BRICS

Basic Research in Computer Science

There and Back Again

Olivier Danvy
Mayer Goldberg

BRICS Report Series **RS-05-3**

ISSN 0909-0878 **January 2005**

Copyright © 2005, Olivier Danvy & Mayer Goldberg.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/05/3/`

There and Back Again *

Olivier Danvy
BRICS †
Dept. of Computer Science
University of Aarhus ‡

Mayer Goldberg
Dept. of Computer Science
Ben Gurion University §

January 2005

Abstract

We present a programming pattern where a recursive function defined over a data structure traverses another data structure at return time. The idea is that the recursive calls get us 'there' by traversing the first data structure and the returns get us 'back again' while traversing the second data structure. We name this programming pattern of traversing a data structure at call time and another data structure at return time "There And Back Again" (TABA).

The TABA pattern directly applies to computing symbolic convolutions and to multiplying polynomials. It also blends well with other programming patterns such as dynamic programming and traversing a list at double speed. We illustrate TABA and dynamic programming with Catalan numbers. We illustrate TABA and traversing a list at double speed with palindromes and we obtain a novel solution to this traditional exercise. Finally, through a variety of tree traversals, we show how to apply TABA to other data structures than lists.

A TABA-based function written in direct style makes full use of an ALGOL-like control stack and needs no heap allocation. Conversely, in a TABA-based function written in continuation-passing style and recursively defined over a data structure (traversed at call time), the continuation acts as an iterator over a second data structure (traversed at return time). In general, the TABA pattern saves one from accumulating intermediate data structures at call time.

*With apologies to Tolkien.
Extended version of an article to appear in Fundamenta Informaticae.
A preliminary version was presented as a functional pearl at ICFP 2002 [6].

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. E-mail: `danvy@brics.dk`
§Be'er Sheva 84105, Israel. E-mail: `gmayer@cs.bgu.ac.il`

Dear Reader:

Before proceeding any further, could you first spend a few minutes thinking about the following three programming exercises?

Computing a symbolic convolution:

Given two lists $[x_1, x_2, ..., x_{n-1}, x_n]$ and $[y_1, y_2, ..., y_{n-1}, y_n]$, where n is not known in advance, write a function that constructs

$$[(x_1, y_n), (x_2, y_{n-1}), ..., (x_{n-1}, y_2), (x_n, y_1)]$$

in n recursive calls and with no auxiliary list.

Detecting a generalized beta-redex:

Given the abstract-syntax tree of a lambda-term, determine whether this term is a generalized beta-redex

$$(...(((\lambda x_1.\lambda x_2. ... \lambda x_n.e)\ e_1)\ e_2) ...\ e_n)$$

where n is not known in advance, in n recursive calls and with no counter.

Detecting a palindrome:

Given a list of length n, where n is not known in advance, determine whether this list is a palindrome in $\lceil n/2 \rceil$ recursive calls and with no auxiliary list.

Thank you.

Contents

1 Symbolic convolution **1**

2 List reversal **3**

3 Polynomial multiplication **4**
3.1 Convolving successive suffixes 4
3.2 Convolving successive prefixes 4
3.3 Memory usage 5

4 Detecting a generalized beta-redex **5**
4.1 A TABA solution in direct style 6
4.2 The TABA solution, CPS-transformed 6
4.3 The TABA solution in CPS, defunctionalized 7
4.4 Changing the representation of the data-structure continuation . . 7

5 The Catalan numbers **8**

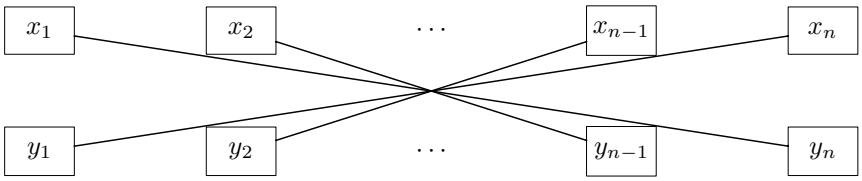
6 Detecting palindromes **9**
6.1 A CPS solution 10
6.2 A direct-style solution 10
6.3 Variations 11

7 Traversing binary trees **11**
7.1 Traversing a tree at call time 12
7.2 Traversing a tree at return time 12

8 Conclusion **14**

1 Symbolic convolution

Symbolically convolving the two lists $[x_1, x_2, ..., x_{n-1}, x_n]$ and $[y_1, y_2, ..., y_{n-1}, y_n]$ yields the list $[(x_1, y_n), (x_2, y_{n-1}), ..., (x_{n-1}, y_2), (x_n, y_1)]$. Graphically:



Numeric convolutions are used, e.g., to multiply generating functions [10, Section 5.4], and they have occurred very early in the history of mathematics [16]. Computing a symbolic convolution is straightforward in a functional programming language such as Standard ML; it is achieved by zipping the first list together with the reverse of the second list, using no accumulator, or equivalently by reversing the first list and zipping the result together with the second list, using an accumulator. We present the second solution below:

```
(*  cnv1 : 'a list * 'b list -> ('a * 'b) list  *)
fun cnv1 (xs, ys)
    = let (*  walk : 'a list * 'a list -> ('a * 'b) list  *)
          fun walk (nil, a)
              = continue (a, ys, nil)
            | walk (x :: xs, a)
              = walk (xs, x :: a)
          (*  continue : 'a list * 'b list * ('a * 'b) list  *)
          (*             -> ('a * 'b) list                   *)
          and continue (nil, nil, r)
              = r
            | continue (x :: a, y :: ys, r)
              = continue (a, ys, (x, y) :: r)
      in walk (xs, nil)
      end
```

This definition induces a compiler warning about non-exhaustive pattern matching, but this warning is not alarming since the two input lists should have the same length. (In a version of ML with dependent types [21], the type of `cnv1` would be $\forall n \in \mathbb{N} \,.\, \alpha\ list(n) \times \beta\ list(n) \rightarrow (\alpha \times \beta)\ list(n)$.)

Two iterations are performed—one to reverse the first list (`walk` above) and one to traverse the resulting reversed list and the second list (`continue` above). In addition, `walk` constructs an intermediate list.

Can we do better, i.e., can we traverse each list only once and construct no intermediate list? Similar issues have motivated the study of fusion and deforestation [3, 9, 19].

In the present case, we observe that `cnv1` is in defunctionalized form [8, 15]: the data type of the intermediate list and `continue` represent a function. In the higher-order counterpart of `cnv1` (shown just below), the first list is traversed (`walk` below)

while building a list iterator (the second parameter of `walk` below). On reaching the end of the first list, the list iterator is applied to the second list and to the initial value of an accumulator to traverse the second list and construct the result:

```
(*  cnv2 : 'a list * 'b list -> ('a * 'b) list  *)
fun cnv2 (xs, ys)
    = let (*  walk : 'a list * ('b list * ('a * 'b) list  *)
          (*                    -> ('a * 'b) list)        *)
          (*         -> ('a * 'b) list                    *)
          fun walk (nil, k)
              = k (ys, nil)
            | walk (x :: xs, k)
              = walk (xs, fn (y :: ys, r) => k (ys, (x, y) :: r))
      in walk (xs, fn (nil, r) => r)
      end
```

Defunctionalizing `walk` in `cnv2` yields `walk` in `cnv1` [8, 15]. Figuratively speaking, traversing the first list explicitly winds up a list-traversal spring and this spring is unwound over the second list.

This higher-order solution is reminiscent of the call-by-value version of Bird's famous `repmin` function [2], where a function is constructed inductively while a tree is traversed and eventually applied to an integer to construct the result [12]. In contrast to `repmin`, however, the function constructed inductively by `walk` is eventually applied to a list *and traverses it* during the construction of the result.

We observe that `walk` is written in continuation-passing style (CPS), since it threads a higher-order accumulator and all of its calls are tail calls. There is, however, nothing intrinsic to CPS about it, and therefore we can write it in direct style. The resulting function traverses the first list at call time *and the second list at return time*:

```
(*  cnv3 : 'a list * 'b list -> ('a * 'b) list  *)
fun cnv3 (xs, ys)
    = let (*  walk : 'a list -> 'b list * ('a * 'b) list  *)
          fun walk nil
              = (ys, nil)
            | walk (x :: xs)
              = let val (y :: ys, r) = walk xs
                in (ys, (x, y) :: r)
                end
          val (nil, r) = walk xs
      in r
      end
```

CPS-transforming `walk` in `cnv3` yields `walk` in `cnv2` [5]. Figuratively speaking, the calls to `walk` implicitly wind up a list-traversal spring and the returns unwind it over the second list.

This direct-style solution only allocates storage to construct the result, and all its intermediate results are held on the control stack if one uses a direct-style implementation of a derivative of ALGOL 60 such as Chez Scheme (`http://www.scheme.com`) or OCaml (`http://www.ocaml.org`).

Overview: The rest of this article further illustrates the TABA programming pattern of traversing one data structure at call time and another at return time, including trivial calls in Section 2, multiple returns in Section 3, and a tree traversal in Section 4. We combine the TABA programming pattern with dynamic programming to compute Catalan numbers in Section 5, and with traversing a list at double speed to detect palindromes in Section 6. Finally, we illustrate TABA with binary trees in Section 7.

2 List reversal

It is immediate to write a self-convolution using the TABA programming pattern. This self-convolution can be simplified into a recursive version of the reverse function that completely traverses the input list at call time and then re-traverses it at return time, constructing the result:

```
(*  taba_rev : 'a list -> 'a list  *)
fun taba_rev xs
    = let (*  walk : 'a list -> 'a list * 'a list  *)
          fun walk nil
              = (xs, nil)
            | walk (_ :: xs)
              = let val (x :: xs, r) = walk xs
                in (xs, x :: r)
                end
          val (nil, r) = walk xs
      in r
      end
```

In this degenerate version of the reverse function, the only purpose of the calls to `walk` is to reach the end of the input list, through a series of successive end-of-list tests. This list is then blindly re-traversed while returning. If we duplicate the end-of-list tests from the calls to the returns, the purpose of the calls disappears and we can optimize away the "tail-returns:"

```
(*  taba_rev_opt : 'a list -> 'a list  *)
fun taba_rev_opt xs
    = let (*  walk_return : 'a list * 'a list -> 'a list  *)
          fun walk_return (nil, r)
              = r
            | walk_return (x :: xs, r)
              = walk_return (xs, x :: r)
      in walk_return (xs, nil)
      end
```

The result is the traditional version of reverse with an accumulator.

3 Polynomial multiplication

Multiplying polynomials together requires their coefficients to be convolved. Indeed multiplying $a_0 + a_1x + a_2x^2 + ... + a_nx^n$ by $b_0 + b_1x + b_2x^2 + ... + b_nx^n$ yields $c_0 + c_1x + c_2x^2 + ... + c_{2n}x^{2n}$, where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$ and $a_k = b_k = 0$ whenever $k > n$. Polynomial multiplication is similar to integer multiplication without carry:

$$
\begin{array}{cccccccccccccc}
 & & & & & & & a_n & & a_{n-1} & & a_{n-2} \ldots & a_2 & a_1 & a_0 \\
 & & & & & & \times & b_n & & b_{n-1} & & b_{n-2} \ldots & b_2 & b_1 & b_0 \\
\hline
 & & & & & & & a_nb_0 & a_{n-1}b_0 & a_{n-2}b_0 & & \ldots a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & & & a_nb_1 & & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & & \ldots a_1b_1 & a_0b_1 & & \\
 & & & & a_nb_2 & a_{n-1}b_2 & & a_{n-2}b_2 & a_{n-3}b_2 & a_{n-4}b_2 & & \ldots a_0b_2 & & & \\
 & & & & \vdots & \vdots & & \vdots & \vdots & \vdots & & & & & \\
 & & a_nb_{n-2} \ldots & & a_4b_{n-2} & a_3b_{n-2} & & a_2b_{n-2} & a_1b_{n-2} & a_0b_{n-2} & & & & & \\
 & a_nb_{n-1} & a_{n-1}b_{n-1} \ldots & & a_3b_{n-1} & a_2b_{n-1} & & a_1b_{n-1} & a_0b_{n-1} & & & & & & \\
a_nb_n & a_{n-1}b_n & a_{n-2}b_n & \ldots & a_2b_n & a_1b_n & & a_0b_n & & & & & & & \\
\hline
c_{2n} & c_{2n-1} & c_{2n-2} & \ldots & c_{n+2} & c_{n+1} & & c_n & c_{n-1} & c_{n-2} & \ldots & c_2 & c_1 & c_0 &
\end{array}
$$

We observe that each of c_n, ..., c_{2n} results from convolving the successive suffixes of $[a_0, ..., a_n]$ and $[b_0, ..., b_n]$, and that each of c_0, ..., c_n results from convolving the successive prefixes of $[a_0, ..., a_n]$ and $[b_0, ..., b_n]$.

3.1 Convolving successive suffixes

The successive suffixes of a list are accessed by traversing this list, and they are easy to collect in another list. Accordingly, convolving the successive suffixes of two lists of length n is straightforwardly achieved by traversing the two lists side by side in $n(n+3)/2$ recursive calls (n calls for traversing the two lists and $n(n+1)/2$ other calls for convolving the successive suffixes), with no auxiliary lists:

```
(*  suffixes : 'a list * 'b list -> ('a * 'b) list list  *)
fun suffixes (xs, ys)
    = let (*  walk : 'a list * 'b list -> ('a * 'b) list list  *)
          fun walk (nil, nil)
              = nil
            | walk (xs, ys)
              = (cnv3 (xs, ys)) :: (walk (tl xs, tl ys))
      in walk (xs, ys)
      end
```

3.2 Convolving successive prefixes

The successive prefixes of a list can be accessed with the successive continuations of a copy function, and these prefixes can be collected in another list by applying these successive continuations to the empty list [4].[1] Accordingly, a simple variant of `cnv2` in Section 1 makes it possible to list the symbolic convolutions of the successive prefixes of two lists of length n in n recursive calls and $n(n+1)/2$ returns, with no auxiliary lists:

[1] "You can enter a room once, and yet leave it twice." – Peter J. Landin

```
(*  prefixes : 'a list * 'b list -> ('a * 'b) list list  *)
fun prefixes (xs, ys)
    = let (*  walk : 'a list * ('b list * ('a * 'b) list  *)
          (*                    -> ('a * 'b) list)        *)
          (*         -> ('a * 'b) list list               *)
          fun walk (nil, k)
              = (k (ys, nil)) :: nil
            | walk (x :: xs, k)
              = (k (ys, nil)) :: (walk (xs, fn (y :: ys, r)
                                          => k (ys, (x, y) :: r)))
      in walk (xs, fn (_, r) => r)
      end
```

The definition of `walk` is not in CPS since two calls to the continuation `k` are not in tail position (which is why we can return $n(n+1)/2$ times with only n recursive calls). One can still write `walk` without `k`, i.e., in "direct style," if one uses the delimited-control operators `shift` and `reset` [1, 5].

3.3 Memory usage

In the definition of `prefixes`, all the continuations are passed as arguments and never returned as results. In Lisp jargon [17], they are 'downward funargs', and `prefixes` can indeed be written in ALGOL 60 and in Pascal, i.e., using no auxiliary heap space.

Therefore, putting `prefixes` and `suffixes` together, we can multiply polynomials in a way that uses no auxiliary heap space at all. (Furthermore, in the terminology of partial evaluation [11], this definition is binding-time separated and therefore ready to be specialized with respect to its first argument.)

4 Detecting a generalized beta-redex

We want to write a function detecting whether a given lambda-term is a generalized beta-redex

$$(...(((\lambda x_1.\lambda x_2. ... \lambda x_n.e)\, e_1)\, e_2) ... e_n)$$

where n is not known in advance. The function should proceed in n recursive calls and should use no counter.

The TABA programming pattern suggests a solution where the applications are traversed at call time and the lambda-abstractions are traversed at return time. Using Wand's continuation-based program-transformation strategy [20], we can then derive the obvious iterative solution that uses a counter. Wand's strategy consists of three steps:

1. CPS-transforming a program,
2. devising a data structure to represent the continuation, and
3. changing this data structure to improve the efficiency of the program.

In practice, Wand's second step is achieved with defunctionalization [8, 15], and accordingly we use defunctionalization below. (In retrospect, we can see that we have used Step 2 and then Step 1 in Section 1.)

We represent the abstract syntax of lambda-terms with the following data type:

```
datatype exp = VAR of string
             | LAM of string * exp
             | APP of exp * exp
```

4.1 A TABA solution in direct style

A term is a generalized beta-redex if it consists of n nested applications where the leftmost innermost term is a (curried) lambda-abstraction expecting at least n arguments, for some n. Using the TABA programming pattern, we write a function that traverses the nested applications at call time and that traverses the nested lambda-abstractions at return time. We use the option data type to account for terms that are not generalized beta-redexes:

```
datatype 'a option = NONE
                   | SOME of 'a

(*  detect_beta_redex : exp -> bool  *)
fun detect_beta_redex (APP (e, _))
    = let (*  visit : exp -> exp option  *)
          fun visit (VAR _)
              = NONE
            | visit (LAM (_, e))
              = SOME e
            | visit (APP (e, _))
              = (case visit e
                   of (SOME (LAM (_, e')))
                      => SOME e'
                    | _
                      => NONE)
      in case visit e
           of (SOME _)
              => true
            | NONE
              => false
      end
  | detect_beta_redex _
    = false
```

Instead of the option data type, we could use a local exception.

4.2 The TABA solution, CPS-transformed

We CPS-transform `detect_beta_redex`, short-circuiting the option data type. The local function `visit` is now passed a term and a continuation that is only applied if the term is well-shaped:

```
(*  detect_beta_redex_cps : exp -> bool  *)
fun detect_beta_redex_cps (APP (e, _))
    = let (*  visit : exp * (exp -> bool) -> bool  *)
          fun visit (VAR _, k)
              = false
            | visit (LAM (_, e), k)
              = k e
            | visit (APP (e, _), k)
              = visit (e, fn (LAM (_, e'))
                             => k e'
                           | _
                             => false)
      in visit (e, fn _ => true)
      end
  | detect_beta_redex_cps _
    = false
```

4.3 The TABA solution in CPS, defunctionalized

We defunctionalize `detect_beta_redex_cps` by representing its continuation as a data structure with two constructors. The first constructor accounts for the initial continuation, and the second for the continuation in the recursive call to `visit`. The constructors are interpreted with an `apply` function.

```
fun detect_beta_redex_cps_def (APP (e, _))
    = let datatype cont = C0
                        | C1 of cont
          fun apply (C0, _)
              = true
            | apply (C1 k, LAM (_, e'))
              = apply (k, e')
            | apply (C1 k, _)
            = false
          fun visit (VAR _, k)
              = false
            | visit (LAM (_, e), k)
              = apply (k, e)
            | visit (APP (e, _), k)
              = visit (e, C1 k)
      in visit (e, C0)
      end
  | detect_beta_redex_cps_def _
    = false
```

4.4 Changing the representation of the data-structure continuation

We observe that the data type `cont` just above implements Peano numbers. Making it implement native integers gives an iterative function that uses a counter (and is exponentially more efficient):

```
fun detect_beta_redex_cps_def_opt (APP (e, _))
    = let fun apply (0, _)
              = true
            | apply (k, LAM (_, e'))
              = apply (k-1, e')
            | apply (k, _)
              = false
          fun visit (VAR _, k)
              = false
            | visit (LAM (_, e), k)
              = apply (k, e)
            | visit (APP (e, _), k)
              = visit (e, k+1)
      in visit (e, 0)
      end
  | detect_beta_redex_cps_def_opt _
    = false
```

In this optimized solution, the applications are iteratively traversed by `visit` and the nested lambda-abstractions are iteratively traversed by `apply`.

5 The Catalan numbers

The Catalan numbers are recursively defined as follows [10]:

$$\begin{aligned} C_0 &= 1 \\ C_n &= C_0 C_{n-1} + \ldots + C_k C_{n-k-1} + \ldots + C_{n-1} C_0 \end{aligned}$$

This specification fits the TABA programming pattern: given a list $[C_0, ..., C_{n-1}]$, one computes C_n with a numeric self-convolution.

We can define a function computing Catalan numbers using course-of-values induction, i.e., iteratively building a list of intermediate Catalan numbers in reverse order. The result reads as follows.

```
(*  catalan : int -> int  *)
fun catalan m
    = let (*  cat : int list -> int  *)
          fun cat a
              = let (*  walk : int list -> int * int list  *)
                    fun walk nil
                        = (a, 0)
                      | walk (n :: ns)
                        = let val (n' :: ns', r) = walk ns
                          in (ns', r + (n * n'))
                          end
                    val (nil, r) = walk a
                in r
                end
```

```
        (*  iterate : int * int list -> int  *)
        fun iterate (i, a)
            = if i > m
              then hd a
              else iterate (i + 1, (cat a) :: a)
    in iterate (1, [1])
    end
```

The local function `iterate` builds an intermediate list of Catalan numbers $[..., C_2, C_1, C_0]$. Given such an intermediate list, the local function `cat` yields C_n if the intermediate list starts with C_{n-1}. It traverses this list using the TABA pattern (and could be written using `foldr` instead of `walk`).

We could even take advantage of the symmetry in the definition of C_n above to traverse the first half of the intermediate list at call time (winding up a list-traversal spring), and to traverse the second half at return time (unwinding the spring).

An analogy: convolving the two halves of a list of even length. The following function takes a list and its length n, which must be even, and yields a convolution of its first and second halves. It does so in only $n/2$ calls:

```
(*  cnv_halves : 'a list * int -> ('a * 'a) list  *)
fun cnv_halves (xs, n)
    = let (* walk : int * 'a list -> ('a * 'a) list  *)
          fun walk (0, xs)
              = (xs, nil)
            | walk (n, x :: xs)
              = let val (y :: ys, r) = walk (n-2, xs)
                in (ys, (x, y) :: r)
                end
          val (nil, r) = walk (n, xs)
      in r
      end
```

Applying `cnv_halves` to `[0,1,2,3,4,5,6,7,8,9]` and `10`, for example, yields `[(0,9),(1,8),(2,7),(3,6),(4,5)]` in five recursive calls. The idea applies directly to defining another function computing Catalan numbers using course-of-values induction, with half as many calls to `walk` in `cat`. We leave this definition as an exercise for the reader.

6 Detecting palindromes

A list L is a palindrome if it is the concatenation of a list and of its reverse, with possibly an element in between if the length of L is odd. To detect whether a list is a palindrome, given its length, we can just traverse half of the list at call time and traverse the other half at return time, as in `cnv_halves` in Section 5. But what if we do not know its length?

Actually, we do not need to know the length of a list to reach its middle if we use two pointers—one going twice as fast as the other, as in the tortoise-and-hare algorithm for detecting circularities [17, Section 15.2]. Eventually, the fast one either points to the empty list or it points to a list whose tail is the empty list. The slow one then points to the middle of the list.

Once we have reached the middle of the list, we can return the second half of the list and use the chain of returns to traverse it, incrementally comparing each of its elements with the corresponding element from the first half. There is no need to test for the end of the list, since by construction, there are precisely enough returns to scan both halves of the input list. Using CPS, the returns manifest themselves as a function traversing a list, i.e., as a list iterator.

6.1 A CPS solution

```
(*  pal_c : ''a list -> bool  *)
fun pal_c xs
    = let (*  walk : ''a list * ''a list * (''a list -> bool) -> bool  *)
          fun walk (xs1, nil, k)
              = k xs1  (* even length *)
            | walk (_ :: xs1, _ :: nil, k)
              = k xs1  (*  odd length *)
            | walk (x :: xs1, _ :: _ :: xs2, k)
              = walk (xs1, xs2, fn (y :: ys) => x = y andalso k ys)
      in walk (xs, xs, fn nil => true)
      end
```

The local function `walk` is passed the input list twice and an initial continuation, and it traverses the list recursively. For the i-th call to `walk` (starting at 0), the three arguments are the i-th tail of the input list, the $2i$-th tail, and a continuation. Eventually, the continuation is applied to the second half of the input list, which is of length n if the input list is of length $2n$ or $2n + 1$. The continuation of the i-th call is only invoked if listing the $n - i$ right-most elements of the first half of the input list and the $n - i$ left-most elements of the second half forms a palindrome.

The continuation of `walk` is a list iterator for scanning the second half of the input list. This iterator either completes the traversal and yields `true`, or it aborts and yields `false`. It is not used linearly and therefore writing this program in direct style requires a control operator [7]. In the following direct-style solution, we choose to use a local exception. (Instead of a local exception, we could use the `option` data type.)

6.2 A direct-style solution

```
(*  pal_d : ''a list -> bool  *)
fun pal_d xs0
    = let exception FALSE
          (*  walk : ''a list * ''a list -> ''a list  *)
          fun walk (xs1, nil)
              = xs1  (* even length *)
```

```
        | walk (_ :: xs1, _ :: nil)
          = xs1  (*  odd length *)
        | walk (x :: xs1, _ :: _ :: xs2)
          = let val (y :: ys) = walk (xs1, xs2)
            in if x = y
               then ys
               else raise FALSE
            end
    val nil = walk (xs0, xs0)
in true
end handle FALSE => false
```

The local function `walk` is passed the input list twice and traverses the list recursively. For the i-th call to `walk` (starting at 0), the two arguments are the i-th tail of the input list and the $2i$-th tail. Eventually, the second half of the input list, which is of length n, is returned. Each i-th call returns normally if listing the $n - i$ right-most elements of the first half of the input list and the $n - i$ left-most elements of the second half forms a palindrome. Otherwise the computation aborts and yields `false`.

This direct-style version demonstrates that one can detect whether a list is a palindrome in one traversal, with no list reversal, and using no other space than what is provided by a traditional control stack—a solution that is more efficient than the traditional solutions from transformational programming [14, Example 3]. Specifically, if a list has length m, Pettorossi and Proietti count $2m$ `hd`-operations, $2m$ `tl`-operations, m `cons`-operations, and m closures both for their solution [13, Section 2, page 410] and for Bird's solution [2]. In contrast, our solution requires m `hd`-operations if m is even and $m - 1$ if m is odd, $2m$ `tl`-operations, 0 `cons`-operations, and 0 closures. On the other hand, the auxiliary data in Pettorossi and Proietti's solution and in Bird's solution could all be allocated in one region and deallocated at once upon completion of the computation [18].

6.3 Variations

For the same number of operations, we could halve the number of recursive calls by using four pointers instead of two to traverse the putative palindrome. We could even halve it further by using eight pointers, etc.

Using three pointers, we could also recognize 3-palindromes (i.e., the concatenation of three occurrences of a list of length n or of its reverse) in n recursive calls. And using m pointers, we could recognize m-palindromes (i.e., the concatenation of m occurrences of a list of length n or of its reverse) in n recursive calls and no auxiliary list, for any given m.

7 Traversing binary trees

We now present some examples where binary trees are traversed using the TABA programming pattern. We first traverse a tree at call time and a list at return time (Section 7.1), and next a list at call time and a tree at return time (Section 7.2).

It is then simple to write a function that traverses a tree at call time and another tree at return time.

We use labelled binary trees:

```
datatype 'a tree = LEAF
                 | NODE of 'a tree * 'a * 'a tree
```

7.1 Traversing a tree at call time

Labelling a tree in infix order or in postfix order illustrates the TABA programming pattern since it requires traversing this tree at call time and the list of labels at return time (we assume this list to be long enough):

```
(*  label_infix : 'a tree * 'b list -> 'b tree  *)
fun label_infix (t, labels)
    = let (*  visit : 'a tree * 'b list -> ('a * 'b) tree * 'b list  *)
          fun visit (LEAF, labels)
              = (LEAF, labels)
            | visit (NODE (t1, v, t2), labels)
              = let val (t1', label :: remaining_labels1)
                        = visit (t1, labels)
                    val (t2', remaining_labels2)
                        = visit (t2, remaining_labels1)
                in (NODE (t1', (v, label), t2'), remaining_labels2)
                end
          val (t', labels') = visit (t, labels)
      in t'
      end
```

The list is traversed every time `visit` returns from a left subtree. To label a tree in postfix order, the list would be traversed every time `visit` returned from a right subtree. (In contrast, to label a tree in prefix order, the list would be traversed every time `visit` is called on a node, and therefore this classical tree traversal does not illustrate the TABA programming pattern.)

7.2 Traversing a tree at return time

The following data type specifies directions for traversing a binary tree:

```
datatype direction = LEFT
                   | RIGHT
```

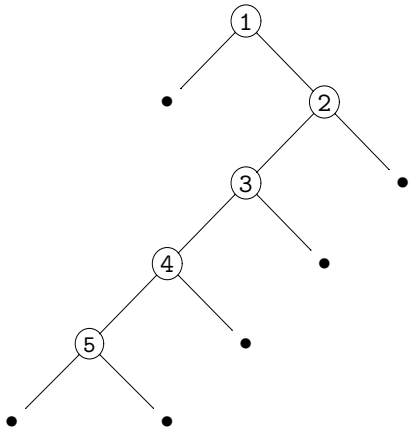
The following function is given a tree of at least depth n and a list of n directions *in reverse order*; it returns the corresponding list of node attributes from inside the tree to its root:

```
(*  traverse : 'a tree * direction list -> 'a list  *)
fun traverse (t, ds)
    = let (*  visit : 'a tree * 'a list * direction list -> 'a list  *)
          fun visit (_, nil, vs)
              = vs
```

```
        | visit (NODE (t, v, _), LEFT :: ds, vs)
          = visit (t, ds, v :: vs)
        | visit (NODE (_, v, t), RIGHT :: ds, vs)
          = visit (t, ds, v :: vs)
    in visit (t, rev ds, nil)
    end
```

This function reverses the list of directions and iteratively traverses it. According to the directions, it traverses the tree iteratively and accumulates the resulting list of node attributes.

For example, given a tree



and a list `[LEFT, LEFT, LEFT, RIGHT]`, `traverse` yields `[4, 3, 2, 1]`.

Instead of reversing the list of directions, a TABA-based function traverses it recursively at call time and then traverses the tree at return time, accumulating the resulting list of node attributes:

```
(*  traverse_taba : 'a tree * direction list -> 'a list  *)
fun traverse_taba (t, ds)
    = let (*  visit : direction list -> 'a tree * 'a list  *)
          fun visit nil
              = (t, nil)
            | visit (LEFT :: ds)
              = let val (NODE (t, v, _), vs) = visit ds
                in (t, v :: vs)
                end
            | visit (RIGHT :: ds)
              = let val (NODE (_, v, t), vs) = visit ds
                in (t, v :: vs)
                end
          val (_, vs) = visit ds
    in vs
    end
```

Perhaps more succinctly, one can use `foldr`:

```
(*  foldr : ('a * 'b -> 'b) * 'b -> 'a list -> 'b  *)
fun foldr (f, b)
    = let (*  visit : 'a list -> 'b  *)
          fun visit nil
              = b
            | visit (x :: xs)
              = f (x, visit xs)
      in visit
      end

(*  traverse_taba_with_foldr : 'a tree * direction list -> 'a list  *)
fun traverse_taba_with_foldr (t, ds)
    = let (*  back_again : direction * ('a tree * 'a list)
                            -> 'a tree * 'a list  *)
          fun back_again (LEFT, (NODE (t, v, _), vs))
              = (t, v :: vs)
            | back_again (RIGHT, (NODE (_, v, t), vs))
              = (t, v :: vs)
          val (_, vs) = foldr (back_again, (t, nil)) ds
      in vs
      end
```

The order of the traversal (i.e., whether to go left or right in the tree) is inherited at call time, since it solely depends on the list of directions. It could also be synthesized at return time if it depended on an intermediate result.

8 Conclusion

The TABA programming pattern stems from the observation that traversing a data structure recursively provides enough computing power to traverse another data structure iteratively, at return time. This second traversal can be either implicit in direct style or explicit in continuation-passing style, taking the form of an iterator.

In this article, we have illustrated the TABA programming pattern. When convolving two lists, we have avoided constructing an intermediate list for the sole purpose of traversing it later on, and we have shown that this design scales for multiplying polynomials. When detecting palindromes, we have also avoided constructing an intermediate list for the sole purpose of traversing it. This last example has led us to a new solution for the traditional palindrome problem. We have also illustrated how other data structures than lists can be traversed at call time as well as at return time.

Acknowledgments: We want to thank all the functional programmers and implicit computational complexity theorists whom we subjected to the examples presented here. We are also grateful to Mads Sig Ager, Małgorzata Biernacka, Patricia

Johann, Julia L. Lawall, Kevin Millikin, Henning Korsholm Rohde, Michael Sperber, and the anonymous reviewers for comments.

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

References

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

[2] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

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

[4] Olivier Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, January 1989.

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

[6] Olivier Danvy and Mayer Goldberg. There and back again. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 230–234, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[7] 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.

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

[9] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[10] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[11] Torben Æ. Mogensen. Glossary for partial evaluation and related topics. *Higher-Order and Symbolic Computation*, 13(4):355–368, 2000.

[12] Alberto Pettorossi. Program development using lambda abstraction. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference*, number 287 in Lecture Notes in Computer Science, pages 420–434, Pune, India, December 1987. Springer-Verlag.

[13] Alberto Pettorossi and Maurizio Proietti. Importing and exporting information in program development. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 405–425. North-Holland, 1988.

[14] Alberto Pettorossi and Maurizio Proietti. A comparative revisitation of some program transformation techniques. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 355–385, Dagstuhl, Germany, February 1996. Springer-Verlag.

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

[16] Jagadguru Swāmī Śrī Bhāratī Kṛṣṇa Tīrthajī Mahārāja. *Vedic Mathematics*. Motilal Banarsidass Publishers Private Limited, 1992.

[17] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.

[18] Mads Tofte, Lars Birkedal, Martin Elsman, and Nils Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[19] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989.

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

[21] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999. ACM Press.

Recent BRICS Report Series Publications

RS-05-3 Olivier Danvy and Mayer Goldberg. *There and Back Again*. January 2005. iii+16 pp. Extended version of an article to appear in *Fundamenta Informatica*. This version supersedes BRICS RS-02-12.

RS-05-2 Dariusz Biernacki and Olivier Danvy. *On the Dynamic Extent of Delimited Continuations*. January 2005. ii+30 pp.

RS-05-1 Mayer Goldberg. *On the Recursive Enumerability of Fixed-Point Combinators*. January 2005. 7 pp. Superseedes BRICS report RS-04-25.

RS-04-41 Olivier Danvy. *Sur un Exemple de Patrick Greussay*. December 2004. 14 pp.

RS-04-40 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. December 2004. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.

RS-04-39 Olivier Danvy and Lasse R. Nielsen. *CPS Transformation of Beta-Redexes*. December 2004. ii+11 pp. Superseedes an article to appear in *Information Processing Letters* and BRICS report RS-00-35.

RS-04-38 Olin Shivers and Mitchell Wand. *Bottom-Up β-Substitution: Uplinks and λ-DAGs*. December 2004.

RS-04-37 Jørgen Iversen and Peter D. Mosses. *Constructive Action Semantics for Core ML*. December 2004. 68 pp. To appear in a special *Language Definitions and Tool Generation* issue of the journal *IEE Proceedings Software*.

RS-04-36 Mark van den Brand, Jørgen Iversen, and Peter D. Mosses. *An Action Environment*. December 2004. 27 pp. Appears in Hedin and Van Wyk, editors, *Fourth ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, LDTA '04, 2004, pages 149–168.

RS-04-35 Jørgen Iversen. *Type Checking Semantic Functions in ASDF*. December 2004.

RS-04-34 Anders Møller and Michael I. Schwartzbach. *The Design Space of Type Checkers for XML Transformation Languages*. December 2004. 21 pp. Appears in Eiter and Libkin, editors, *Database Theory: 10th International Conference*, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.