Basic Research in Computer Science

BRICS RS-01-39 Danvy & Goldberg: There and Back Again

There and Back Again

Olivier Danvy
Mayer Goldberg

BRICS Report Series **RS-01-39**

ISSN 0909-0878 **October 2001**

**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/01/39/`

There and Back Again *

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

Mayer Goldberg
Dept. of Computer Science
Ben Gurion University §

October 4, 2001

Abstract

We illustrate a variety of programming problems that seemingly require two separate list traversals, but that can be efficiently solved in one recursive descent, without any other auxiliary storage but what can be expected from a control stack. The idea is to perform the second traversal when returning from the first.

This programming technique yields new solutions to traditional problems. For example, given a list of length $2n$ or $2n + 1$, where n is unknown, we can detect whether this list is a palindrome in $n + 1$ recursive calls and no heap allocation.

Keywords: Functional programming, program derivation, recursive descent, list processing, palindrome detection, ML, direct style, continuation-passing style (CPS).

*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

List of Figures

1 Introduction

Ever since the inception of functional programming [1, 3], lists have stood as a consistent source of inspiration for functional programmers, encouraging skill and even exuberance to the point of fostering striking new discoveries as well as hiding simple solutions. In this article, we substantiate this observation with a series of examples that are traditionally used to illustrate higher-order functions, list iterators, linearly-ordered continuations, etc. and that typically require one to copy or to reverse the input list. We show that each of these examples can also be solved using a simple, first-order recursive descent that gets us there (i.e., to the base case) and back again (with the result).

Overview: Section 2 starts off with the classical problem of segmenting a list of characters into a list of words. Our final solution is first-order, traverses the input list recursively, and constructs the result as it returns. It uses no other auxiliary space than what is ordinarily provided by a control stack. Section 3 presents a convolution subroutine for multiplying polynomials or numbers. Our final solution is first-order, traverses the first list of digits, and processes both lists as it returns. Section 4 addresses the problem of detecting whether a list of length $2n$ or $2n + 1$, where n is unknown, is a palindrome, i.e., equal to its reverse. Our final solution is first-order, traverses the input list in $n + 1$ calls, and tests both halves of the list as it returns. Section 5 concludes.

2 Segmenting a string into words

Let us revisit the classical `fields` function segmenting a list of characters into a list of words. To simplify the presentation, we consider it a function mapping a list of integers into a list of lists of integers, and we consider that zero is a separator in the input list. So our `fields` function maps the list `[0,1,2,0,3,4,5,0,6]` to the list `[[1,2],[3,4,5],[6]]`.

2.1 A higher-order solution

Fifteen years ago, John Hughes chose the `fields` function to illustrate his novel representation of lists [8]. Given a curried list constructor, Hughes's solution reads in ML [10] as follows.

```
fun cons x xs  (* 'a -> 'a list -> 'a list *)
    = x :: xs

fun fields_novel xs  (* int list -> int list list *)
    = let fun fields nil
                = nil
            | fields (0 :: xs)
                = fields xs
            | fields (x :: xs)
                = word (cons x) xs
```

```
        and word c nil
            = (c nil) :: nil
          | word c (0 :: xs)
            = (c nil) :: (fields xs)
          | word c (x :: xs)
            = word (c o (cons x)) xs
    in fields xs
    end
```

Description: `fields` traverses the input list, skipping zeroes. When it encounters a non-zero entry, it delegates the traversal to `word`, which accumulates a list constructor until it meets either zero or the end of the list, at which point it unleashes the list constructor by applying it to the empty list. If the input list is non-empty, `fields` is resumed and eventually the final result is constructed.

Analysis: The input list is traversed either by `fields` or by `word`. When `word` is in action, it carries an element of the monoid of functions from lists to lists and incrementally extends it to the right. When `word` completes, it maps the element of the monoid of functions to the corresponding element of the monoid of lists by applying it to the empty list. Hughes's general point is that concatenation exercises a constant cost in the monoid of functions (where it is implemented by function composition) whereas it has a linear cost in the monoid of lists (where it is implemented by append). Hughes's specific point is that his solution departs from the traditional solution that accumulates words in reverse order and then reverses them when they are complete.

But why turn to a higher-order representation of data when a higher-order representation of control can give the same benefit? We can write `word` in continuation-passing style (CPS) for the same effect as follows. (See Appendix A for a systematic derivation of `fields_c` from `fields_novel`.)

2.2 A CPS solution

```
fun fields_c xs  (* int list -> int list list *)
    = let fun fields nil
                = nil
            | fields (0 :: xs)
              = fields xs
            | fields (x :: xs)
              = word xs (fn (w, r) => (x :: w) :: (fields r))
          and word nil k
              = k (nil, nil)
            | word (0 :: xs) k
              = k (nil, xs)
            | word (x :: xs) k
              = word xs (fn (w, r) => k (x :: w, r))
      in fields xs
      end
```

Description: `fields` traverses the input list, skipping zeroes. When it encounters a non-zero entry, it delegates the traversal to `word` with a continuation that will resume the traversal on the rest of the input list and eventually construct the final result. Whereas `fields` is in direct style, `word` is in CPS. It traverses the input list until it meets either zero or the end of the list, at which point it sends an empty word and the rest of the list to its continuation. This continuation incrementally extends the word to the left and eventually resumes `fields`.

Analysis: The input list is traversed recursively by `fields` and iteratively by `word`. When `word` is in action, it carries a continuation and incrementally extends it while traversing the input list. When `word` completes, it activates its continuation. Like Hughes's, this solution involves no list concatenation and no list reversal.

But why turning to higher-order continuation-passing style when first-order direct style gives the same result? We write `word` in direct style for the same effect as follows.

2.3 A direct-style solution

```
fun fields_d xs  (* int list -> int list list *)
    = let fun fields nil
              = nil
            | fields (0 :: xs)
              = fields xs
            | fields (x :: xs)
              = let val (w, r) = word xs
                in (x :: w) :: (fields r)
                end
          and word nil
              = (nil, nil)
            | word (0 :: xs)
              = (nil, xs)
            | word (x :: xs)
              = let val (w, r) = word xs
                in (x :: w, r)
                end
      in fields xs
      end
```

Description: `fields` traverses the input list, skipping zeroes. When it encounters a non-zero entry, it entrusts `word` to traverse the list and to return both the next word and the rest of the list. Both `fields` and `word` are in direct style and recursive: `word` traverses the input list until it meets either zero or the end of the list, at which point it returns an empty word and the rest of the list. Each of its intermediate results consists of an incomplete word (which is then extended to the left) and the rest of the list.

Analysis: The input list is traversed recursively by both `fields` and `word`. When `fields` returns, it is to construct the result list. When `word` returns, it is to construct a word. Like its CPS counterpart, this solution involves no list concatenation and no list reversal.

2.4 Assessment

Let us compare the space requirements of the three solutions. Hughes's novel solution and the CPS solution allocate heap space to represent the auxiliary list constructors and continuations. Indeed, defunctionalizing them (in the sense of John Reynolds [6, 14]) readily shows that in effect both solutions construct an intermediate copy of the reversed words, as in the traditional solution. It would take an optimizing compiler to detect that the list constructors and the continuations are ordered linearly [7, 13, 17] and thus that they can be allocated LIFO. In contrast, the direct-style version only uses cons to construct the result, and all its intermediate results are held on the control stack if one uses Chez Scheme (`http://www.scheme.com`), OCaml (`http://caml.inria.fr`), or another derivative of ALGOL 60.

3 A convolution

We consider the problem of zipping a list and the reverse of another list of the same length. Typically this is done in two iterations—one to reverse one list (`rev` below), and one to traverse both lists (`zip` below):

```
fun convolution_t (xs, ys)  (* 'a list * 'b list -> ('a * 'b) list *)
    = let fun zip (nil, nil)
              = nil
            | zip (xs, ys)
              = (hd xs, hd ys) :: (zip (tl xs, tl ys))
      in zip (xs, rev ys)
      end
```

As in Section 2.2, however, we can traverse one of the lists (`walk` below), build a list iterator (the second parameter of `walk`), and eventually apply it to the other list to traverse it and construct the result:

```
fun convolution_c (xs, ys)  (* 'a list * 'b list -> ('a * 'b) list *)
    = let fun walk nil k
              = k (nil, ys)
            | walk (x :: xs) k
              = walk xs (fn (zs, ys) => k ((x, hd ys) :: zs, tl ys))
      in walk xs (fn (zs, _) => zs)
      end
```

Defunctionalizing the continuations [6, 14] precisely yields the program reversing a list and traversing the other together with the reversed list.

Alternatively, as in Section 2.3, we can traverse the first list recursively (calls) and then the second (returns) to implement the convolution in direct style:

```
fun convolution_d (xs, ys)  (* 'a list * 'b list -> ('a * 'b) list *)
    = let fun walk nil
              = (nil, ys)
            | walk (x :: xs)
              = let val (zs, ys) = walk xs
                in ((x, hd ys) :: zs, tl ys)
                end
          val (zs, _) = walk xs
      in zs
      end
```

This discrete convolution can be used, e.g., to multiply polynomials or numbers. It was used very early in the history of mathematics (see Appendix C).

4 Detecting palindromes

Say that we need to detect whether a list is a palindrome (and thus is of length $2n$ or $2n+1$, for some n). (The situation easily generalizes to a list being built as the multiple concatenation of the same list and of its reverse.) Can we solve this problem with $n+1$ recursive calls?

The answer is positive. We can traverse the entire list in $n+1$ recursive calls with two pointers—one going twice as fast as the other. After $n+1$ calls, the fast one points to the empty list and the slow one points to the middle of the list. We can then 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.

4.1 A CPS solution

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

Description: The local function `walk` is passed the original list twice and a constant 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 original list, the $2i$-th tail, and the continuation. Eventually, the continuation is sent the

second half of the 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 of the input list 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 mapping this program back to direct style requires a control operator [5]. Using an exception would do, but since `walk` is written in CPS rather than returning a disjoint sum, we choose to use `call/cc` [4], which Standard ML of New Jersey provides in `SMLofNJ.Cont`.

4.2 A direct-style solution

```
fun pal_d xs0  (* ''a list -> bool *)
    = callcc (fn k => let fun walk (xs1, nil)
                              = xs1      (* even length *)
                            | walk (xs1, _ :: nil)
                              = tl xs1   (*  odd length *)
                            | walk (xs1, _ :: xs2)
                              = let val ys = walk (tl xs1, tl xs2)
                                in if hd xs1 = hd ys
                                   then tl ys
                                   else throw k false
                                end
                      in let val _ = walk (xs0, xs0)
                         in true
                         end
                      end)
```

This direct-style version demonstrates that one can detect whether a list is a palindrome in one (and a half) 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 [12, Example 3]. Specifically [11, Section 2, page 410], 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 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.

In Appendix B, we reproduce the code of our solution in Scheme [9]. This code does not use pattern matching and thus it makes explicit all the occurrences of the hd- and tl-operations (i.e., in Scheme, `car` and `cdr`).

5 Conclusion and issues

Processing a list does not merely reduce to traversing it iteratively. A recursive descent provides just enough expressive power to traverse another list iteratively, at return time. To put it otherwise, a list can be traversed at call time, and another one can be traversed at return time. As an added incentive to using recursive descent, 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.

In this article, we have put these observations to use in three situations. In the two first examples (segmenting a list and computing a discrete convolution) we have avoided constructing an intermediate list for the sole purpose of reversing it. In the third example, 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.

A From higher-order lists to continuations

In some sense, the `word` function in Hughes's solution (see Section 2.1) is mostly in CPS, in that it is iterative and accumulates what to do next by composing the list constructor as if it were a continuation. The only hitch is the base case, where the list constructor is not used tail recursively. We can, however, express the base case as the composition of the list constructor and of a `continue` function as follows.

```
fun fields_novel' xs  (* int list -> int list list *)
    = let fun fields nil
              = nil
            | fields (0 :: xs)
              = fields xs
            | fields (x :: xs)
              = word (cons x) xs
          and continue w xs
              = w :: (fields xs)
          and word c nil
              = (continue o c) nil nil
            | word c (0 :: xs)
              = (continue o c) nil xs
            | word c (x :: xs)
              = word (c o (cons x)) xs
      in fields xs
      end
```

Now, since function composition is associative, we can relocate `continue` to the initialization of the list constructor.

```
fun fields_novel'' xs  (* int list -> int list list *)
    = let fun fields nil
              = nil
            | fields (0 :: xs)
              = fields xs
            | fields (x :: xs)
              = word (continue o (cons x)) xs
          and continue w xs
              = w :: (fields xs)
          and word c nil
              = c nil nil
            | word c (0 :: xs)
              = c nil xs
            | word c (x :: xs)
              = word (c o (cons x)) xs
      in fields xs
      end
```

The result is a `word` function in CPS. Inlining function composition, uncurrying the continuation, and swapping the two parameters of `word` yield the definition displayed in Section 2.2.

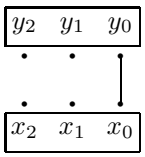
B Palindrome detection in Scheme

```
(define pal_d
  (lambda (xs)
    (call/cc
      (lambda (k)
        (letrec ([walk (lambda (xs1 xs2)
                         (if (null? xs2)
                             xs1
                             (let ([xs3 (cdr xs2)])
                               (if (null? xs3)
                                   (cdr xs1)
                                   (let ([ys (walk (cdr xs1) (cdr xs3))])
                                     (if (equal? (car xs1) (car ys))
                                         (cdr ys)
                                         (k #f)))))))])
          (begin (walk xs xs) #t))))))
```

C Background: Vedic mathematics

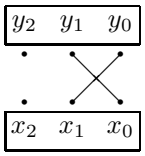
The early stage of the mathematical heritage of India is known as *Vedic Mathematics* [16]. Much of Vedic mathematics concerns algorithms for computing common number-theoretic functions in ways that require writing down little or no intermediate results. Therefore, computations can generally be carried out mentally [15, Chapter 10, page 110].

Computing s_0:



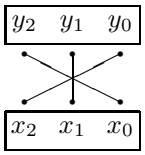
$s_0 = x_0 \cdot y_0$
$z_0 = (s_0 + c_0)$ **mod** 10
$c_1 = \lfloor (s_0 + c_0)/10 \rfloor$

Computing s_1:



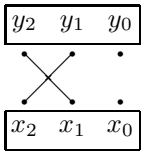
$s_1 = x_0 \cdot y_1 + x_1 \cdot y_0$
$z_1 = (s_1 + c_1)$ **mod** 10
$c_2 = \lfloor (s_1 + c_1)/10 \rfloor$

Computing s_2:



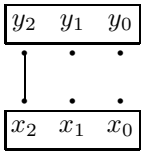
$s_2 = x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0$
$z_2 = (s_2 + c_2)$ **mod** 10
$c_3 = \lfloor (s_2 + c_2)/10 \rfloor$

Computing s_3:



$s_3 = x_1 \cdot y_2 + x_2 \cdot y_1$
$z_3 = (s_3 + c_3)$ **mod** 10
$c_4 = \lfloor (s_3 + c_3)/10 \rfloor$

Computing s_4



$s_4 = x_2 \cdot y_2$
$z_4 = (s_4 + c_4)$ **mod** 10
$c_5 = \lfloor (s_4 + c_4)/10 \rfloor$

The diagrams depict how $s_0, \ldots, s_4$ are computed when multiplying the two 3-digit numbers $x_2 \cdot 100 + x_1 \cdot 10 + x_0$ and $y_2 \cdot 100 + y_1 \cdot 10 + y_0$. Note that numbers are written left-to-right in "big-endian" fashion, so the coefficients of the lower powers are to the right. The number $x_2 \cdot 100 + x_1 \cdot 10 + x_0$ is thus written as the concatenation of the three digits x_2, x_1, and x_0, and similarly the product is computed right-to-left starting from s_0.

Figure 1: Computing $s_0, \ldots, s_4$ out of x_0, x_1, x_2 and y_0, y_1, y_2

In one of the classical expositions on the subject of Vedic Mathematics [16, Chapter 3], the author describes an algorithm for computing products of numbers digit-by-digit, that is, one digit at a time, starting with the lower powers. Given the numbers X, Y with respective digits $x_0, x_1, \ldots, x_n$, and $y_0, y_1, \ldots, y_n$, the product Z (with digits $z_0, z_1, \ldots, z_{2n+1}$) is computed as follows. Starting with the lower powers, the 0-based k-th digit of the product, z_k, is computed from the sum s_k of all the products of a_i, b_j such that $k = i + j$:

$$s_k = \sum_{i,j:i+j=k} a_i \cdot b_j$$

Let c_k be the k-th carry, then

$$\begin{aligned} z_k &= (s_k + c_k) \textbf{ mod } 10 \\ c_{k+1} &= \lfloor (s_k + c_k)/10 \rfloor \end{aligned}$$

where $c_0 = 0, z_{2n+1} = c_{2n}$. The originality of this algorithm is that (1) at any point in computing a product, one only needs to remember the information necessary for computing the next digit, and (2) this information can be maintained via a single accumulator, which amounts to remembering the partial sum used for computing s_k. (See Figure 1.)

The process of computing s_n is known in Vedic Mathematics as *Ūrdhva Tiryagbhyām*, which is Sanskrit for "vertically and crosswise", The process can be thought of as a discrete *convolution*, or a dot product of the list of digits $(x_0, x_1, \ldots, x_n)$ and the *reverse* of the list of digits $(y_0, y_1, \ldots, y_n)$, i.e., $(y_n, \ldots, y_1, y_0)$.

References

[1] David W. Barron and Christopher Strachey. Programming. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 49–82. Pergammon Press, 1966.

[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] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[5] 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.

[6] 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. An extended version is available as the technical report BRICS RS-01-23.

[7] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.

[8] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[9] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[11] 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.

[12] 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.

[13] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, number 1581 in Lecture Notes in Computer Science, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag.

[14] 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).

[15] Steven Smith. *The Great Mental Calculators: The Psychology, Methods, and Lives of Calculating Prodigies Past and Present*. Columbia University Press, 1983.

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

[17] Steve Zdancewic and Andrew Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 46–61, Genova, Italy, April 2001. Springer-Verlag.

Recent BRICS Report Series Publications

RS-01-39 Olivier Danvy and Mayer Goldberg. *There and Back Again*. October 2001. 14 pp.

RS-01-38 Zoltán Ésik. *Free De Morgan Bisemigroups and Bisemilattices*. October 2001. 13 pp.

RS-01-37 Ronald Cramer and Victor Shoup. *Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption*. October 2001. 34 pp.

RS-01-36 Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. *Cache Oblivious Search Trees via Binary Trees of Small Height*. October 2001.

RS-01-35 Mayer Goldberg. *A General Schema for Constructing One-Point Bases in the Lambda Calculus*. September 2001. 6 pp.

RS-01-34 Flemming Friche Rodler and Rasmus Pagh. *Fast Random Access to Wavelet Compressed Volumetric Data Using Hashing*. August 2001. 31 pp.

RS-01-33 Rasmus Pagh and Flemming Friche Rodler. *Lossy Dictionaries*. August 2001. 14 pp. Short version appears in Meyer auf der Heide, editor, *9th Annual European Symposiumon on Algorithms*, ESA '01 Proceedings, LNCS 2161, 2001, pages 300–311.

RS-01-32 Rasmus Pagh and Flemming Friche Rodler. *Cuckoo Hashing*. August 2001. 21 pp. Short version appears in Meyer auf der Heide, editor, *9th Annual European Symposiumon on Algorithms*, ESA '01 Proceedings, LNCS 2161, 2001, pages 121–133.

RS-01-31 Olivier Danvy and Lasse R. Nielsen. *Syntactic Theories in Practice*. July 2001. 37 pp. Extended version of an article to appear in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001 (Firenze, Italy, September 4, 2001).