Basic Research in Computer Science

BRICS RS-02-12 Danvy & Goldberg: There and Back Again

There and Back Again

Olivier Danvy
Mayer Goldberg

BRICS Report Series **RS-02-12**

ISSN 0909-0878 **March 2002**

**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/02/12/`

Functional pearl presented at the ACM SIGPLAN
2002 International Conference on Functional Programming
Pittsburgh, Pennsylvania, September 2002
⟨http://icfp2002.cs.brown.edu/⟩

There and Back Again

Olivier Danvy and Mayer Goldberg

Dear Reader:

Before proceeding any further, could you first ponder on the two following riddles?

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 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.

There and Back Again *

Olivier Danvy

BRICS †
Dept. of Computer Science
University of Aarhus ‡

Mayer Goldberg

Dept. of Computer Science
Ben Gurion University §

Abstract

We present a programming pattern where a recursive function traverses a data structure—typically a list—at return time. The idea is that the recursive calls get us there (typically to a base case) and the returns get us back again *while traversing the data structure*. We name this programming pattern of traversing a data structure at return time "There And Back Again" (TABA).

The TABA pattern directly applies to computing a symbolic convolution. It also synergizes well with other programming patterns, e.g., 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.

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, the continuation acts as a list iterator. In general, the TABA pattern saves one from constructing intermediate lists in reverse order.

*With apologies to Tolkien.

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

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: `danvy@brics.dk`

§Be'er Sheva 84105, Israel. E-mail: `gmayer@cs.bgu.ac.il`

Contents

1 A symbolic convolution **3**

2 The Catalan numbers **5**

3 Palindromes **6**
3.1 A CPS solution 7
3.2 A direct-style solution 7

4 Conclusion and issues **8**

A List reversal **9**

B Convolving successive prefixes **9**

1 A 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 $[(x_1, y_n), (x_2, y_{n-1}), ..., (x_{n-1}, y_2), (x_n, y_1)]$. Numeric convolutions are used, e.g., to multiply generating functions [6, Section 5.4], and they have occurred very early in the history of mathematics [12]. Computing a symbolic convolution is straightforward for a functional programmer; it is achieved by zipping the first list and the reverse of the second list. In Standard ML:

```
(*  cnv1 : 'a list * 'b list -> ('a * 'b) list  *)
(*   zip : 'a list * 'b list -> ('a * 'b) list  *)
(*   rev : 'b list -> 'b list                    *)

fun cnv1 (xs, ys)
    = let fun zip (nil, nil)
              = nil
            | zip (x :: xs, y :: ys)
              = (x, y) :: (zip (xs, ys))
      in zip (xs, rev ys)
      end
```

This definition induces a compiler warning about non-exhaustive pattern matching, but this warning is unfounded since the two input lists have the same length. (In a version of ML with dependent types [16], the type of `cnv1` would be $\forall n \in \mathbb{N} \,.\, \alpha\ list(n) \times \beta\ list(n) \to (\alpha \times \beta)\ list(n)$.)

At any rate, `cnv1` performs two iterations—one to reverse the second list (`rev` above), and one to traverse the first list and the reversed list (`zip` above). In addition, `rev` constructs an intermediate list.

Could we do better, i.e., could we traverse each list only once and construct no intermediate list? Similar problems have been considered before. For example, Launchbury and Sheard's warm fusion comes to mind [8].

In our solution, we traverse the first list (`walk` below) while building a list iterator (the second parameter of `walk` below). On reaching the end of the first list, we apply the list iterator to the second list to traverse it and construct the result:

```
(*  cnv2 : 'a list * 'b list -> ('a * 'b) list  *)
(*  walk : 'a list * (('a * 'b) list * 'b list  *)
(*                    -> ('a * 'b) list)        *)
(*         -> ('a * 'b) list                    *)

fun cnv2 (xs, ys)
    = let fun walk (nil, k)
              = k (nil, ys)
            | walk (x :: xs, k)
              = walk (xs, fn (r, y :: ys) => k ((x, y) :: r, ys))
      in walk (xs, fn (r, nil) => r)
      end
```

Figuratively speaking, traversing the first list winds up a list-traversal spring, which we explicitly unwind over the second list.

This higher-order solution is reminiscent of the call-by-value version of Bird's famous `repmin` function [1], where a function is constructed while a tree is traversed, and eventually applied. In contrast to `repmin`, however, `walk` is written in continuation-passing style (CPS), since its carries a higher-order accumulator and all of its calls are tail calls.

Having identified that `walk` is in CPS, and since there is nothing intrinsic to CPS about it, let us 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  *)
(*  walk : 'a list -> ('a * 'b) list * 'b list  *)

fun cnv3 (xs, ys)
    = let fun walk nil
              = (nil, ys)
            | walk (x :: xs)
              = let val (r, y :: ys) = walk xs
                in ((x, y) :: r, ys)
                end
      in let val (r, nil) = walk xs
         in r
         end
      end
```

Figuratively speaking, the calls implicitly wind up a list-traversal spring and the returns unwind it.

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 an implementation of a derivative of ALGOL 60 such as Chez Scheme (`http://www.scheme.com`) or OCaml (`http://caml.inria.fr`).

Generalizing, we can see that every time we want to fold a function over the result of zipping a list and the reverse of another list, we can avoid reversing the other list and avoid zipping. Instead, we can use only one recursive descent to traverse one list at call time and to traverse the other at return time. The situation is crystallized in the following fusion-like law, which is reminiscent of Launchbury and Sheard's warm fusion [8], though distinct from it.

Proposition 1 (There And Back Again)
For all suitably typed `f` *and* `b`*, and for all lists* `xs` *and* `ys` *with the same length,*

```
foldr f b (zip (xs, rev ys))
= let val (r, nil) = foldr (fn (x, (r, y :: ys))
                            => (f ((x, y), r), ys))
                           (b, ys)
                           xs
  in r
  end
```

The rest of this article illustrates further the TABA programming pattern of traversing a list at return time, including trivial calls in Appendix A and multiple returns in Appendix B.

2 The Catalan numbers

The Catalan numbers are recursively defined as follows [6, 7, 15]:

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

This specification fits the TABA pattern very well: 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                  *)
(*      cat : int list -> int             *)
(*     walk : int list -> int * int list  *)
(*  iterate : int * int list -> int       *)

fun catalan m
    = let fun cat a
              = let fun walk nil
                        = (0, a)
                      | walk (n :: ns)
                        = let val (r, n' :: ns')
                                  = walk ns
                          in (r + (n * n'), ns')
                          end
                in let val (r, nil) = walk a
                   in r
                   end
                end
          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 in the TABA fashion.

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, and to traverse the

second half at return time. Let us illustrate this idea of traversing the second half of a list on the momentum of traversing the first half.

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 $n/2$ calls only:

```
(*  cnv_halves : 'a list * int -> ('a * 'a) list  *)
(*        walk : int * 'a list -> ('a * 'a) list  *)

fun cnv_halves (xs, n)
    = let fun walk (0, xs)
              = (nil, xs)
            | walk (n, x :: xs)
              = let val (r, y :: ys) = walk (n-2, xs)
                in ((x, y) :: r, ys)
                end
      in let val (r, nil) = walk (n, xs)
         in r
         end
      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.

3 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 `cnv_halves` in Section 2. 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 [14, 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 in 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.

3.1 A CPS solution

```
(*  pal_c : ''a list -> bool                          *)
(*   walk : ''a list * ''a list * (''a list -> bool)  *)
(*          -> bool                                   *)

fun pal_c xs
    = let 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
```

Description: 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 parameters are the i-th tail of the input list, the $2i$-th tail, and a continuation. Eventually, the continuation is sent the second half of the input list, which is of length n. 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.

Analysis: `pal_c` constructs 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`.

The continuation is not used linearly and therefore writing this program in direct style requires a control operator [4]. In the following direct-style solution, we choose to use an exception.

3.2 A direct-style solution

```
(*  pal_d : ''a list -> bool                *)
(*   walk : ''a list * ''a list -> ''a list  *)

fun pal_d xs0
    = let exception FALSE
          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
```

```
in let val nil = walk (xs0, xs0)
   in true
   end handle FALSE => false
end
```

Description: 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 parameters 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`.

Analysis: 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 [11, 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 for their solution [10, Section 2, page 410] and for Bird's solution [1]. 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.

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 halving 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, for any given m.

4 Conclusion and issues

Maybe because of the `map` functional, lists make one think iteratively (or do we still think that recursive calls are expensive and should be avoided? [9, 13]). There is more to processing a list, however, than simply traversing it. The TABA programming pattern hinges on the fact that a recursive descent provides just enough expressive power to traverse another list iteratively, at return time. Besides, since ALGOL 60, the infrastructure for running recursive programs is geared to hold multiple intermediate results without having to represent them explicitly, e.g., in an auxiliary list. (This does not necessarily mean that a control stack is cheaper to use than the heap, especially in the presence of first-class continuations [2].)

In this article, we have put these observations to use. When convolving two lists, we have avoided constructing an intermediate list for the sole purpose of reversing it. When detecting palindromes, we have avoided constructing an intermediate list for the sole purpose of traversing it again. This last example has led us to a new solution for the traditional palindrome problem.

Acknowledgments: We want to thank all the functional programmers and implicit computational complexity theorists whom we subjected with the examples presented here. We are also grateful to Mads Sig Ager, Julia L. Lawall, Henning Korsholm Rohde, Michael Sperber, and the anonymous reviewers for comments.

A List reversal

The TABA programming pattern makes it possible to write 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            *)
(*      walk : 'a list -> 'a list * 'a list  *)

fun taba_rev xs
    = let fun walk nil
              = (nil, xs)
            | walk (_ :: xs)
              = let val (r, x :: xs) = walk xs
                in (x :: r, xs)
                end
      in let val (r, nil) = walk xs
         in r
         end
      end
```

This extreme definition is not that alien, though, since CPS-transforming it and defunctionalizing the result yields the usual reverse function with an accumulator [5].

B Convolving successive prefixes

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:

```
(*  cnv2' : 'a list * 'b list -> ('a * 'b) list list  *)
(*   walk : 'a list * (('a * 'b) list * 'b list       *)
(*                     -> ('a * 'b) list)             *)
(*          -> ('a * 'b) list list                    *)
```

```
fun cnv2' (xs, ys)
    = let fun walk (nil, k)
              = (k (nil, ys)) :: nil
            | walk (x :: xs, k)
              = (k (nil, ys)) :: (walk (xs, fn (r, y :: ys)
                                           => k ((x, y) :: r, ys)))
      in walk (xs, fn (r, _) => r)
      end
```

Indeed applying `cnv2'` to `[1,2,3,4]` and `[10,20,30,40]` yields

```
[[],
 [(1,10)],
 [(1,20),(2,10)],
 [(1,30),(2,20),(3,10)],
 [(1,40),(2,30),(3,20),(4,10)]]
```

The definition of `walk` is not in CPS since two calls to `k` are not in tail position. It can still be written without the higher-order accumulator, i.e., in "direct style," if one uses the control operator `shift` and the control delimiter `reset` [3].

References

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

[2] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

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

[4] 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.

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

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

[7] Luo Jian-Jin. Catalan numbers in the history of mathematics in China. In H.P. Yap et al., editor, *Combinatorics and Graph Theory: Proc. Spring School and International Conference on Combinatorics*, pages 68–70, Hefei, China, 1993. World Scientific.

[8] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 314–323, La Jolla, California, June 1995. ACM Press.

[9] Y. Annie Liu and Scott D. Stoller. From recursion to iteration: what are the optimizations? In Julia L. Lawall, editor, *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 34, No 11, pages 73–82, Boston, Massachusetts, November 2000. ACM Press.

[10] 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.

[11] 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.

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

[13] Guy L. Steele Jr. Debunking the 'expensive procedure call' myth. In *Proceedings of the ACM National Conference*, pages 133–162, Seattle, Washington, October 1977. Extended version available as MIT AI Memo 443.

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

[15] Robert A. Sulanke. Moments of generalized Motzkin paths. *Journal of Integer Sequences*, 3(00.1.1), 2000. `http://www.math.uwaterloo.ca/JIS/`.

[16] 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-02-12 **Olivier Danvy and Mayer Goldberg.** ***There and Back Again*****. March 2002. ii+11 pp. This report supersedes the earlier report BRICS RS-01-39.**

RS-02-11 **Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach.** ***Extending Java for High-Level Web Service Construction*****. March 2002.**

RS-02-10 **Ulrich Kohlenbach.** ***Uniform Asymptotic Regularity for Mann Iterates*****. March 2002. 17 pp.**

RS-02-9 **Anna Östlin and Rasmus Pagh.** ***One-Probe Search*****. February 2002. 17 pp. To appear in** ***29th International Colloquium on Automata, Languages, and Programming*****, ICALP '02 Proceedings, LNCS, 2002.**

RS-02-8 **Ronald Cramer and Serge Fehr.** ***Optimal Black-Box Secret Sharing over Arbitrary Abelian Groups*****. February 2002. 19 pp.**

RS-02-7 **Anna Ingólfsdóttir, Anders Lyhne Christensen, Jens Alsted Hansen, Jacob Johnsen, John Knudsen, and Jacob Illum Rasmussen.** ***A Formalization of Linkage Analysis*****. February 2002. vi+109 pp.**

RS-02-6 **Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir.** ***Equational Axioms for Probabilistic Bisimilarity (Preliminary Report)*****. February 2002. 22 pp. To appear in Kirchner and Ringeissen, editors,** ***Algebraic Methodology and Software Technology: 9th International Conference,*** **AMAST '02 Proceedings, LNCS, 2002.**

RS-02-5 **Federico Crazzolara and Glynn Winskel.** ***Composing Strand Spaces*****. February 2002. 30 pp.**

RS-02-4 **Olivier Danvy and Lasse R. Nielsen.** ***Syntactic Theories in Practice*****. January 2002. 34 pp. This revised report supersedes the earlier BRICS report RS-01-31.**

RS-02-3 **Olivier Danvy and Lasse R. Nielsen.** ***On One-Pass CPS Transformations*****. January 2002. 18 pp.**

RS-02-2 **Lasse R. Nielsen.** ***A Simple Correctness Proof of the Direct-Style Transformation*****. January 2002.**