# On Listing List Prefixes

Olivier Danvy
DIKU – Computer Science Department, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
danvy@diku.dk

March 27, 1989

## Abstract

The Lisp Puzzles feature in Lisp Pointers, Volume 1, Number 6 proposed the following exercise: given a list, compute the list of its prefixes. Surprisingly, the solutions proposed in later issues all used intermediary copies and/or traversed the original list repeatedly. This note presents a higher-order solution that does not use copies and that traverses the original list only once. Further, this solution can be simply expressed by abstracting control procedurally.

**Keywords**
First-class procedures and continuations.

## Introduction

Listing list suffixes is a simple exercise in Lisp because *it can be* done by traversing the source list once:[1]

```
(maplist (lambda (x) x) '(a b c d))
  ⇒ ((a b c d) (b c d) (c d) (d))
```

On the other hand, listing list prefixes:

```
(xpl (a b c d))
⇒ ((a) (a b) (a b c) (a b c d))
```

is an interesting exercise because Lisp lists are singly-linked. This means that the beginnings of the source list cannot be shared, and thus successive prefixes must be physically copied.

Using maplist requires reversing the list to have it in the standard order, reversing all of its prefixes and reversing the result. It seems that xpl was made to be programmed in Scheme:

```
(define xpl
  ;; List(A) -> List(List(A))
  (lambda (l)
    (reverse
      (maplist reverse
               (reverse l)))))
```

This solution is a bit luxurious since it wastes $2 \times length(l)$ cons-cells for reversing the argument and the result.

Since the tails of the prefixes cannot be shared, it is logical to wonder whether their construction could be shared. This note shows that such sharing is indeed possible.

## 1 How to solve it with first-class procedures

The possibility of sharing the construction of prefixes appears in the definitions of list copying

[1] Given the functional maplist of [5]:

```
(define maplist
  ;; [List(A) -> B] * List(A) -> List(B)
  (lambda (f l)
    (if(null? l)
      '()
      (cons (f l) (maplist f (cdr l))))))
```

```
(define direct-copy      ; List(A) -> List(A)
    (lambda (l)
        (if (null? l)
            '()
            (cons (car l) (direct-copy (cdr l))))))

(define other-copy        ; List(A) -> List(A)
    (letrec ((cps-copy    ; List(A) * [List(A) -> List(A)] -> List(A)
               (lambda (l c)
                   (if (null? l)
                       (c '())
                       (cps-copy (cdr l) (lambda (r)
                                           (c (cons (car l) r))))))))
        (lambda (l)
            (cps-copy l (lambda (r) r)))))
```

Figure 1: Two implementations of list copying

shown in Figure 1. The first is in direct style and the second in continuation-passing style.

The continuation of each recursive call abstracts the copy of the list up to this program point, *i.e.*, the successive continuations abstract the construction of the successive prefixes. Considering the list `(a b c d)`, the continuations at each recursive call are extensionally equal to the following procedures:

$P_0$ = `(lambda (r) r)`
$P_1$ = `(lambda (r) (`$P_0$` (cons 'a r))`
$P_2$ = `(lambda (r) (`$P_1$` (cons 'b r)))`
$P_3$ = `(lambda (r) (`$P_2$` (cons 'c r)))`
$P_4$ = `(lambda (r) (`$P_3$` (cons 'd r)))`

This observation leads fairly naturally to the definition of `xpl` given in Figure 2. The procedure `xpl-aux` is defined locally in `xpl`. Superficially, it resembles the definition of `cps-copy` in `other-copy`. The second argument of `xpl-aux` performs the construction of the successive prefixes of the list; it is applied at each recursive call. In the base case, it is not applied; instead, the empty list is returned. The prefixes are collected in a list, in direct style.

Following the benchmarks in Lisp Puzzles, let us count the calls to `car`, `cdr`, `cons`, and `null?`, and the number of closures built with two (immutable) free variables. For a list of 100 elements, the results are:

- 100 calls each to `car` and `cdr` because there are 100 elements in the list;
- 5150 calls to `cons` because summing the length of the prefixes yields

  $$\begin{aligned} 1+2+\ldots+100 &= (100\times 101)/2 \\ &= 5050 \end{aligned}$$

  and the result is the list of the 100 prefixes;
- 101 calls to `null?` because the list is tested from its beginning to its end, and
- 100 closures because there are 100 prefixes and each *closure* builds a prefix (by calling all its predecessors).

Considering that all these closures are downward funargs and thus are stack-allocatable, this solution compares well with the benchmarks given in Volume 2, Number 1 of Lisp

```
(define xpl               ; List(A) -> List(List(A))
    (letrec ((xpl-aux     ; List(A) * [List(A) -> List(A)] -> List(List(A))
               (lambda (l c)
                 (if (null? l)
                     '()
                     (let ((a (car l)))
                        (let ((k (lambda (r) (c (cons a r)))))
                           (cons (k '()) (xpl-aux (cdr l) k))))))))
        (lambda (l)
            (xpl-aux l (lambda (r) r)))))
```

Figure 2: A continuation-composing implementation of xpl

Pointers. Each of those solutions makes at least 5150 calls to car, 5250 calls to cdr, 5250 calls to cons, and 5352 calls to consp (Common Lisp's pair?, used instead of null?).

It is possible but beyond the scope of this note to relate the present solution to the solution in the introduction by program transformation.

Noting that this solution is almost in continuation-passing style,[2] we may wonder whether there exists a solution in direct style given first-class access to the continuation. The following section investigates such a solution.

## 2 How to solve it with first-class continuations

Actually, we cannot, by accessing the continuation of each recursive call and applying it, express the above solution in direct style. The reason is that we *never return* from applying a first-class continuation, since applying it discards the current continuation. The following Scheme example illustrates this point:

```
(add1
   (call-with-current-continuation
      (lambda (k) (+ 39 (k 2)))))
```

evaluates to 3, not to 42, as it would if k only abstracted the function computed by add1.

[2] "Almost" because continuations are not applied tail-recursively but are composed instead.

The point is that a continuation abstracts *all* the rest of the computation. To limit the extent of this abstraction, Matthias Felleisen introduced *prompts* [3].

The idea of a prompt is to define a new context of computation and to make a continuation abstract this context and this context only. A continuation is accessed with the operator control, that has the same syntax as John Reynolds' escape operator. For example,

```
(prompt
   (add1
      (control k (+ 39 (k 2)))))
```

actually evaluates to $39 + (1 + 2) = 42$, and so does

```
(prompt
   (add1
      (control k
         (+ 19 (k (+ 19 (k 2)))))))
```

since k abstracts the function computed by add1 in the context delimited by the prompt. Note that the context abstracted by control is also erased; that is, in contrast to Scheme's call-with-current-continuation, that context must be invoked explicitly. This explains why these examples evaluate to 42 and not 43.

Using prompt and control, we can express the solution of Section 1 in direct style, as shown in Figure 3. This code can be implemented with the same performance.

```
(define xpl              ; List(A) -> List(List(A))
   (letrec ((xpl-aux     ; List(A) -> List(List(A))
             (lambda (l)
                (if (null? l)
                   (control c '())
                   (let ((a (car l)))
                      (cons a
                            (control c
                               (cons (c '())
                                     (prompt (c (xpl-aux (cdr l)))))))))))
      (lambda (l)
         (prompt (xpl-aux l)))))
```

Figure 3: A direct implementation of xpl, using prompt and control

The procedure xpl-aux is defined *locally* in xpl and *applied* in a *new* context. It is in direct style and accesses the current continuation. In the base case, the continuation is captured and not used (as in the solution of Section 1). The construction of a new prefix is captured, performed, and the computation continues in a new context.

It is possible but beyond the scope of this note to convert this procedure into continuation-composing style. The result would be exactly the procedure of Section 1.

## 3 Related work

Felleisen *et al.* have addressed how to abstract control procedurally [3,4]. This work has been pursued in two general directions: Dybvig and Hieb investigated how to abstract control over embedding contexts instead of merely up to the last prompt [2]; Danvy and Filinski have proposed a framework where first-class continuations can be given a static scope and accordingly can be typed statically, and have described how to convert expressions from direct style to continuation-composing style [1].

## 4 Conclusions and issues

Listing successive prefixes of a list can be solved by sharing their construction. This exercise turns out to be a nice example where abstracting control needs to be done with true procedures that can be applied and be expected to return a result. Abstracting control with procedures is not possible in traditional programming languages: non-local exits in Lisp, first-class continuations in Scheme, and exceptions in ML all behave as imperative "black holes".

The new issues offered by abstracting control procedurally remain to be explored.

## Acknowledgements

To Andrzej Filinski and Karoline Malmkjær for their interaction.

## References

[1] Olivier Danvy, Andrzej Filinski: *A Functional Abstraction of Typed Contexts*, DIKU Report No 89/5, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (1989)

[2] R. Kent Dybvig, Robert Hieb: *Continuations and Concurrency*, Technical Report

No 256, Computer Science Department, Indiana University, Bloomington, Indiana (July 1988)

[3] Matthias Felleisen: *The Theory and Practice of First-Class Prompts*, Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp. 180–190, San Diego, California (January 1988)

[4] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, Bruce F. Duba: *Abstract Continuations: a Mathematical Semantics for Handling Full Functional Jumps*, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah (July 1988)

[5] John McCarthy: *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I*, CACM Vol. 3, No 3 pp. 184–195 (1960)

## Appendix – How to solve it in assembly language

We can expect control abstractions to be efficient on a conventional architecture, since `xpl` can be coded in a very compact way, in assembly language. Consider the labelled sequence of four instructions constructing the list `(1 2 3 4)` in the register `A1`, initialized with `nil`:[3]

```
label-4:    A1 := cons(4, A1)
label-3:    A1 := cons(3, A1)
label-2:    A1 := cons(2, A1)
label-1:    A1 := cons(1, A1)
label-0:
```

Any call to one of the labels `label-1`, ..., `label-4` with the empty list in the register `A1` will return a prefix of the list `(1 2 3 4)`. Building the sequence of prefixes of this list is solved with the following sequence of instructions, where the result is built in the register `A0` and `A1` is used as an auxiliary:

```
label-0:    A0 := cons(A1, A0)
            A1 := nil
            return
xpl-1234    A0 := nil
            A1 := nil
            call label-4
            call label-3
            call label-2
            jump label-1
```

which reflects precisely the computation of `xpl` with control abstractions. The functions computed by the calls to `label-1`, etc., are extensionally equal to the continuations at each construction point of copying the list `(1 2 3 4)`. ▢

[3] This idiom works as well with a stack-based expression machine.

## Some notes on Scheme for Common Lisp programmers

The code discussed in this issue's column is written in Scheme, but should be readable by most Common Lisp programmers. There are a few features of the Scheme language that deserve explanation, though.

The Scheme form

```
(define (name arg ...)
    body ...)
```

is analogous to the Common Lisp form

```
(defun name (arg ...)
    body ...)
```

Several functions exist in both Scheme and Common Lisp, but with different names. Of particular interest for this issue is the Scheme function `null?`, which corresponds closely to the Common Lisp functions `null`.

Scheme does not treat the names of functions differently from normal variables. It thus does not need a facility akin to the `function` special form in Common Lisp. Where a Common Lisp program might say

```
(mapcar #'(lambda (f) (funcall f 2 3))
        (list #'+ #'* #'-))
```

the equivalent Scheme program is

```
(map (lambda (f) (f 2 3))
     (list + * -))
```

Both programs yield the list `(5 6 -1)`.

FORTUNES FROM THE MARCH X3J13 COMMON LISP LUNCH

Luck will visit you on the next new moon. — DAVID MOON

BARRY MARGOLIN — Make serious decisions in the last few days of the month.

When things are hectic, it is best to accentuate safety. — SANDRA LOOSEMORE