VAL_SUCC, v, k)
    = WRONG "non-integer value"
  | apply6 (VAL_FUNC (x, t, bs), v, k)
    = eval6 (t, Env.extend (x, v, bs), k)
  | apply6 (v0, v1, k)
    = WRONG "non-applicable value"

(*  normalize6 : term -> result  *)
fun normalize6 t
    = eval6 (t, initial_bindings, fn v => RESULT v)
```

The resulting refunctionalized program is a familiar eval/apply evaluation function in CPS.

7.7 Back to direct style

The refunctionalized definition of Section 7.6 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:

```
datatype    value = VAL_INT of int
                  | VAL_SUCC
                  | VAL_FUNC of string * Syn.term * bindings
withtype bindings = value Env.env

val initial_bindings = Env.extend ("succ", VAL_SUCC, Env.empty)

exception ERROR of string

(*  eval7 : term * bindings -> value  *)
fun eval7 (Syn.LIT n, bs)
    = VAL_INT n
  | eval7 (Syn.IDE x, bs)
    = (case Env.lookup (x, bs)
         of NONE
            => raise (ERROR "undeclared identifier")
          | (SOME v)
            => v)
  | eval7 (Syn.LAM (x, t), bs)
    = VAL_FUNC (x, t, bs)
  | eval7 (Syn.APP (t0, t1), bs)
    = apply7 (eval7 (t0, bs), eval7 (t1, bs))

(*  apply7 : value * value -> value  *)
and apply7 (VAL_SUCC, VAL_INT n)
    = VAL_INT (n + 1)
  | apply7 (VAL_SUCC, v)
    = raise (ERROR "non-integer value")
  | apply7 (VAL_FUNC (x, t, bs), v)
    = eval7 (t, Env.extend (x, v, bs))
  | apply7 (v0, v1)
    = raise (ERROR "non-applicable value")

datatype result = RESULT of value
                | WRONG of string

(*  normalize7 : term -> result  *)
fun normalize7 t
    = RESULT (eval7 (t, initial_bindings))
      handle (ERROR s) => WRONG s
```

The resulting program is a traditional eval/apply evaluation function in direct style and using a top-level exception for run-time errors, à la McCarthy, i.e., a reduction-free normalization function of the kind usually crafted by hand.

7.8 Closure unconversion

The direct-style definition of Section 7.7 is in closure-converted form since its applicable values are introduced with `VAL_SUCC` and `VAL_FUNC`, and eliminated in the clauses of `apply7`. Its higher-order, closure-unconverted equivalent reads as follows.

Expressible and denotable values. The `VAL_FUN` value constructor is higher-order, and caters both for the predefined successor function and for the value of source lambda-abstractions:

```
datatype value = VAL_INT of int
               | VAL_FUN of value -> value

type bindings = value Env.env
```

The occurrences of `VAL_FUN` are shaded below.

Stuck terms. Run-time errors are still implemented by raising an exception:

```
exception ERROR of string
```

Initial bindings. The successor function is now defined in the initial environment:

```
val val_succ = VAL_FUN (fn (VAL_INT n)
                           => VAL_INT (n + 1)
                         | v
                           => raise (ERROR "non-integer value"))

val initial_bindings = Env.extend ("succ", val_succ, Env.empty)
```

The eval/apply component. In `eval8`, the denotation of an abstraction is now inlined, and in `apply8`, applicable values are now directly applied:

```
(*  eval8 : term * bindings -> value  *)
fun eval8 (Syn.LIT n, bs)
    = VAL_INT n
  | eval8 (Syn.IDE x, bs)
    = (case Env.lookup (x, bs)
         of NONE
            => raise (ERROR "undeclared identifier")
          | (SOME v)
            => v)
  | eval8 (Syn.LAM (x, t), bs)
    = VAL_FUN (fn v => eval8 (t, Env.extend (x, v, bs)))
  | eval8 (Syn.APP (t0, t1), bs)
    = apply8 (eval8 (t0, bs), eval8 (t1, bs))
```

```
(*  apply8 : value * value -> value  *)
and apply8 (VAL_FUN f, v)
    = f v
  | apply8 (v0, v1)
    = raise (ERROR "non-applicable value")
```

The top-level definition. A term `t` is evaluated in the initial environment. If this evaluation completes, the resulting value is the result of the normalization function. If this evaluation goes wrong, the given term is stuck.

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

(*  normalize8 : term -> result  *)
fun normalize8 t
    = RESULT (eval8 (t, initial_bindings))
      handle (ERROR s) => WRONG s
```

The resulting program is a traditional eval/apply function in direct style that uses a top-level exception for run-time errors. It is also compositional.

7.9 Summary

We have refocused the reduction-based normalization function of Section 6 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions. Most of the members of this family are ML implementations of independently known semantic artifacts and coincide with what one would have independently written by hand.

7.10 Exercises

Exercise 39 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 33 in Section 6.7. At each step of the derivation, run the tests of Exercise 32 in Section 6.7.*

Exercise 40 *Up to and including the normalization function of Section 7.5, it is simple to visualize the successive closures in the reduction sequence, namely by instrumenting* `iterate1`, `iterate2`, `iterate3`, `iterate4`, *and* `apply5`. *Do you agree? What about from Section 7.6 and onwards?*

Exercise 41 *Would it make sense, in the definition of* `normalize6`, *to take* `fn v => v` *as the initial continuation? If so, what would be the definition of* `normalize7` *and what would be its type?*

Exercise 42 *In Section 7.7, we have transformed the evaluator of Section 7.6 into direct style, and then in Section 7.8, we have closure-unconverted it. However, the the evaluator of Section 7.6 is also in closure-converted form:*

1. *closure-unconvert the evaluator of Section 7.6; the result should be a compositional evaluator in CPS with the following data type of expressible values:*

```
datatype value = VAL_INT of int
               | VAL_FUN of value * (value -> result) -> result
    and result = RESULT of value
               | WRONG of string
```

2. *transform this compositional evaluator into direct style, and verify that the result coincides with the evaluator of Section 7.8.*

Exercise 43 *Compare the evaluation functions of Section 7.8 and of Appendix B; of Section 7.7 and of Appendix C; of Section 7.6 and of Appendix D; and of Section 7.5 and of Appendix E. This comparison should explain your feeling of* déjà vu*.*

8 A reduction semantics for normalizing lambda-terms with integers and first-class continuations

In this section, we extend the source language of Section 6 with one more predefined identifier in the initial environment: `call/cc`. Presentationally, we therefore single out the increment over Section 6 rather than giving a stand-alone reduction semantics.

8.1 Abstract syntax: closures, values, and contexts

In addition to being an integer, a ground closure pairing a term and an environment, a combination of closures, or the successor function, a closure can also be the call/cc function or a reified context. Correspondingly, in addition to being an integer, the successor function, or a ground closure pairing a lambda-abstraction and an environment, a value can also be the call/cc function or a reified context. Environments bind identifiers to values.

```
datatype  closure = CLO_INT of int
                  | CLO_GND of Syn.term * bindings
                  | CLO_APP of closure * closure
                  | CLO_SUCC
                  | CLO_CWCC
                  | CLO_CONT of context
and         value = VAL_INT of int
```

```
                   | VAL_SUCC
                   | VAL_FUNC of string * Syn.term * bindings
                   | VAL_CWCC
                   | VAL_CONT of context
and        context = CTX_MT
                   | CTX_FUN of context * closure
                   | CTX_ARG of value * context
withtype bindings = value Env.env
```

Values are specified with a separate data type. The corresponding embedding of values in closures reads as follows:

```
fun embed_value_in_closure (VAL_INT n)
    = CLO_INT n
  | embed_value_in_closure (VAL_FUNC (x, t, bs))
    = CLO_GND (Syn.LAM (x, t), bs)
  | embed_value_in_closure VAL_SUCC
    = CLO_SUCC
  | embed_value_in_closure VAL_CWCC
    = CLO_CWCC
  | embed_value_in_closure (VAL_CONT C)
    = CLO_CONT C
```

The initial environment also binds the identifier `call/cc` to the value `VAL_CWCC`:

```
val initial_bindings = Env.extend ("call/cc", VAL_CWCC,
                         Env.extend ("succ", VAL_SUCC,
                           Env.empty))
```

8.2 Notion of contraction

A potential redex is as in Section 6.2. The contraction function also accounts for first-class continuations, and is therefore context sensitive in that it maps a potential redex and its reduction context to a contractum and a reduction context (possibly another one):

```
datatype contractum_or_error = CONTRACTUM of closure * context
                             | ERROR of string
```

Compared to Section 6.2, the new clauses are shaded:

```
(*  contract : potential_redex * context -> contractum_or_error  *)
fun contract (PR_IDE (x, bs), C)
    = (case Env.lookup (x, bs)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
```

```
          => CONTRACTUM (embed_value_in_closure v, C))
    | contract (PR_APP (VAL_SUCC, VAL_INT n), C)
      = CONTRACTUM (embed_value_in_closure (VAL_INT (n + 1)), C)
    | contract (PR_APP (VAL_SUCC, v), C)
      = ERROR "non-integer value"
    | contract (PR_APP (VAL_FUNC (x, t, bs), v), C)
      = CONTRACTUM (CLO_GND (t, Env.extend (x, v, bs)), C)
    | contract (PR_APP (VAL_CWCC, v), C)
      = CONTRACTUM (CLO_APP (embed_value_in_closure v, CLO_CONT C), C)

    | contract (PR_APP (VAL_CONT C', v), C)
      = CONTRACTUM (embed_value_in_closure v, C')
    | contract (PR_APP (v0, v1), C)
      = ERROR "non-applicable value"
    | contract (PR_PROP (t0, t1, bs), C)
      = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)), C)
```

Each of the clauses implements a contraction rule, and all of the rules are context insensitive, except the two shaded ones:

- Applying call/cc to a value leads to this value being applied to a representation of the current context. This context is then said to be "captured" and its representation is said to be "reified."
- Applying a captured context to a value yields a contractum consisting of this value *and the captured context* (instead of the current context, which is discarded).

8.3 Reduction strategy

We seek the left-most inner-most potential redex in a closure.

Decomposition: The decomposition function is defined as in Section 6.3 but for the following two clauses:

```
fun decompose_closure ...
    = ...
  | decompose_closure (CLO_CWCC, C)
    = decompose_context (C, VAL_CWCC)
  | decompose_closure (CLO_CONT C', C)
    = decompose_context (C, VAL_CONT C')
```

Recomposition: The recomposition function is defined as in Section 6.3.

Lemma 4 *A closure* `c` *is either a value or there exists a unique context* `C` *such that* `decompose t` *evaluates to* `DEC (pr, C)`*, where* `pr` *is a potential redex.*

Proof 4 *Straightforward.*

8.4 One-step reduction

The one-step reduction function is as in Section 6.4, save for the contraction function being context-sensitive, as shaded just below:

```
(*  reduce : closure -> reduct  *)
fun reduce c
    = (case decompose c
         of (VAL v)
            => REDUCT (embed_value_in_closure v)
          | (DEC (pr, C))
            => (case contract (pr, C)
                  of (CONTRACTUM (c', C'))
                     => REDUCT (recompose (C', c'))
                   | (ERROR s)
                     => STUCK s))
```

8.5 Reduction-based normalization

The reduction-based normalization function is as in Section 8.5, save for the contraction function being context-sensitive, as shaded just below:

```
(*  iterate0 : value_or_decomposition -> result  *)
fun iterate0 (VAL v)
    = RESULT v
  | iterate0 (DEC (pr, C))
    = (case contract (pr, C)
         of (CONTRACTUM (c', C'))
            => iterate0 (decompose (recompose (C', c')))
          | (ERROR s)
            => WRONG s)

(*  normalize0 : term -> result  *)
fun normalize0 t
    = iterate0 (decompose (CLO_GND (t, initial_bindings)))
```

8.6 Summary

We have minimally extended the applicative-order reduction semantics of Section 6 with call/cc.

8.7 Exercises

As a warmup for Exercise 44, here is an interface to first-class continuations in Standard ML of New Jersey that reifies the current continuation as a function:

```
fun callcc f
    = SMLofNJ.Cont.callcc
        (fn k => f (fn v => SMLofNJ.Cont.throw k v))
```

We also assume that `succ` denotes the successor function.

- Consider the following term:

  ```
  succ (succ (callcc (fn k => succ 10)))
  ```

  In the course of reduction, `k` is made to denote a first-class continuation that is not used. This term is equivalent to one that does not use call/cc, namely

  ```
  succ (succ (succ 10))
  ```

  and evaluating it yields `13`.

- Consider now the following term that captures a continuation and then applies it:

  ```
  succ (succ (callcc (fn k => succ (k 10))))
  ```

  In the course of reduction, `k` is made to denote a first-class continuation that is then applied. When it is applied, the current continuation is discarded and replaced by the captured continuation, as if the source term had been

  ```
  succ (succ 10)
  ```

  and the result of evaluation is `12`.

  In the reduction semantics of this section, the source term reads as follows:

  ```
  APP (IDE "succ",
       APP (IDE "succ",
            APP (IDE "call/cc",
                 LAM ("k", APP (IDE "succ",
                               APP (IDE "k", LIT 10))))))
  ```

  As for the captured continuation, it reads as follows:

  ```
  CLO_CONT (CTX_ARG (VAL_SUCC, CTX_ARG (VAL_SUCC, CTX_MT)))
  ```

  Applying it to `VAL_INT 10` has the effect of discarding the current context, and eventually leads to `RESULT (VAL_INT 12)`.

Exercise 44 *Write a handful of test terms that use call/cc and specify the expected outcome of their normalization.*

Exercise 45 *Implement the reduction semantics above in the programming language of your choice (e.g., Haskell or Scheme), and run the tests of Exercise 44.*

Exercise 46 *Extend the unparser of Exercise 34 in Section 6.7 to cater for first-class continuations, and visualize the reduction sequence of a closure that uses call/cc.*

9 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 8.5 into a family of reduction-free normalization functions. Presentationally, we single out the increment over Section 7 rather than giving a stand-alone derivation.

9.1 Refocusing: from reduction-based to reduction-free normalization

As usual, the refocus function is defined as continuing the decomposition in situ:

```
(*  refocus : closure * context -> value_or_decomposition  *)
fun refocus (c, C)
    = decompose_closure (c, C)
```

The refocused evaluation function reads as follows. Except for the context-sensitive contraction function, it is the same as in Section 7.1:

```
(*  iterate1 : value_or_decomposition -> result  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, C))
    = (case contract (pr, C)
         of (CONTRACTUM (c', C'))
            => iterate1 (refocus (c', C'))
          | (ERROR s)
            => WRONG s)

(*  normalize1 : term -> result  *)
fun normalize1 ...
    = ...
```

9.2 Inlining the contraction function

Compared to Section 7.2, there are two new clauses:

```
(*  iterate2 : value_or_decomposition -> result  *)
fun iterate2 ...
    = ...
  | iterate2 (DEC (PR_APP (VAL_CWCC, v), C))
    = iterate2 (refocus (CLO_APP (embed_value_in_closure v, CLO_CONT C), C))
  | iterate2 (DEC (PR_APP (VAL_CONT C', v), C))
    = iterate2 (refocus (embed_value_in_closure v, C'))
  | iterate2 ...
    = ...
```

...

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

Compared to Section 7.3, there are two new clauses in `refocus3_closure` and in `iterate3`; the definition of `refocus3_context` is not affected:

```
(*  refocus3_closure : closure * context -> result  *)
fun refocus3_closure ...
    = ...
  | refocus3_closure (CLO_CWCC, C)
    = refocus3_context (C, VAL_CWCC)
  | refocus3_closure (CLO_CONT C', C)
    = refocus3_context (C, VAL_CONT C')

(*  refocus3_context : context * value -> result  *)
fun refocus3_context ...
    = ...

(*  iterate3 : value_or_decomposition -> result  *)
and iterate3 ...
    = ...
  | iterate3 (DEC (PR_APP (VAL_CWCC, v), C))
    = refocus3_closure (CLO_APP (embed_value_in_closure v, CLO_CONT C), C)
  | iterate3 (DEC (PR_APP (VAL_CONT C', v), C))
    = refocus3_closure (embed_value_in_closure v, C')
  | iterate3 ...
    = ...
```

9.4 Compressing corridor transitions

Compared to Section 7.4, there are two new opportunities to compress corridor transitions:

Clause `iterate4 (DEC (PR_APP (VAL_CWCC, v), C))`:

```
iterate4 (DEC (PR_APP (VAL_CWCC, v), C))
= (* by unfolding the call to iterate4 *)
refocus4_closure (CLO_APP (embed_value_in_closure v, CLO_CONT C), C)
= (* by unfolding the call to refocus4_closure *)
refocus4_closure (embed_value_in_closure v, CTX_FUN (C, CLO_CONT C))
= (* eureka *)
refocus4_context (CTX_FUN (C, CLO_CONT C), v)
= (* by unfolding the call to refocus4_context *)
refocus4_closure (CLO_CONT C, CTX_ARG (v, C))
= (* by unfolding the call to refocus4_closure *)
refocus4_context (CTX_ARG (v, C), VAL_CONT C)
= (* by unfolding the call to refocus4_context *)
iterate4 (DEC (PR_APP (v, VAL_CONT C), C))
```

Clause `iterate4 (DEC (PR_APP (VAL_CONT C', v), C))`:

```
iterate4 (DEC (PR_APP (VAL_CONT C', v), C))
= (* by unfolding the call to iterate4 *)
refocus4_closure (embed_value_in_closure v, C')
= (* eureka *)
refocus4_context (C', v)
```

The corollaries to the compressions above are the same as in Section 7.4:

Dead clauses: The clauses for non-ground closures are dead, and so is the clause "`iterate4 (VAL v)`." They can therefore be implemented as raising a "`DEAD_CLAUSE`" exception.

Invariants: All transitions to `refocus_closure` are now over ground closures. All live transitions to `iterate4` are now over `DEC (PR_APP (v0, v1), C)`, for some `v0`, `v1`, and `C`.

9.5 Renaming transition functions and flattening configurations

The renamed and flattened abstract machine is the familiar CEK machine with call/cc:

```
datatype    value = ...
                  | VAL_CWCC
                  | VAL_CONT of context
and       context = ...
withtype bindings = ...

val initial_bindings = Env.extend ("call/cc", VAL_CWCC,
                         Env.extend ("succ", VAL_SUCC,
                           Env.empty))
```

```
(*  eval5 : term * bindings * context -> result  *)
fun eval5 ...
    = ...

(*  continue5 : context * value -> result  *)
and continue5 ...
    = ...

(*  apply5 : value * value * context -> result  *)
and apply5 ...
    = ...
  | apply5 (VAL_CWCC, v, C)
    = apply5 (v, VAL_CONT C, C)
  | apply5 (VAL_CONT C', v, C)
    = continue5 (C', v)
  | apply5 ...
    = ...

(*  normalize5 : term -> result  *)
fun normalize5 ...
    = eval5 ...
```

9.6 Refunctionalization

The higher-order counterpart of the abstract machine of Section 9.5 reads as follows:

```
datatype    value = ...
                  | VAL_CWCC
                  | VAL_CONT of value -> result
withtype bindings = ...

val initial_bindings = Env.extend ("call/cc", VAL_CWCC,
                         Env.extend ("succc", VAL_SUCC,
                           Env.empty))

(*  eval6 : term * bindings * (value -> result) -> result  *)
fun eval6 ...
    = ...

(*  apply6 : value * value * (value -> result) -> result  *)
and apply6 ...
    = ...
  | apply6 (VAL_CWCC, v, k)
    = apply6 (v, VAL_CONT k, k)
  | apply6 (VAL_CONT k', v, k)
    = k' v
  | apply6 ...
    = ...
```

```
(*  normalize6 : term -> result  *)
fun normalize6 ...
    = ...
```

The resulting refunctionalized program is a familiar eval/apply evaluation function in CPS [46, Fig. 1, p. 295].

9.7 Back to direct style

The direct-style counterpart of the evaluation function of Section 9.6 reads as follows [32]:

```
(*  eval7 : term * bindings -> value  *)
fun eval7 ...
    = ...

(*  apply7 : value * value -> value  *)
and apply7 ...
    = ...
  | apply7 (VAL_CWCC, v)
    = SMLofNJ.Cont.callcc (fn k => apply7 (v, VAL_CONT k))
  | apply7 (VAL_CONT k', v)
    = SMLofNJ.Cont.throw k' v
  | apply7 ...
    = ...

(*  normalize7 : term -> result  *)
fun normalize7 ...
    = ...
```

The resulting program is a traditional eval/apply evaluation function in direct style that uses call/cc to implement call/cc, meta-circularly.

9.8 Closure unconversion

As in Section 7.8, the direct-style definition of Section 9.7 is in closure-converted form since its applicable values are introduced with `VAL_SUCC` and `VAL_FUNC`, and eliminated in the clauses of `apply7`. Its higher-order, closure-unconverted equivalent reads as follows.

Expressible and denotable values. The `VAL_FUN` value constructor is higher-order, and caters both for the predefined successor function, for the predefined call/cc function, for the value of source lambda-abstractions, and for captured continuations:

```
datatype value = VAL_INT of int
               | VAL_FUN of value -> value

type bindings = value Env.env
```

Initial bindings. The successor function is now defined in the initial environment:

```
val val_succ = VAL_FUN ...

val val_cwcc = VAL_FUN (fn (VAL_FUN f)
                           => SMLofNJ.Cont.callcc (fn k =>
                                f (VAL_FUN (fn v =>
                                     SMLofNJ.Cont.throw k v)))
                         | _
                           => raise (WRONG "non-applicable value"))

val initial_bindings = Env.extend ("call/cc", val_cwcc,
                         Env.extend ("succ", val_succ,
                           Env.empty))
```

The eval/apply component. The evaluation function is the same as in Section 7.8:

```
(*  eval8 : term * bindings -> value  *)
fun eval8 ...
    = ...

(*  apply8 : value * value -> value  *)
and apply8 ...
    = ...
```

The top-level definition. The top-level definition is the same as in Section 7.8:

```
(*  normalize8 : term -> result  *)
fun normalize8 ...
    = ...
```

The resulting program is a traditional eval/apply function in direct style that uses a top-level exception for run-time errors. It is also compositional.

9.9 Summary

We have outlined the derivation from the reduction-based normalization function of Section 8 into a small-step abstract machine and into a family of corresponding reduction-free normalization functions. Most of the members of this family are ML implementations of independently known semantic artifacts and coincide with what one usually writes by hand.

9.10 Exercises

Exercise 47 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 45 in Section 8.7. At each step of the derivation, run the tests of Exercise 44 in Section 8.7.*

Exercise 48 *Up to and including the normalization function of Section 9.5, it is simple to visualize the successive closures in the reduction sequence, namely by instrumenting* `iterate1`, `iterate2`, `iterate3`, `iterate4`, *and* `apply5`. *Do you agree? What about from Section 9.6 and onwards?*

Exercise 49 *Would it make sense, in the definition of* `normalize6`, *to take* `fn v => v` *as the initial continuation? If so, what would be the definition of* `normalize7` *and what would be its type?*

Exercise 50 *In Section 9.7, we have transformed the evaluator of Section 9.6 into direct style, and then in Section 9.8, we have closure-unconverted it. However, the the evaluator of Section 9.6 is also in closure-converted form:*

1. *closure-unconvert the evaluator of Section 9.6; the result should be a compositional evaluator in CPS with the following data type of expressible values:*

```
datatype value = VAL_INT of int
               | VAL_FUN of value * (value -> result) -> result
    and result = RESULT of value
               | WRONG of string
```

2. *transform this compositional evaluator into direct style, and verify that the result coincides with the evaluator of Section 9.8.*

10 A reduction semantics for flattening binary trees outside in

The goal of this section is to define a one-step flattening function over binary trees, using a left-most outermost strategy, and to construct the corresponding reduction-based flattening function.

To define a reduction semantics for binary trees, we specify their abstract syntax (Section 10.1), a notion of contraction (Section 10.2), and the left-most outermost reduction strategy (Section 10.3). We then define a one-step reduction function that decomposes a tree which is not in normal form into a redex and a reduction context, contracts the redex, and recomposes the context with the contractum (Section 10.4). 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 (Section 10.5).

10.1 Abstract syntax: terms and values

Terms: A tree is either a stub, a leaf holding an integer, or a node holding two subtrees:

```
datatype tree = STUB
              | LEAF of int
              | NODE of tree * tree
```

The flattening rules are as follows: the unit element is neutral on the left and on the right of the node constructor, and the product is associative.

$$
\begin{aligned}
\texttt{NODE (STUB, t)} &\longleftrightarrow \texttt{t} \\
\texttt{NODE (t, STUB)} &\longleftrightarrow \texttt{t} \\
\texttt{NODE (NODE (t1, t2), t3)} &\longleftrightarrow \texttt{NODE (t1, NODE (t2, t3))}
\end{aligned}
$$

Normal forms: Arbitrarily, we pick flat, list-like trees as normal forms. We specify them with the following specialized data type:

```
datatype tree_nf = STUB_nf
                 | NODE_nf of int * tree_nf
```

Values: Rather than defining values as normal forms, as in the previous sections, we choose to represent them as a pair: a term of type `tree` and its isomorphic representation of type `tree_nf`:

```
type value = tree * tree_nf
```

This representation is known as "glueing" since Yves Lafont's PhD thesis [52, Appendix A], and is also classically used in the area of partial evaluation [6].

10.2 Notion of contraction

We introduce a notion of reduction by orienting the conversion rules into contraction rules, and by specializing the second one as mapping a leaf into a flat binary tree:

$$
\begin{aligned}
\texttt{NODE (STUB, t)} &\longrightarrow \texttt{t} \\
\texttt{NODE (LEAF n, STUB)} &\longleftarrow \texttt{LEAF n} \\
\texttt{NODE (NODE (t11, t12), t2)} &\longrightarrow \texttt{NODE (t11, NODE (t12, t2))}
\end{aligned}
$$

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

```
datatype potential_redex = PR_LEFT_STUB of tree
                         | PR_LEAF of int
                         | PR_ASSOC of tree * tree * tree

datatype contractum_or_error = CONTRACTUM of tree
                             | ERROR of string

(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_LEFT_STUB t)
    = CONTRACTUM t
  | contract (PR_LEAF n)
    = CONTRACTUM (NODE (LEAF n, STUB))
  | contract (PR_ASSOC (t11, t12, t2))
    = CONTRACTUM (NODE (t11, NODE (t12, t2)))
```

10.3 Reduction strategy

We seek the left-most outer-most redex in a tree.

Reduction contexts: The grammar of reduction contexts reads as follows:

```
datatype context = CTX_MT
                 | CTX_RIGHT of int * context
```

Decomposition: A tree is in normal form (i.e., it does not contain any potential redex) or it can be decomposed into a potential redex and a reduction context:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * context
```

The decomposition function recursively searches for the left-most outer-most redex in a term. As always, we define it as a big-step abstract machine. This abstract machine has three auxiliary functions, `decompose_tree`, `decompose_node`, and `decompose_context` between three states – a term and a context, two sub-terms and a context, and a context and a value.

- `decompose_tree` dispatches over the given tree;
- `decompose_node` dispatches over the left sub-tree of a given tree;
- `decompose_context` dispatches on the accumulated context to determine whether the given term is a value, a potential redex has been found, or the search must continue.

```
(*  decompose_tree : tree * context -> value_or_decomposition  *)
fun decompose_tree (STUB, C)
    = decompose_context (C, (STUB, STUB_nf))
  | decompose_tree (LEAF n, C)
    = DEC (PR_LEAF n, C)
  | decompose_tree (NODE (t1, t2), C)
    = decompose_node (t1, t2, C)

(*  decompose_node : tree * tree * context -> value_or_decomposition  *)
and decompose_node (STUB, t2, C)
    = DEC (PR_LEFT_STUB t2, C)
  | decompose_node (LEAF n, t2, C)
    = decompose_tree (t2, CTX_RIGHT (n, C))
  | decompose_node (NODE (t11, t12), t2, C)
    = DEC (PR_ASSOC (t11, t12, t2), C)

(*  decompose_context : context * value -> value_or_decomposition  *)
and decompose_context (CTX_MT, (t', t_nf))
    = VAL (t', t_nf)
  | decompose_context (CTX_RIGHT (n, C), (t', t_nf))
    = decompose_context (C, (NODE (LEAF n, t'), NODE_nf (n, t_nf)))

(*  decompose : tree -> value_or_decomposition  *)
fun decompose t
    = decompose_tree (t, CTX_MT)
```

Recomposition: The recomposition function peels off context layers and constructs the resulting tree, iteratively:

```
(*  recompose : context * tree -> tree  *)
fun recompose (CTX_MT, t)
    = t
  | recompose (CTX_RIGHT (n1, C), t2)
    = recompose (C, NODE (LEAF n1, t2))
```

Lemma 5 *A tree* `t` *is either in normal form or there exists a unique context* `C` *such that* `decompose t` *evaluates to* `DEC (pr, C)`*, where* `pr` *is a potential redex.*

Proof 5 *Straightforward (see Exercise 56 in Section 10.7).*

10.4 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a tree that is not in normal form 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. The following data type accounts for whether the contraction is successful or the non-value term is stuck:

```
datatype reduct = REDUCT of tree
                | STUCK of string

(*  reduce : tree -> reduct  *)
fun reduce t
    = (case decompose t
         of (VAL (t', t_nf))
            => REDUCT t'
          | (DEC (pr, C))
            => (case contract pr
                  of (CONTRACTUM t')
                     => REDUCT (recompose (C, t'))
                   | (ERROR s)
                     => STUCK s))
```

10.5 Reduction-based normalization

The following reduction-based normalization function iterates the one-step reduction function until it yields a normal form:

```
datatype result = RESULT of tree_nf
                | WRONG of string

(*  iterate0 : value_or_decomposition -> result  *)
fun iterate0 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate0 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate0 (decompose (recompose (C, t')))
          | (ERROR s)
            => WRONG s)

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

10.6 Summary

We have implemented a reduction semantics for flattening binary trees, in complete detail. Using this reduction semantics, we have presented a reduction-based normalization function.

10.7 Exercises

Exercise 51 *Define a function* `embed_potential_redex_in_tree` *that maps a potential redex into a tree.*

Exercise 52 *Show that, for any tree* `t`*, if evaluating* `decompose t` *yields* `DEC (pr, C)`*, then evaluating* `recompose (C, embed_potential_redex_in_tree pr)` *yields* `t`*.*
(Hint: Reason by structural induction over `t`*, using inversion at each step.)*

Exercise 53 *Write a handful of test trees and specify the expected outcome of their normalization.*

Exercise 54 *Implement the reduction semantics above in the programming language of your choice, and run the tests of Exercise 53.*

Exercise 55 *Write an unparser from trees to the concrete syntax of your choice, and instrument the normalization function of Section 10.5 so that (one way or another) it displays the successive trees in the reduction sequence.*

Exercise 56 *In the proof of Lemma 5, do as in the proof of Lemma 1 and write the refunctionalized counterpart of* `decompose` *et al.*

Exercise 57 *Pick another notion of normal form (e.g., flat, list-like trees on the left instead of on the right) and define the corresponding reduction-based normalization function,* mutatis mutandis.

Exercise 58 *Revisit either of the previous pairs of sections using glueing.*

11 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 10.5 into a family of reduction-free normalization functions, i.e., one where no intermediate tree is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate trees, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 11.1). After inlining the contraction function (Section 11.2), we transform this small-step abstract machine into a big-step one (Section 11.3). This abstract machine exhibits a number of corridor transitions, and we compress them (Section 11.4). We then flatten its configurations and rename its transition functions into something more intuitive (Section 11.5). The resulting abstract machine is in defunctionalized form, and we refunctionalize it (Section 11.6). The result is in continuation-passing style and we re-express it in direct style (Section 11.7). The resulting direct-style function is a traditional flatten function that incrementally flattens its input from the top down.

Modus operandi: In each of the following subsections, and as always, we derive successive versions of the normalization function, indexing its components with the number of the subsection. In practice, the reader should run the tests of Exercise 53 in Section 10.7 at each step of the derivation, for sanity value.

11.1 Refocusing: from reduction-based to reduction-free normalization

The normalization function of Section 10.5 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`:

```
(*  normalize0' : tree -> result  *)
fun normalize0' t
    = iterate0 (decompose (recompose (CTX_MT, t)))
```

Refocusing, extensionally: The composition of `decompose` and `recompose` can be deforested into a '`refocus`' function to avoid constructing the intermediate terms in the reduction sequence. Such a deforestation makes the normalization function reduction-free.

Refocusing, intensionally: As usual, the `refocus` function can be expressed very simply in terms of the decomposition functions of Section 10.3:

```
(*  refocus : term * context -> value_or_decomposition  *)
fun refocus (t, C)
    = decompose_tree (t, C)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate1 : value_or_decomposition -> result  *)
fun iterate1 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate1 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate1 (refocus (t', C))
          | (ERROR s)
            => WRONG s)

(*  normalize1 : tree -> result  *)
fun normalize1 t
    = iterate1 (refocus (t, CTX_MT))
```

This 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 contractum and a reduction context to the next redex and reduction context, if there are any in the reduction sequence.

11.2 Inlining the contraction function

We first inline 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`:

```
(*  iterate2 : value_or_decomposition -> result  *)
fun iterate2 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate2 (DEC (PR_LEFT_STUB t, C))
    = iterate2 (refocus (t, C))
  | iterate2 (DEC (PR_LEAF n, C))
    = iterate2 (refocus (NODE (LEAF n, STUB), C))
  | iterate2 (DEC (PR_ASSOC (t11, t12, t2), C))
    = iterate2 (refocus (NODE (t11, NODE (t12, t2)), C))

(*  normalize2 : tree -> result  *)
fun normalize2 t
    = iterate2 (refocus (t, CTX_MT))
```

We are now ready to fuse the composition of `iterate2` with `refocus` (shaded just above).

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

The refocused normalization function is a small-step abstract machine in the sense that `refocus` (i.e., `decompose_tree`, `decompose_node`, and `decompose_context`) acts as a transition function and `iterate1` as a driver loop that keeps activating `refocus` until a value is obtained. We fuse `iterate2` and `refocus` (i.e., `decompose_tree`, `decompose_node`, and `decompose_context`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_tree`, `decompose_node`, and `decompose_context`. The result is a big-step abstract machine consisting of four (mutually tail-recursive) state-transition functions:

- `refocus3_tree` is the composition of `iterate2` and `decompose_tree` and a clone of `decompose_tree` that directly calls `iterate3` over a leaf instead of returning it to `iterate2` as `decompose_tree` did;
- `refocus3_node` is the composition of `iterate2` and `decompose_node` and a clone of `decompose_node` that directly calls `iterate3` over a decomposition instead of returning it to `iterate2` as `decompose_node` did;
- `refocus3_context` is the composition of `iterate2` and `decompose_context` that directly calls `iterate3` over a value or a decomposition instead of returning it to `iterate2` as `decompose_context` did;

- `iterate3` is a clone of `iterate2` that calls the fused function `refocus3_tree`.

```
(*  refocus3_tree : tree * context -> result  *)
fun refocus3_tree (STUB, C)
    = refocus3_context (C, (STUB, STUB_nf))
  | refocus3_tree (LEAF n, C)
    = iterate3 (DEC (PR_LEAF n, C))
  | refocus3_tree (NODE (t1, t2), C)
    = refocus3_node (t1, t2, C)

(*  refocus3_node : tree * tree * context -> result  *)
and refocus3_node (STUB, t2, C)
    = iterate3 (DEC (PR_LEFT_STUB t2, C))
  | refocus3_node (LEAF n, t2, C)
    = refocus3_tree (t2, CTX_RIGHT (n, C))
  | refocus3_node (NODE (t11, t12), t2, C)
    = iterate3 (DEC (PR_ASSOC (t11, t12, t2), C))

(*  refocus3_context : context * value -> result  *)
and refocus3_context (CTX_MT, (t', t_nf))
    = iterate3 (VAL (t', t_nf))
  | refocus3_context (CTX_RIGHT (n, C), (t', t_nf))
    = refocus3_context (C, (NODE (LEAF n, t'), NODE_nf (n, t_nf)))

(*  iterate3 : value_or_decomposition -> result  *)
and iterate3 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate3 (DEC (PR_LEFT_STUB t, C))
    = refocus3_tree (t, C)
  | iterate3 (DEC (PR_LEAF n, C))
    = refocus3_tree (NODE (LEAF n, STUB), C)
  | iterate3 (DEC (PR_ASSOC (t11, t12, t2), C))
    = refocus3_tree (NODE (t11, NODE (t12, t2)), C)

(*  normalize3 : tree -> result  *)
fun normalize3 t
    = refocus3_tree (t, CTX_MT)
```

This abstract machine is staged since `iterate3` implements the contraction rules of the reduction semantics separately from its congruence rules, which are implemented by `refocus3_tree`, `refocus3_node` and `refocus3_context`.

11.4 Compressing corridor transitions

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

Clause `refocus4_tree (LEAF n, C)`:

```
refocus4_tree (LEAF n, C)
= (* by unfolding the call to refocus4_tree *)
iterate4 (DEC (PR_LEAF n, C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (NODE (LEAF n, STUB), C)
= (* by unfolding the call to refocus4_tree *)
refocus4_node (LEAF n, STUB, C)
= (* by unfolding the call to refocus4_node *)
refocus4_tree (STUB, CTX_RIGHT (n, C))
= (* by unfolding the call to refocus4_tree *)
refocus4_context (CTX_RIGHT (n, C), (STUB, STUB_nf))
= (* by unfolding the call to refocus4_context *)
refocus4_context (C, (NODE (LEAF n, STUB), NODE_nf (n, STUB_nf)))
```

Clause `refocus4_node (STUB, t2, C)`:

```
refocus4_node (STUB, t2, C)
= (* by unfolding the call to refocus4_node *)
iterate4 (DEC (PR_LEFT_STUB t2, C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (t2, C)
```

Clause `refocus4_node (NODE (t11, t12), t2, C)`:

```
refocus4_node (NODE (t11, t12), t2, C)
= (* by unfolding the call to refocus4_node *)
iterate4 (DEC (PR_ASSOC (t11, t12, t2), C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (NODE (t11, NODE (t12, t2)), C)
= (* by unfolding the call to refocus4_tree *)
refocus4_node (t11, NODE (t12, t2), C)
```

Clause `refocus4_context (CTX_MT, (t', t_nf))`:

```
refocus4_context (CTX_MT, (t', t_nf))
= (* by unfolding the call to refocus4_context *)
iterate4 (VAL (t', t_nf))
= (* by unfolding the call to iterate4 *)
RESULT t_nf
```

There are two corollaries to the compressions above:

Dead clauses: All of the calls to `iterate4` have been unfolded, and therefore the definition of `iterate4` is dead.

Dead component: The term component of the values is now dead. We eliminate it in Section 11.5.

11.5 Renaming transition functions and flattening configurations

The resulting simplified machine is an 'eval/apply/continue' abstract machine. We therefore rename `refocus4_tree` to `flatten5`, `refocus4_node` to `flatten5_node`, and `refocus4_context` to `continue5`. The result reads as follows:

```
(*  flatten5 : tree * context -> result  *)
fun flatten5 (STUB, C)
    = continue5 (C, STUB_nf)
  | flatten5 (LEAF n, C)
    = continue5 (C, NODE_nf (n, STUB_nf))
  | flatten5 (NODE (t1, t2), C)
    = flatten5_node (t1, t2, C)

(*  flatten5_node : tree * tree * context -> result  *)
and flatten5_node (STUB, t2, C)
    = flatten5 (t2, C)
  | flatten5_node (LEAF n, t2, C)
    = flatten5 (t2, CTX_RIGHT (n, C))
  | flatten5_node (NODE (t11, t12), t2, C)
    = flatten5_node (t11, NODE (t12, t2), C)

(*  continue5 : context * tree_nf -> result  *)
and continue5 (CTX_MT, t_nf)
    = RESULT t_nf
  | continue5 (CTX_RIGHT (n, C), t_nf)
    = continue5 (C, NODE_nf (n, t_nf))

(*  normalize5 : tree -> result  *)
fun normalize5 t
    = flatten5 (t, CTX_MT)
```

11.6 Refunctionalization

The definitions of Section 11.5 are in defunctionalized form. The reduction contexts, together with `continue5`, are the first-order counterpart of a function. The higher-order counterpart of this abstract machine reads as follows:

```
(*  flatten6 : tree * (tree_nf -> 'a) -> 'a  *)
fun flatten6 (STUB, k)
    = k STUB_nf
  | flatten6 (LEAF n, k)
    = k (NODE_nf (n, STUB_nf))
  | flatten6 (NODE (t1, t2), k)
    = flatten6_node (t1, t2, k)
```

```
(*  flatten6_node : tree * tree * (tree_nf -> 'a) -> 'a  *)
and flatten6_node (STUB, t2, k)
    = flatten6 (t2, k)
  | flatten6_node (LEAF n, t2, k)
    = flatten6 (t2, fn t2_nf => k (NODE_nf (n, t2_nf)))
  | flatten6_node (NODE (t11, t12), t2, k)
    = flatten6_node (t11, NODE (t12, t2), k)

(*  normalize6 : tree -> result  *)
fun normalize6 t
    = flatten6 (t, fn t_nf => RESULT t_nf)
```

The resulting refunctionalized program is a familiar eval/apply evaluation function in CPS.

11.7 Back to direct style

The refunctionalized definition of Section 11.6 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:

```
(*  flatten7 : tree -> tree_nf  *)
fun flatten7 STUB
    = STUB_nf
  | flatten7 (LEAF n)
    = NODE_nf (n, STUB_nf)
  | flatten7 (NODE (t1, t2))
    = flatten7_node (t1, t2)

(*  flatten7_node : tree * tree -> tree_nf  *)
and flatten7_node (STUB, t2)
    = flatten7 t2
  | flatten7_node (LEAF n, t2)
    = NODE_nf (n, flatten7 t2)
  | flatten7_node (NODE (t11, t12), t2)
    = flatten7_node (t11, NODE (t12, t2))

(*  normalize7 : tree -> result  *)
fun normalize7 t
    = RESULT (flatten7 t)
```

The resulting definition is that of an traditional flatten function that iteratively flattens the current left subtree before recursively descending on the current right subtree.

11.8 Closure unconversion

This section is intentionally left blank, since the tree leaves are integers.

11.9 Summary

We have refocused the reduction-based normalization function of Section 10 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions. Most of the members of this family correspond to something one usually writes by hand.

11.10 Exercises

Exercise 59 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 54 in Section 10.7. At each step of the derivation, run the tests of Exercise 53 in Section 10.7.*

Exercise 60 *Would it make sense, in the definition of* `normalize6`*, to take* `fn v => v` *as the initial continuation? If so, what would be the definition of* `normalize7` *and what would be its type? What about* `normalize7'`*?*

12 A reduction semantics for flattening binary trees inside out

The goal of this section is to define a one-step flattening function over binary trees, using a right-most innermost strategy, and to construct the corresponding reduction-based flattening function.

To define a reduction semantics for binary trees, we specify their abstract syntax (Section 12.1, which is identical to Section 10.1), a notion of contraction (Section 12.2), and the right-most innermost reduction strategy (Section 12.3). We then define a one-step reduction function that decomposes a tree which is not in normal form into a redex and a reduction context, contracts the redex, and recomposes the context with the contractum (Section 12.4). 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 (Section 12.5).

12.1 Abstract syntax: terms and values

This section is is identical to Section 10.1.

12.2 Notion of contraction

We orient the conversion rules into contraction rules as in Section 10.2. To reflect the inside-out reduction strategy, we represent redexes as another data type:

```
datatype potential_redex = PR_LEFT_STUB of value
                         | PR_LEAF of int
                         | PR_ASSOC of tree * tree * value
```

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

(*  contract : potential_redex -> contractum_or_error  *)
fun contract (PR_LEFT_STUB (t, t_nf))
    = CONTRACTUM t
  | contract (PR_LEAF n)
    = CONTRACTUM (NODE (LEAF n, STUB))
  | contract (PR_ASSOC (t11, t12, (t2, t2_nf)))
    = CONTRACTUM (NODE (t11, NODE (t12, t2)))
```

12.3 Reduction strategy

We seek the right-most inner-most redex in a tree.

Reduction contexts: The grammar of reduction contexts reads as follows:

```
datatype context = CTX_MT
                 | CTX_RIGHT of tree * context
```

Decomposition: A tree is in normal form (i.e., it does not contain any potential redex) or it can be decomposed into a potential redex and a reduction context:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * context
```

The decomposition function recursively searches for the right-most inner-most redex in a term. As always, we define it as a big-step abstract machine. This abstract machine has three auxiliary functions, `decompose_tree`, `decompose_node`, and `decompose_context` between three states – a term and a context, two sub-terms and a context, and a context and a value.

- `decompose_tree` dispatches over the given tree;
- `decompose_node` dispatches over the left sub-tree of a given tree;
- `decompose_context` dispatches on the accumulated context to determine whether the given term is a value, a potential redex has been found, or the search must continue.

```
(*  decompose_tree : tree * context -> value_or_decomposition  *)
fun decompose_tree (STUB, C)
    = decompose_context (C, (STUB, STUB_nf))
  | decompose_tree (LEAF n, C)
    = DEC (PR_LEAF n, C)
  | decompose_tree (NODE (t1, t2), C)
    = decompose_tree (t2, CTX_RIGHT (t1, C))
```

```
(*  decompose_node : tree * value * context -> value_or_decomposition  *)
and decompose_node (STUB, v2, C)
    = DEC (PR_LEFT_STUB v2, C)
  | decompose_node (LEAF n, (t2, t2_nf), C)
    = decompose_context (C, (NODE (LEAF n, t2), NODE_nf (n, t2_nf)))
  | decompose_node (NODE (t11, t12), v2, C)
    = DEC (PR_ASSOC (t11, t12, v2), C)

(*  decompose_context : context * value -> value_or_decomposition  *)
and decompose_context (CTX_MT, (t', t_nf))
    = VAL (t', t_nf)
  | decompose_context (CTX_RIGHT (t1, C), (t2', t2_nf))
    = decompose_node (t1, (t2', t2_nf), C)

(*  decompose : tree -> value_or_decomposition  *)
fun decompose t
    = decompose_tree (t, CTX_MT)
```

Recomposition: The recomposition function peels off context layers and constructs the resulting tree, iteratively:

```
fun recompose (CTX_MT, t)
    = t
  | recompose (CTX_RIGHT (t1, C), t2)
    = recompose (C, NODE (t1, t2))
```

Lemma 6 *A tree* `t` *is either in normal form or there exists a unique context* `C` *such that* `decompose t` *evaluates to* `DEC (pr, C)`*, where* `pr` *is a potential redex.*

Proof 6 *Straightforward (see Exercise 66 in Section 12.7).*

12.4 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a tree that is not in normal form 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. The following data type accounts for whether the contraction is successful or the non-value term is stuck:

```
datatype reduct = REDUCT of tree
                | STUCK of string

(*  reduce : tree -> reduct  *)
fun reduce t
    = (case decompose t
```

```
          of (VAL (t', t_nf))
             => REDUCT t'
           | (DEC (pr, C))
             => (case contract pr
                   of (CONTRACTUM t')
                      => REDUCT (recompose (C, t'))
                    | (ERROR s)
                      => STUCK s))
```

12.5 Reduction-based normalization

The following reduction-based normalization function iterates the one-step reduction function until it yields a normal form:

```
datatype result = RESULT of tree_nf
                | WRONG of string

(*  iterate0 : value_or_decomposition -> result  *)
fun iterate0 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate0 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate0 (decompose (recompose (C, t')))
          | (ERROR s)
            => WRONG s)

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

12.6 Summary

We have implemented a reduction semantics for flattening binary trees, in complete detail. Using this reduction semantics, we have presented a reduction-based normalization function.

12.7 Exercises

Exercise 61 *Define a function* `embed_potential_redex_in_tree` *that maps a potential redex into a tree. (This exercise is the same as Exercise 51.)*

Exercise 62 *Show that, for any tree* `t`*, if evaluating* `decompose t` *yields* `DEC (pr, C)`*, then evaluating* `recompose (C, embed_potential_redex_in_tree pr)` *yields* `t`*.*
(Hint: Reason by structural induction over `t`*, using inversion at each step.)*

Exercise 63 *Write a handful of test trees and specify the expected outcome of their normalization. (This exercise is the same as Exercise 53.)*

Exercise 64 *Implement the reduction semantics above in the programming language of your choice, and run the tests of Exercise 63.*

Exercise 65 *Write an unparser from trees to the concrete syntax of your choice, as in Exercise 55, and instrument the normalization function of Section 12.5 so that (one way or another) it displays the successive trees in the reduction sequence.*

Exercise 66 *In the proof of Lemma 6, do as in the proof of Lemma 1 and write the refunctionalized counterpart of* `decompose` *et al.*

Exercise 67 *Pick another notion of normal form (e.g., flat, list-like trees on the left instead of on the right) and define the corresponding reduction-based normalization function,* mutatis mutandis.

13 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 12.5 into a family of reduction-free normalization functions, i.e., one where no intermediate tree is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate trees, and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 13.1). After inlining the contraction function (Section 13.2), we transform this small-step abstract machine into a big-step one (Section 13.3). This abstract machine exhibits a number of corridor transitions, and we compress them (Section 13.4). We then flatten its configurations and rename its transition functions into something more intuitive (Section 13.5). The resulting abstract machine is in defunctionalized form, and we refunctionalize it (Section 13.6). The result is in continuation-passing style and we re-express it in direct style (Section 13.7). The resulting direct-style function is a traditional flatten function with an accumulator; in particular, it is compositional and reduction-free.

Modus operandi: In each of the following subsections, and as always, we derive successive versions of the normalization function, indexing its components with the number of the subsection. In practice, the reader should run the tests of Exercise 63 in Section 12.7 at each step of the derivation, for sanity value.

13.1 Refocusing: from reduction-based to reduction-free normalization

The normalization function of Section 12.5 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`:

```
(*  normalize0' : tree -> result  *)
fun normalize0' t
    = iterate0 (decompose (recompose (CTX_MT, t)))
```

Refocusing, extensionally: The composition of `decompose` and `recompose` can be deforested into a '`refocus`' function to avoid constructing the intermediate terms in the reduction sequence. Such a deforestation makes the normalization function reduction-free.

Refocusing, intensionally: As usual, the `refocus` function can be expressed very simply in terms of the decomposition functions of Section 12.3:

```
(*  refocus : term * context -> value_or_decomposition  *)
fun refocus (t, C)
    = decompose_tree (t, C)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate1 : value_or_decomposition -> result  *)
fun iterate1 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate1 (DEC (pr, C))
    = (case contract pr
         of (CONTRACTUM t')
            => iterate1 (refocus (t', C))
          | (ERROR s)
            => WRONG s)

(*  normalize1 : tree -> result  *)
fun normalize1 t
    = iterate1 (refocus (t, CTX_MT))
```

This 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 contractum and a reduction context to the next redex and reduction context, if there are any in the reduction sequence.

13.2 Inlining the contraction function

We first inline 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`:

```
(*  iterate2 : value_or_decomposition -> result  *)
fun iterate2 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate2 (DEC (PR_LEFT_STUB (t, t_nf), C))
    = iterate2 (refocus (t, C))
  | iterate2 (DEC (PR_LEAF n, C))
    = iterate2 (refocus (NODE (LEAF n, STUB), C))
  | iterate2 (DEC (PR_ASSOC (t11, t12, (t2, t2_nf)), C))
    = iterate2 (refocus (NODE (t11, NODE (t12, t2)), C))

(*  normalize2 : tree -> result  *)
fun normalize2 t
    = iterate2 (refocus (t, CTX_MT))
```

We are now ready to fuse the composition of `iterate2` with `refocus` (shaded just above).

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

The refocused normalization function is a small-step abstract machine in the sense that `refocus` (i.e., `decompose_tree`, `decompose_node`, and `decompose_context`) acts as a transition function and `iterate1` as a driver loop that keeps activating `refocus` until a value is obtained. We fuse `iterate2` and `refocus` (i.e., `decompose_tree`, `decompose_node`, and `decompose_context`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_tree`, `decompose_node`, and `decompose_context`. The result is a big-step abstract machine consisting of four (mutually tail-recursive) state-transition functions:

- `refocus3_tree` is the composition of `iterate2` and `decompose_tree` and a clone of `decompose_tree` that directly calls `iterate3` over a leaf instead of returning it to `iterate2` as `decompose_tree` did;

- `refocus3_context` is the composition of `iterate2` and `decompose_context` that directly calls `iterate3` over a value or a decomposition instead of returning it to `iterate2` as `decompose_context` did;

- `refocus3_node` is the composition of `iterate2` and `decompose_node` and a clone of `decompose_node` that directly calls `iterate3` over a decomposition instead of returning it to `iterate2` as `decompose_node` did;

- `iterate3` is a clone of `iterate2` that calls the fused function `refocus3_tree`.

```
(*  refocus3_tree : tree * context -> result  *)
fun refocus3_tree (STUB, C)
    = refocus3_context (C, (STUB, STUB_nf))
```

```
  | refocus3_tree (LEAF n, C)
    = iterate3 (DEC (PR_LEAF n, C))
  | refocus3_tree (NODE (t1, t2), C)
    = refocus3_tree (t2, CTX_RIGHT (t1, C))

(*  refocus3_node : tree * value * context -> result  *)
and refocus3_node (STUB, v2, C)
    = iterate3 (DEC (PR_LEFT_STUB v2, C))
  | refocus3_node (LEAF n, (t2, t2_nf), C)
    = refocus3_context (C, (NODE (LEAF n, t2), NODE_nf (n, t2_nf)))
  | refocus3_node (NODE (t11, t12), v2, C)
    = iterate3 (DEC (PR_ASSOC (t11, t12, v2), C))

(*  refocus3_context : context * value -> result  *)
and refocus3_context (CTX_MT, (t', t_nf))
    = iterate3 (VAL (t', t_nf))
  | refocus3_context (CTX_RIGHT (t1, C), (t2', t2_nf))
    = refocus3_node (t1, (t2', t2_nf), C)

(*  iterate3 : value_or_decomposition -> result  *)
and iterate3 (VAL (t', t_nf))
    = RESULT t_nf
  | iterate3 (DEC (PR_LEFT_STUB (t, t_nf), C))
    = refocus3_tree (t, C)
  | iterate3 (DEC (PR_LEAF n, C))
    = refocus3_tree (NODE (LEAF n, STUB), C)
  | iterate3 (DEC (PR_ASSOC (t11, t12, (t2, t2_nf)), C))
    = refocus3_tree (NODE (t11, NODE (t12, t2)), C)

(*  normalize3 : tree -> result  *)
fun normalize3 t
    = refocus3_tree (t, CTX_MT)
```

This abstract machine is staged since `iterate3` implements the contraction rules of the reduction semantics separately from its congruence rules, which are implemented by `refocus3_tree`, `refocus3_context` and `refocus3_node`.

13.4 Compressing corridor transitions

In the abstract machine above, many of the transitions are 'corridor' ones in that they yield configurations for which there is a unique further transition, and so on. Let us compress these transitions. To this end, we cut-and-paste the transition functions above, renaming their indices from `3` to `4`, and consider each of their clauses in turn , making use of the equivalence between `refocus4_tree (t, C)` and `refocus4_context (C, t_nf)` when `t` is in normal form (and `t_nf` directly represents this normal form):

Clause `refocus4_tree (LEAF n, C)`:

```
refocus4_tree (LEAF n, C)
= (* by unfolding the call to refocus4_tree *)
iterate4 (DEC (PR_LEAF n, C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (NODE (LEAF n, STUB), C)
= (* by unfolding the call to refocus4_tree *)
refocus4_tree (STUB, CTX_RIGHT (LEAF n, C))
= (* by unfolding the call to refocus4_tree *)
refocus4_context (CTX_RIGHT (LEAF n, C), (STUB, STUB_nf))
= (* by unfolding the call to refocus4_context *)
refocus4_node (LEAF n, (STUB, STUB_nf), C)
= (* by unfolding the call to refocus4_node *)
refocus4_context (C, (NODE (LEAF n, STUB), NODE_nf (n, STUB_nf)))
```

Clause `refocus4_node (STUB, (t2, t2_nf), C)`:

```
refocus4_node (STUB, (t2, t2_nf), C)
= (* by unfolding the call to refocus4_node *)
iterate4 (DEC (PR_LEFT_STUB (t2, t2_nf), C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (t2, C)
= (* since t2 is in normal form *)
refocus4_context (C, (t2, t2_nf))
```

Clause `refocus4_node (NODE (t11, t12), (t2, t2_nf), C)`:

```
refocus4_node (NODE (t11, t12), (t2, t2_nf), C)
= (* by unfolding the call to refocus4_node *)
iterate4 (DEC (PR_ASSOC (t11, t12, (t2, t2_nf)), C))
= (* by unfolding the call to iterate4 *)
refocus4_tree (NODE (t11, NODE (t12, t2)), C)
= (* by unfolding the call to refocus4_tree *)
refocus4_tree (NODE (t12, t2), CTX_RIGHT (t11, C))
= (* by unfolding the call to refocus4_tree *)
refocus4_tree (t2, CTX_RIGHT (t12, CTX_RIGHT (t11, C)))
= (* since t2 is in normal form *)
refocus4_context (CTX_RIGHT (t12, CTX_RIGHT (t11, C)), (t2, t2_nf))
= (* by unfolding the call to refocus4_context *)
refocus4_node (t12, (t2, t2_nf), CTX_RIGHT (t11, C))
```

There are two corollaries to the compressions above:

Dead clauses: All of the calls to `iterate4` have been unfolded, and therefore the definition of `iterate4` is dead.

Dead component: The term component of the values is now dead. We eliminate it in Section 13.5.

13.5 Renaming transition functions and flattening configurations

The resulting simplified machine is an 'eval/apply/continue' abstract machine. We therefore rename `refocus4_tree` to `flatten5`, `refocus4_node` to `flatten5_node`, and `refocus4_context` to `continue5`. The result reads as follows:

```
(*  flatten5 : tree * context -> result  *)
fun flatten5 (STUB, C)
    = continue5 (C, STUB_nf)
  | flatten5 (LEAF n, C)
    = continue5 (C, NODE_nf (n, STUB_nf))
  | flatten5 (NODE (t1, t2), C)
    = flatten5 (t2, CTX_RIGHT (t1, C))

(*  flatten5_node : tree * tree_nf * context -> result  *)
and flatten5_node (STUB, t2_nf, C)
    = continue5 (C, t2_nf)
  | flatten5_node (LEAF n, t2_nf, C)
    = continue5 (C, NODE_nf (n, t2_nf))
  | flatten5_node (NODE (t11, t12), t2_nf, C)
    = flatten5_node (t12, t2_nf, CTX_RIGHT (t11, C))

(*  continue5 : context * tree_nf -> result  *)
and continue5 (CTX_MT, t_nf)
    = RESULT t_nf
  | continue5 (CTX_RIGHT (t1, C), t2_nf)
    = flatten5_node (t1, t2_nf, C)

(*  normalize5 : tree -> result  *)
fun normalize5 t
    = flatten5 (t, CTX_MT)
```

13.6 Refunctionalization

The definitions of Section 13.5 are in defunctionalized form. The reduction contexts, together with `continue5`, are the first-order counterpart of a function. The higher-order counterpart of this abstract machine reads as follows:

```
(*  flatten6 : tree * (tree_nf -> 'a) -> 'a  *)
fun flatten6 (STUB, k)
    = k STUB_nf
  | flatten6 (LEAF n, k)
    = k (NODE_nf (n, STUB_nf))
  | flatten6 (NODE (t1, t2), k)
    = flatten6 (t2, fn t2_nf => flatten6_node (t1, t2_nf, k))
```

```
(*  flatten6_node : tree * tree_nf * (tree_nf -> 'a) -> 'a  *)
and flatten6_node (STUB, t2_nf, k)
    = k t2_nf
  | flatten6_node (LEAF n, t2_nf, k)
    = k (NODE_nf (n, t2_nf))
  | flatten6_node (NODE (t11, t12), t2_nf, k)
    = flatten6_node (t12, t2_nf, fn t2_nf => flatten6_node (t11, t2_nf, k))

(*  normalize6 : tree -> result  *)
fun normalize6 t
    = flatten6 (t, fn t_nf => RESULT t_nf)
```

The resulting refunctionalized program is a familiar eval/apply evaluation function in CPS.

13.7 Back to direct style

The refunctionalized definition of Section 13.6 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:

```
(*  flatten7 : tree -> tree_nf  *)
fun flatten7 STUB
    = STUB_nf
  | flatten7 (LEAF n)
    = NODE_nf (n, STUB_nf)
  | flatten7 (NODE (t1, t2))
    = flatten7_node (t1, flatten7 t2)

(*  flatten7_node : tree * tree_nf -> tree_nf  *)
and flatten7_node (STUB, t2_nf)
    = t2_nf
  | flatten7_node (LEAF n, t2_nf)
    = NODE_nf (n, t2_nf)
  | flatten7_node (NODE (t11, t12), t2_nf)
    = flatten7_node (t11, flatten7_node (t12, t2_nf))

(*  normalize7 : tree -> result  *)
fun normalize7 t
    = RESULT (flatten7 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 [9, 7, 11, 51]. It also coincides with the definition of the flatten function in Yves Bertot's concise presentation of the Coq proof assistant [8, Section 4.8].

13.8 Closure unconversion

This section is intentionally left blank, since the tree leaves are integers.

13.9 Summary

We have refocused the reduction-based normalization function of Section 12 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions. Most of the members of this family correspond to something one usually writes by hand.

13.10 Exercises

Exercise 68 *Reproduce the construction above in the programming language of your choice, starting from your solution to Exercise 64 in Section 12.7. At each step of the derivation, run the tests of Exercise 63 in Section 12.7.*

Exercise 69 *Would it make sense, in the definition of* `normalize6`*, to take* `fn v => v` *as the initial continuation? If so, what would be the definition of* `normalize7` *and what would be its type? What about* `normalize7'`*?*

Exercise 70 *In Section 13.7, the reduction-free normalization function could be streamlined by skipping* `flatten7` *as follows:*

```
(*  normalize7' : tree -> result  *)
fun normalize7' t
    = RESULT (flatten7_node (t, STUB_nf))
```

This streamlined 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. Which reduction semantics gives rise to this streamlined flatten function?

14 Conclusion

In Jean-Jacques Beineix's movie "Diva," Gorodish shows Postman Jules the Zen aspects of buttering a French baguette. He starts from a small-step description of the baguette that is about as fetching as the one in the more recent movie "Ratatouille" and progressively detaches himself from the bread, the butter and the knife to culminate with a movement, a gesture, big steps. So is it for reduction-free normalization compared to reduction-based normalization: we start from an abstract syntax and a reduction strategy where everything is explicit, and we end up skipping the reduction sequence altogether and reaching a state where everything is implicit, expressed that it is in the meta-language, as in Per Martin Löf's original vision of normalization by evaluation [28, 55]. It is the author's hope that the reader is now in position to butter a French baguette at home with harmony and efficiency, computationally speaking, that is: whether, e.g., calculating an arithmetic expression, recognizing a Dyck word, normalizing a lambda-term with explicit substitutions and possibly call/cc, or flattening a binary tree, one can either use small steps and adopt a notion of reduction and a reduction strategy, or use big steps and adopt a notion of evaluation and an evaluation strategy. Plotkin, 30 years ago [64], extensionally connected the two by showing that for the lambda-calculus, applicative order (resp. normal order) corresponds to call by value (resp. call by name). In these lecture notes, we have shown that this extensional connection also makes sense intensionally: small-step implementations and big-step implementations can be mechanically inter-derived; it is the same elephant.

Acknowledgments: These lecture notes are a revised and substantially expanded version of an invited talk at WRS 2004 [21], for which the author is still grateful to Sergio Antoy and Yoshihito Toyama. Thanks are also due to Rinus Plasmeijer for the opportunity to present this material at AFP 2008; to the other organizers and co-lecturers for a wonderful event; to Alain Crémieux, Diana Fulger and the other AFP 2008 attendees for their interaction and feedback; to Pieter Koopman for his editorship; to Jacob Johannsen, Ian Zerny and the anonymous reviewers and editors for their comments; and to Sivert Bertelsen, Sebastian Erdweg, Alexander Hansen, Dennis Decker Jensen, Finn Rosenbech Jensen and Tillmann Rendel for their extra comments in the spring of 2009.

—

The goal of the following appendices is to review closure conversion, CPS transformation, defunctionalization, lightweight fission, and lightweight fusion. To this end, we retrace John Reynolds's steps from a compositional evaluation function to an abstract machine [67] and then move on to lightweight fission and fusion.

A Lambda-terms with integers

We first specify lambda-terms with integers (arbitrary literals and a predefined successor function) and then present a computationally representative sample of lambda-terms.

A.1 Abstract syntax

A lambda-term is an integer literal, an identifier, a lambda-abstraction or an application:

```
datatype term = LIT of int
              | IDE of string
              | LAM of string * term
              | APP of term * term
```

We assume predefined identifiers, e.g., "`succ`" to denote the successor function.

A.2 A sample of lambda-terms

Church numerals [17] and mappings between native natural numbers and Church numerals form a good ground to illustrate the expressive power of lambda-terms with integers.

Church numerals. A Church numeral is a functional encoding of a natural number that abstracts a zero value and a successor function:

```
val cn0 = LAM ("s", LAM ("z", IDE "z"))
val cns = LAM ("cn",
               LAM ("s", LAM ("z", APP (APP (IDE "cn", IDE "s"),
                                        APP (IDE "s", IDE "z")))))
```

For example, here is the Church numeral representing the natural number 3:

```
val cn3 = APP (cns, APP (cns, APP (cns, cn0)))
```

Mappings between natural numbers and Church numerals. Given a natural number n, one constructs the corresponding Church numeral by recursively applying `cns` n times to `cn0`. Conversely, applying a Church numeral that represents the natural number n to the native successor function and the native natural number 0 yields a term that reduces to the native representation of n.

```
fun n2cn 0 = cn0
  | n2cn n = APP (cns, n2cn (n - 1))

fun cn2n cn
    = APP (APP (cn, IDE "succ"), LIT 0)
```

Computing with Church numerals. As is well known, applying a Church numeral to another one implements exponentiation. The following term therefore reduces to the native representation of 1024:

```
val n1024 = cn2n (APP (n2cn 10, n2cn 2))
```

B A call-by-value evaluation function

Let us write a canonical evaluator for lambda-terms with integers as specified in Section A. The evaluator uses an environment, and proceeds by recursive descent over a given term. It is compositional.

Environments. The environment is a canonical association list (i.e., list of pairs associating identifiers and values):

```
structure Env
= struct
    type 'a env = (string * 'a) list

    val empty = []

    fun extend (x, v, env)
        = (x, v) :: env

    fun lookup (x, env)
        = let fun search []
                    = NONE
                | search ((x', v) :: env)
                    = if x = x' then SOME v else search env
          in search env
          end
  end
```

Values. Values are integers or functions:

```
datatype value = VAL_INT of int
               | VAL_FUN of value -> value
```

Evaluation function. The evaluation function is a traditional, Scott-Tarski one. (Scott because of the reflexive data type of values, and Tarski because of its meta-circular fashion of interpreting a concept in term of the same concept at the meta-level: syntactic lambda-abstractions are interpreted in terms of ML function abstractions, and syntactic applications in terms of ML function applications.) Evaluating a program might go wrong because an undeclared identifier is used, because the successor function is applied to a non-integer, or because a non-function is applied; these events are summarily interpreted by raising an exception to the top level.

```
exception WRONG of string
```

```
(*  eval0 : term * value Env.env -> value  *)
fun eval0 (LIT n, e)
    = VAL_INT n
  | eval0 (IDE x, e)
    = (case Env.lookup (x, e)
         of NONE
            => raise (WRONG "undeclared identifier")
          | (SOME v)
            => v)
  | eval0 (LAM (x, t), e)
    = VAL_FUN (fn v => eval0 (t, Env.extend (x, v, e)))
  | eval0 (APP (t0, t1), e)
    = apply0 (eval0 (t0, e), eval0 (t1, e))

(*  apply0 : value * value -> value  *)
and apply0 (VAL_FUN f, v)
    = f v
  | apply0 (v0, v1)
    = raise (WRONG "non-applicable value")
```

Initial environment. The initial environment binds, e.g., the identifier `succ` to the successor function:

```
val val_succ = VAL_FUN (fn (VAL_INT n)
                           => VAL_INT (n + 1)
                         | v
                           => raise (WRONG "non-integer value"))

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

Main function. A term is interpreted by evaluating it in the initial environment in the presence of an exception handler. Evaluating a term may diverge; otherwise it either yields a value or an error message if evaluation goes wrong:

```
datatype value_or_error = VALUE of value
                        | ERROR of string

(*  interpret0 : term -> value_or_error  *)
fun interpret0 t
    = VALUE (eval0 (t, e_init))
      handle (WRONG s) => ERROR s
```

C Closure conversion

Let us "firstify" the domain of values by defunctionalizing it: the function space, in the data type of values in Appendix B, is inhabited by function values that arise from evaluating two (and only two) function abstractions: one in the `LAM` clause in the definition of `eval0` as the denotation of a syntactic lambda-abstraction, and one in the initial environment as the successor function. We therefore modify the domain of values by replacing the higher-order constructor `VAL_FUN` by two first-order constructors `VAL_SUCC` and `VAL_CLO`:

```
datatype value = VAL_INT of int
               | VAL_SUCC
               | VAL_CLO of string * term * value Env.env
```

The first-order representation tagged by `VAL_CLO` is known as a "closure" since Landin's pioneering work [53]: it pairs a lambda-abstraction and its environment of declaration.

Introduction: `VAL_SUCC` is produced in the initial environment as the denotation of `succ`; and `VAL_CLO` is produced in the `LAM` clause and holds the free variables of `fn v => eval0 (t, Env.extend (x, v, e))`.

Elimination: `VAL_SUCC` and `VAL_CLO` are consumed in new clauses of the `apply` function, which dispatches over applicable values. As in Appendix B, applying `VAL_SUCC` to an integer yields the successor of this integer and applying it to a non-integer raises an exception; and applying `VAL_CLO (x, t, e)`, i.e., the result of evaluating `LAM (x, t)` in an environment `e`, to a value `v` leads `t` to be evaluated in an extended environment, as in Appendix B.

Compared to Appendix B, the new parts of the following closure-converted interpreter are shaded:

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

(*  eval1 : term * value Env.env -> value  *)
fun eval1 (LIT n, e)
    = VAL_INT n
  | eval1 (IDE x, e)
    = (case Env.lookup (x, e)
         of NONE
            => raise (WRONG "undeclared identifier")
          | (SOME v)
            => v)
  | eval1 (LAM (x, t), e)
    = VAL_CLO (x, t, e)
```

```
  | eval1 (APP (t0, t1), e)
    = apply1 (eval1 (t0, e), eval1 (t1, e))
(*  apply1 : value * value -> value  *)
and apply1 (VAL_SUCC, VAL_INT n)
    = VAL_INT (n + 1)
  | apply1 (VAL_SUCC, v)
    = raise (WRONG "non-integer value")
  | apply1 (VAL_CLO (x, t, e), v)
    = eval1 (t, Env.extend (x, v, e))
  | apply1 (v0, v1)
    = raise (WRONG "non-applicable value")

datatype value_or_error = VALUE of value
                        | ERROR of string

(*  interpret1 : term -> value_or_error  *)
fun interpret1 t
    = VALUE (eval1 (t, e_init))
      handle (WRONG s) => ERROR s
```

The resulting interpreter is a traditional McCarthy-Landin one. (McCarthy because of his original definition of Lisp in Lisp [56] and Landin because of the closures.) It can also be seen as an implementation of Kahn's natural semantics [49].

D CPS transformation

Let us transform `eval1` and `apply1`, in Appendix C, into continuation-passing style (CPS). To this end, we name each of their intermediate results, we sequentialize their computation, and we pass them an extra (functional) parameter, the continuation. As a result, the intermediate results are named by the formal parameter of each of the lambda-abstractions that define the continuation (shaded below):

```
(*  eval2 : term * value Env.env * (value -> value_or_error)
            -> value_or_error  *)
fun eval2 (LIT n, e, k)
    = k (VAL_INT n)
  | eval2 (IDE x, e, k)
    = (case Env.lookup (x, e)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
            => k v)
```

```
  | eval2 (LAM (x, t), e, k)
    = k (VAL_CLO (x, t, e))
  | eval2 (APP (t0, t1), e, k)
    = eval2 (t0, e, fn v0 =>
        eval2 (t1, e, fn v1 =>
          apply2 (v0, v1, k)))
(*  apply2 : value * value * (value -> value_or_error)
             -> value_or_error  *)
and apply2 (VAL_SUCC, VAL_INT n, k)
    = k (VAL_INT (n + 1))
  | apply2 (VAL_SUCC, v, k)
    = ERROR "non-integer value"
  | apply2 (VAL_CLO (x, t, e), v, k)
    = eval2 (t, Env.extend (x, v, e), k)
  | apply2 (v0, v1, k)
    = ERROR "non-applicable value"

(*  interpret2 : term -> value_or_error  *)
fun interpret2 t
    = eval2 (t, e_init, fn v => VALUE v)
```

The resulting interpreter is a traditional continuation-passing one, as can be found in Morris's early work [60], in Steele and Sussman's lambda-papers [71, 69], and in "Essentials of Programming Languages" [42].

E Defunctionalization

Let us defunctionalize the continuation of Appendix D's interpreter. This function space is inhabited by function values that arise from evaluating three (and only three) function abstractions—those whose formal parameter is shaded above. We therefore partition the function space into three summands and represent it as the following first-order data type:

```
datatype cont = CONT_MT
              | CONT_FUN of cont * term * value Env.env
              | CONT_ARG of value * cont
```

This first-order representation is known as that of an evaluation context [40].

Introduction: `CONT_MT` is produced in the initial call to `eval3`; `CONT_FUN` is produced in the recursive self-call in `eval3`; and `CONT_ARG` is produced in the function that dispatches upon the evaluation context, `continue3`. Each constructor holds the free variables of the function abstraction it represents.

Elimination: The three constructors are consumed in `continue3`.

Compared to Appendix D, the new parts of the following defunctionalized interpreter are shaded:

```
(*  eval3 : term * value Env.env * cont -> value_or_error  *)
fun eval3 (LIT n, e, C)
    = continue3 (C, VAL_INT n)
  | eval3 (IDE x, e, C)
    = (case Env.lookup (x, e)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
            => continue3 (C, v))
  | eval3 (LAM (x, t), e, C)
    = continue3 (C, VAL_CLO (x, t, e))
  | eval3 (APP (t0, t1), e, C)
    = eval3 (t0, e, CONT_FUN (C, t1, e))

(*  apply3 : value * value * cont -> value_or_error  *)
and apply3 (VAL_SUCC, VAL_INT n, C)
    = continue3 (C, VAL_INT (n + 1))
  | apply3 (VAL_SUCC, v, C)
    = ERROR "non-integer value"
  | apply3 (VAL_CLO (x, t, e), v, C)
    = eval3 (t, Env.extend (x, v, e), C)
  | apply3 (v0, v1, C)
    = ERROR "non-applicable value"

(*  continue3 : context * value -> value_or_error   *)
and continue3 (CONT_MT, v)
    = VALUE v
  | continue3 (CONT_FUN (C, t1, e), v0)
    = eval3 (t1, e, CONT_ARG (v0, C))
  | continue3 (CONT_ARG (v0, C), v1)
    = apply3 (v0, v1, C)

(*  interpret3 : term -> value_or_error  *)
fun interpret3 t
    = eval3 (t, e_init, CONT_MT )
```

Reynolds pointed at the "machine-like" qualities of this defunctionalized interpreter, and indeed the alert reader will already have recognized that this interpreter implements a big-step version of the CEK abstract machine [41]. Indeed each (tail-)call implements a state transition.

F Lightweight fission

Let us explicitly represent the states of the abstract machine of Appendix E with the following data type:

```
datatype state = STOP of value
               | WRONG of string
               | EVAL of term * value Env.env * cont
               | APPLY of value * value * cont
               | CONTINUE of cont * value
```

Non-accepting states: The STOP state marks that a value has been computed for the given term, and the WRONG state that the given term is a stuck one.

Accepting states: The EVAL, APPLY, and CONTINUE states mark that the machine is ready to take a transition corresponding to one (tail-)call in Appendix E, as respectively implemented by the following transition functions move_eval, move_apply, and move_continue.

```
(*  move_eval : term * value Env.env * cont -> state  *)
fun move_eval (LIT n, e, C)
    = CONTINUE (C, VAL_INT n)
  | move_eval (IDE x, e, C)
    = (case Env.lookup (x, e)
         of NONE
            => WRONG "undeclared identifier"
          | (SOME v)
            => CONTINUE (C, v))
  | move_eval (LAM (x, t), e, C)
    = CONTINUE (C, VAL_CLO (x, t, e))
  | move_eval (APP (t0, t1), e, C)
    = EVAL (t0, e, CONT_FUN (C, t1, e))

(*  move_apply : value * value * cont -> state  *)
fun move_apply (VAL_SUCC, VAL_INT n, C)
    = CONTINUE (C, VAL_INT (n + 1))
  | move_apply (VAL_SUCC, v, C)
    = WRONG "non-integer value"
  | move_apply (VAL_CLO (x, t, e), v, C)
    = EVAL (t, Env.extend (x, v, e), C)
  | move_apply (v0, v1, C)
    = WRONG "non-applicable value"

(*  move_continue : cont * value -> state  *)
fun move_continue (CONT_MT, v)
    = STOP v
  | move_continue (CONT_FUN (C, t1, e), v0)
    = EVAL (t1, e, CONT_ARG (v0, C))
  | move_continue (CONT_ARG (v0, C), v1)
    = APPLY (v0, v1, C)
```

The following driver loop maps a non-accepting state to a final result or (1) activates the transition corresponding to the current accepting state and (2) iterates:

```
(*  drive : state -> value_or_error  *)
fun drive (STOP v)
    = VALUE v
  | drive (WRONG s)
    = ERROR s
  | drive (EVAL c)
    = drive (move_eval c)
  | drive (APPLY c)
    = drive (move_apply c)
  | drive (CONTINUE c)
    = drive (move_continue c)
```

For a given term `t`, the initial state of machine is `EVAL (t, e_init, CONT_MT)`:

```
(*  interpret4 : term -> value_or_error  *)
fun interpret4 t
    = drive (EVAL (t, e_init, CONT_MT))
```

The resulting interpreter is a traditional small-step abstract machine [65], namely the CEK machine [41]. As spelled out in Appendix G, fusing the driver loop and the transition functions yields the big-step abstract machine of Appendix E.

G Lightweight fusion by fixed-point promotion

Let us review Ohori and Sasano's lightweight fusion by fixed-point promotion [63]. This calculational transformation operates over functional programs in the form of the small-step abstract machine of Appendix F: a (strict) top-level driver function `drive` activating (total) tail-recursive transition functions. The transformation consists in three steps:

1. Inline the definition of the transition function in the composition.
2. Distribute the tail call to the driver function in the conditional branches.
3. Simplify by inlining the applications of the driver function to known arguments.

One then uses the result of the third step to define new mutually recursive functions that are respectively equal to the compositions obtained in the third step.

Let us consider the following function compositions in turn:

- `fn g => drive (move_eval g)` in Appendix G.1;
- `fn g => drive (move_apply g)` in Appendix G.2; and
- `fn g => drive (move_continue g)` in Appendix G.3.

G.1 `drive o move_eval`

1. We inline the definition of `move_eval` in the composition:

```
fn g => drive (case g
                 of (LIT n, e, C)
                    => CONTINUE (C, VAL_INT n)
                  | (IDE x, e, C)
                    => (case Env.lookup (x, e)
                          of NONE
                             => WRONG "undeclared identifier"
                           | (SOME v)
                             => CONTINUE (C, v))
                  | (LAM (x, t), e, C)
                    => CONTINUE (C, VAL_CLO (x, t, e))
                  | (APP (t0, t1), e, C)
                    => EVAL (t0, e, CONT_FUN (C, t1, e)))
```

2. We distribute the tail call to `drive` in the conditional branches:

```
fn c => case c
          of (LIT n, e, C)
             => drive (CONTINUE (C, VAL_INT n))
           | (IDE x, e, C)
             => (case Env.lookup (x, e)
                   of NONE
                      => drive (WRONG "undeclared identifier")
                    | (SOME v)
                      => drive (CONTINUE (C, v)))
           | (LAM (x, t), e, C)
             => drive (CONTINUE (C, VAL_CLO (x, t, e)))
           | (APP (t0, t1), e, C)
             => drive (EVAL (t0, e, CONT_FUN (C, t1, e)))
```

   Or again, more concisely, with a function declared by cases:

```
fn (LIT n, e, C)
   => drive (CONTINUE (C, VAL_INT n))
 | (IDE x, e, C)
   => (case Env.lookup (x, e)
         of NONE
            => drive (WRONG "undeclared identifier")
          | (SOME v)
            => drive (CONTINUE (C, v)))
 | (LAM (x, t), e, C)
   => drive (CONTINUE (C, VAL_CLO (x, t, e)))
 | (APP (t0, t1), e, C)
   => drive (EVAL (t0, e, CONT_FUN (C, t1, e)))
```

3. We simplify by inlining the applications of `drive` to known arguments:

```
fn (LIT n, e, C)
   => drive (move_continue (C, VAL_INT n))
 | (IDE x, e, C)
   => (case Env.lookup (x, e)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
            => drive (move_continue (C, v)))
 | (LAM (x, t), e, C)
   => drive (move_continue (C, VAL_CLO (x, t, e)))
 | (APP (t0, t1), e, C)
   => drive (move_eval (t0, e, CONT_FUN (C, t1, e)))
```

G.2 `drive o move_apply`

1. We inline the definition of `move_apply` in the composition:

```
fn g => drive (case g
                 of (VAL_SUCC, VAL_INT n, C)
                    => CONTINUE (C, VAL_INT (n + 1))
                  | (VAL_SUCC, v, C)
                    => WRONG "non-integer value"
                  | (VAL_CLO (x, t, e), v, C)
                    => EVAL (t, Env.extend (x, v, e), C)
                  | (v0, v1, C)
                    => WRONG "non-applicable value")
```

2. We distribute the tail call to `drive` in the conditional branches:

```
fn (VAL_SUCC, VAL_INT n, C)
   => drive (CONTINUE (C, VAL_INT (n + 1)))
 | (VAL_SUCC, v, C)
   => drive (WRONG "non-integer value")
 | (VAL_CLO (x, t, e), v, C)
   => drive (EVAL (t, Env.extend (x, v, e), C))
 | (v0, v1, C)
   => drive (WRONG "non-applicable value")
```

3. We simplify by inlining the applications of `drive` to known arguments:

```
fn (VAL_SUCC, VAL_INT n, C)
   => drive (move_continue (C, VAL_INT (n + 1)))
 | (VAL_SUCC, v, C)
```

```
    => ERROR "non-integer value"
 | (VAL_CLO (x, t, e), v, C)
    => drive (move_eval (t, Env.extend (x, v, e), C))
 | (v0, v1, C)
    => ERROR "non-applicable value"
```

G.3 `drive o move_continue`

1. We inline the definition of `move_continue` in the composition:

```
fn g => drive (case g
                 of (CONT_MT, v)
                    => STOP v
                  | (CONT_FUN (C, t1, e), v0)
                    => EVAL (t1, e, CONT_ARG (v0, C))
                  | (CONT_ARG (v0, C), v1)
                    => APPLY (v0, v1, C))
```

2. We distribute the tail call to `drive` in the conditional branches:

```
fn (CONT_MT, v)
    => drive (STOP v)
 | (CONT_FUN (C, t1, e), v0)
    => drive (EVAL (t1, e, CONT_ARG (v0, C)))
 | (CONT_ARG (v0, C), v1)
    => drive (APPLY (v0, v1, C))
```

3. We simplify by inlining the applications of `drive` to known arguments:

```
fn (CONT_MT, v)
    => VALUE v
 | (CONT_FUN (C, t1, e), v0)
    => drive (move_eval (t1, e, CONT_ARG (v0, C)))
 | (CONT_ARG (v0, C), v1)
    => drive (move_apply (v0, v1, C))
```

G.4 Synthesis

We now use the result of the third steps above to define three new mutually recursive functions `drive_move_eval`, `drive_move_apply`, and `drive_move_continue` that are respectively equal to `drive o move_eval`, `drive o move_apply`, and `drive o move_continue`:

```
fun drive_move_eval (LIT n, e, C)
    = drive_move_continue (C, VAL_INT n)
  | drive_move_eval (IDE x, e, C)
    = (case Env.lookup (x, e)
         of NONE
            => ERROR "undeclared identifier"
          | (SOME v)
            => drive_move_continue (C, v))
  | drive_move_eval (LAM (x, t), e, C)
    = drive_move_continue (C, VAL_CLO (x, t, e))
  | drive_move_eval (APP (t0, t1), e, C)
    = drive_move_eval (t0, e, CONT_FUN (C, t1, e))

and drive_move_apply (VAL_SUCC, VAL_INT n, C)
    = drive_move_continue (C, VAL_INT (n + 1))
  | drive_move_apply (VAL_SUCC, v, C)
   = ERROR "non-integer value"
  | drive_move_apply (VAL_CLO (x, t, e), v, C)
    = drive_move_eval (t, Env.extend (x, v, e), C)
  | drive_move_apply (v0, v1, C)
    = ERROR "non-applicable value"

and drive_move_continue (CONT_MT, v)
   = VALUE v
  | drive_move_continue (CONT_FUN (C, t1, e), v0)
    = drive_move_eval (t1, e, CONT_ARG (v0, C))
  | drive_move_continue (CONT_ARG (v0, C), v1)
    = drive_move_apply (v0, v1, C)

fun interpret5 t
    = drive_move_eval (t, e_init, CONT_MT)
```

Except for the function names (`drive_move_eval` instead of `eval3`, `drive_move_apply` instead of `apply3`, and `drive_move_continue` instead of `continue3`), the fused definition coincides with the definition in Appendix E.

H Exercises

Exercise 71 *Implement all the interpreters of this appendix in the programming language of your choice, and verify that each of them maps* `n1024` *(defined in Appendix A.2) to* `VALUE (VAL_INT 1024)`.

Exercise 72 *In Appendices C and D, we closure-converted* and then *CPS-transformed the interpreter of Appendix B. Do the converse, i.e., CPS-transform the interpreter of Appendix B and then closure-convert it. The result should coincide with the interpreter of Appendix D. You will need the following data type of values:*

```
datatype value = VAL_INT of int
               | VAL_FUN of value * (value -> value_or_error)
                            -> value_or_error
```

Naturally, your continuation-passing interpreter should not use exceptions. Since it is purely functional and compositional, it can be seen as an implementation of a denotational semantics [72].

Exercise 73 *Fold the term and the environment, in either of the abstract machines of Appendix E or F, into the following data type of ground closures:*

```
datatype closure = CLO_GND of term * value Env.env
```

- *the type of the* `eval` *transition function should read*

  ```
  closure * cont -> value_or_error
  ```

- *the type of the* `move_eval` *transition function should read*

  ```
  closure * cont -> state
  ```

In either case, the resulting interpreter is a CK abstract machine [40], i.e., an environment-less machine that operates over ground closures. Conversely, unfolding these closures into a simple pair and flattening the resulting configurations mechanically yields either of the environment-based CEK machines of Appendix E or F.

I Mini project: call by name

Exercise 74 *Write a few lambda-terms that would make a call-by-value evaluation function and a call-by-name evaluation function not yield the same result.*

Exercise 75 *Modify the code of the evaluation function of Appendix B to make it call by name, using the following data type of values:*

```
datatype value = VAL_INT of int
               | VAL_FUN of thunk -> value
withtype thunk = unit -> value
```

Verify that the lambda-terms of Exercise 74 behave as expected.

Exercise 76 *In continuation of Exercise 75, closure-convert your call-by-name evaluation function, and verify that the lambda-terms of Exercise 74 behave as expected.*

Exercise 77 *In continuation of Exercise 76, CPS transform your closure-converted call-by-name evaluation function, and verify that the lambda-terms of Exercise 74 behave as expected.*

Exercise 78 *For the sake of comparison, CPS-transform* first *the call-by-name evaluation function from Exercise 75, using the optimized data type*

```
datatype value = VAL_INT of int
               | VAL_FUN of thunk * (value -> value_or_error)
                            -> value_or_error
withtype thunk = (value -> value_or_error) -> value_or_error
```

(`thunk` *would be* `unit * (value -> value_or_error) -> value_or_error` *in an unoptimized version), and then closure-convert it. Do you obtain the same result as in Exercise 77?*
(Hint: You should.)

Exercise 79 *Defunctionalize the closure-converted, CPS-transformed call-by-name evaluation function of Exercise 77, and compare the result with the Krivine machine [3, 25].*

Exercise 80 *Using the* call-by-name *CPS transformation [30, 64], CPS transform the evaluation function of Appendix C. Do you obtain the same result as in Exercise 77?*
(Hint: You should [45].)

Exercise 81 *Again, using the* call-by-name *CPS transformation [30, 64], CPS transform the evaluation function of Appendix B. Do you obtain the same interpreter as in Exercise 78 before closure conversion?*
(Hint: Again, you should [45].)

Exercise 82 *Start from the call-by-name counterpart of Section 6 and, through refocusing, move towards an abstract machine and compare this abstract machine with the Krivine machine.*
(Hint: See Section 3 of "A Concrete Framework for Environment Machines" [12].)

J Further projects

- The reader interested in other abstract machines is directed to "A Functional Correspondence between Evaluators and Abstract Machines [3].
- For a call-by-need counterpart of Section 6, the reader is directed to "A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines" [4] and to Section 7 of "A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines" [12].
- The reader interested in computational effects is directed to "A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects" [5] and "A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines" [12].

- The reader interested in the SECD machine is directed to "A Rational Deconstruction of Landin's SECD Machine" [23].
- The reader interested in the SECD machine and the J operator is directed to "A Rational Deconstruction of Landin's SECD Machine with the J Operator" [34].
- The reader interested in delimited continuations and the CPS hierarchy is directed to "An Operational Foundation for Delimited Continuations in the CPS Hierarchy" [11].
- The reader interested in Abadi and Cardelli's untyped calculus of objects [1] is directed to "Inter-deriving Semantic Artifacts for Object-Oriented Programming" [31], the extended version of which also features negational normalization for Boolean formulas.
- The reader interested in the semantics of the Scheme programming language is directed to Parts I and II of "Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language" [14, 27].

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. {**103**}

[2] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006. {**6, 103**}

[3] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press. {**4, 6, 18, 47, 102, 103**}

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3. {**4, 18, 47, 102, 103**}

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172,

2005. Extended version available as the research report BRICS RS-04-28. {**4, 6, 18, 47, 102, 103**}

[6] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–51, 2002. A preliminary version is available in the proceedings of SAS 1999 (LNCS 1694). {**64, 104**}

[7] 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, GPCE 2002*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag. {**85, 104**}

[8] Yves Bertot. Coq in a hurry. CoRR, May 2006. http://arxiv.org/abs/cs/0603118v2. {**85, 104**}

[9] 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. {**85, 104**}

[10] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006. {**6, 104**}

[11] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04). {**85, 103, 104**}

[12] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3. {**6, 35, 102, 104**}

[13] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18. {**6, 104**}

[14] Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language,

Part II: Reduction semantics and abstract machines. In Clinger [18]. {**103, 104**}

[15] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2005. {**6, 104**}

[16] 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. {**4, 18, 47, 105**}

[17] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. {**88, 105**}

[18] Will Clinger, editor. *2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Victoria, British Columbia, September 2008. {**104–106**}

[19] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991. {**35, 105**}

[20] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992). {**4, 5, 19, 33, 105**}

[21] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk. {**6, 87, 105**}

[22] 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 (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk. {**6, 105**}

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

2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33. {**4, 18, 47, 103, 105**}

[24] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006. {**6, 105**}

[25] Olivier Danvy, editor. *Special Issue on the Krivine Abstract Machine, part I*, volume 20, number 3 of *Higher-Order and Symbolic Computation*. Springer, 2007. {**102, 105**}

[26] Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, September 2008. ACM Press. Invited talk. {**6, 8, 106**}

[27] Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines. In Clinger [18]. {**103, 106**}

[28] 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, Aarhus University. Available online at `<http://www.brics.dk/~nbe98/programme.html>`. {**87, 106**}

[29] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press. {**13, 106**}

[30] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. {**5, 19, 33, 102, 106**}

[31] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In Wilfrid Hodges and Ruy de Queiroz, editors, *Proceedings of the 15th Workshop on Logic, Language, Information and Computation (WoLLIC 2008)*, number 5110 in Lecture Notes in Artificial Intelligence, pages 1–16, Edinburgh, Scotland, July 2008. Springer-Verlag. {**103, 106**}

[32] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press. {**4, 5, 61, 106**}

[33] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008. {**5, 16, 29, 106**}

[34] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, November 2008. {**103, 106**}

[35] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04. {**4, 5, 106**}

[36] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *Journal of Functional Programming*, 17(6):793–812, 2007. {**5, 16, 29, 107**}

[37] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23. {**4, 5, 18, 47, 107**}

[38] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4. {**2, 5, 6, 14, 107**}

[39] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987. {**4, 107**}

[40] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at `<http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>` and last accessed in April 2008, 1989-2001. {**1, 4, 5, 8, 38, 93, 101, 107**}

[41] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986. {**47, 94, 96, 107**}

[42] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. The MIT Press, third edition, 2008. {**93, 107**}

[43] 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*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, September 1999. ACM Press. {**15, 107**}

[44] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998. {**16, 30, 45, 107**}

[45] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997. {**102, 107**}

[46] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press. {**61, 108**}

[47] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. {**8, 37, 108**}

[48] Jacob Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, June 2008. BRICS research report RS-08-6. {**6, 108**}

[49] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, February 1987. Springer-Verlag. {**92, 108**}

[50] Delia Kesner. The theory of calculi with explicit substitutions revisited. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL*, number 4646 in Lecture Notes in Computer Science, pages 238–252, Lausanne, Switzerland, September 2007. Springer. {**35, 108**}

[51] Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998. {**85, 108**}

[52] Yves Lafont. *Logiques, Catégories et Machines*. PhD thesis, Université de Paris VII, Paris, France, January 1988. {**64, 108**}

[53] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. {**6, 91, 108**}

[54] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006. A preliminary version was presented at the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP 2004). {**18, 46, 108**}

[55] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975. {**87, 108**}

[56] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960. {**92, 108**}

[57] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, June 2007. {**6, 109**}

[58] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, May 2007. {**6, 109**}

[59] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. {**5, 109**}

[60] F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970. {**93, 109**}

[61] Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2007. BRICS research report RS-08-3. {**6, 109**}

[62] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, July 2001. BRICS DS-01-7. {**6, 109**}

[63] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, January 2007. ACM Press. {**5, 16, 29, 96, 109**}

[64] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. {**1, 87, 102, 109**}

[65] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark, September 1981. Reprinted in the Journal of Logic and Algebraic Programming 60-61:17-139, 2004, with a foreword [66]. {**5, 16, 29, 96, 109**}

[66] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. {**109**}

[67] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [68]. {**5, 87, 109**}

[68] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998. {**109**}

[69] Guy L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976. {**93, 110**}

[70] 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. {**5, 110**}

[71] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976. {**93, 110**}

[72] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977. {**101, 110**}

[73] 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. {**4, 110**}