Information Processing Letters 22 (1986) 141–144
North-Holland

3 March 1986

A NOVEL REPRESENTATION OF LISTS AND ITS APPLICATION TO THE FUNCTION "REVERSE"

R. John Muir HUGHES *

Institute for Dataprocessing, Chalmers Technical University, 41296 Göteborg, Sweden

Communicated by L. Boasson
Received November 1984
Revised May 1985

A representation of lists as first-class functions is proposed. Lists represented in this way can be appended together in constant time, and can be converted back into ordinary lists in time proportional to their length. Programs which construct lists using append can often be improved by using this representation. For example, naive reverse can be made to run in linear time, and the conventional 'fast reverse' can then be derived easily. Examples are given in KRC (Turner, 1982), the notation being explained as it is introduced. The method can be compared to Sleep and Holmström's proposal (1982) to achieve a similar effect by a change to the interpreter.

Keywords: Functional programming, list processing, data representation, program transformation

Before describing the new representation for lists, we review the general theory of representations and abstractions. Whenever an abstract data type A is represented by a concrete data type R, two functions must be defined:

$\text{abs} : R \to A$,
$\text{rep} : A \to R$.

The function rep converts an abstract object into its representation, and the function abs recovers the abstract object from the representation. There are two senses in which a concrete object can represent an abstract object: either it can be the representation constructed by rep, or it can be any object which represents the abstract object according to abs. For these two senses to agree, we require

FORALL a : A.abs(rep a) = a.

Now, suppose that $f : A \to A$ is an operation on abstract objects which must be implemented on representations. We require a function $g : R \to R$ which 'implements' f in some sense. This sense can be made precise by referring to abs and rep. G implements f precisely if

FORALL r : R.f(abs r) = abs(g r).

This law can be used to prove the correctness of such implementations.

In the example under consideration, A, the set of abstract objects, is the set of ordinary lists. Lists are represented by functions from lists to lists—so R is a subset of $A \to A$. A list is represented by the function that appends it onto the front of another list. Thus, the representation function, rep, is just append.

rep L = append L.

(Here we are making use of the KRC convention that all functions are curried. Since append takes two arguments, append L is a function which takes one argument, say X, and returns append L X.)

Such representations may be converted back

* This work was carried out with the support of a Research Fellowship from the U.K. Science and Engineering Research Council.

into ordinary lists using the abstraction function abs defined by

abs f = f[ ].

(Lists in KRC are written by enclosing the elements in square brackets, so [ ] is the empty list or "NIL".) In order to verify that this representation is correct, we must prove that

abs(rep a) = a.

But

abs(rep a) = (rep a)[ ]
= (append a)[ ]
= append a [ ]
= a (by properties of append).

So, abs and rep are a correct implementation of lists.

The new representation is interesting because two representations can be appended together by composing them as functions, so we can define

appendR f g = f · g.

(Function composition is written using a dot in KRC.) To see that this is correct, observe that, for all lists F, G, and H,

(appendR(rep F)(rep G))H
= (rep F · rep G)H
= rep F (rep G H)
= append F (append G H)
(since rep = append)
= append(append F G)H (since append is associative)
= (rep(append F G))H,

and, therefore,

appendR(rep F)(rep G) = rep(append F G).

This proves that appendR maps representations of lists into representations of lists. Since any list representations f and g must be equal to rep F and rep G for some F and G, we can also deduce

abs(appendR f g)
= abs(rep(append F G))
= append F G
= append(abs f)(abs g)

and so appendR correctly implements append.

Function composition is an efficient operation. It can always be performed in constant time, since no actual computation is involved. The function rep also takes constant time, so it remains to be shown that abs is reasonably efficient.

Provided that a 'functional list' is built using only rep and appendR, abs can convert it into an ordinary list in time proportional to its length. To see why, consider the general form of such a functional list. It is a composition of n partial applications of append, i.e.,

append L1 · append L2 · . . . · append Ln.

When abs applies such a function to the empty list to convert it back into an ordinary list, it computes

append L1 (append L2 . . . (append Ln [ ]) . . .).

The cost of computing (append A B) is independent of B—it is just the length of A. Therefore, the cost of computing the expression above is the sum of the lengths of L1, L2, . . . , Ln, that is, the length of the final list. Often the cost of constructing a list of length n using append is greater than this, because recursive calls of append appear as left arguments of other calls of append, and so contribute several times to the total cost. This can lead to a total cost proportional to the square of the length of the list being constructed. In such cases our 'functional representation' offers a substantial improvement.

As an example, consider the function that reverses the elements of a list. This can be defined naively as follows:

reverse[ ] = [ ],

reverse(cons a x) = append(reverse x)[a].

The cost of reversing a list of n elements with this function is proportional to n − 1 (the cost of appending [a] to reverse x) plus the cost of reversing a list of length n − 1: a total cost proportional to the square of the length of the list. The function reverse can be modified to construct a reversed functional list which is then converted back into

an ordinary one.

reverse x = abs(rev x),

rev[ ] = append[ ],

rev(cons a x) = rev x · append[a].

Taking advantage of the fact that append[ ] is an identity function, and append[a] is the same as cons a, this is equivalent to

reverse x = rev x [ ] (replacing abs by its definition),

rev[ ] = id,

rev(cons a x) = rev x · cons a.

The cost of reversing a list of length n using this version is the cost of computing rev of the list, plus the cost of converting the reversed functional list back into an ordinary one. Since this functional list is constructed using append and composition only, the cost of converting it back into an ordinary one is linear in n. Moreover, since composition only takes constant time, the cost of computing rev of the list is also linear in n. The new algorithm therefore reverses an ordinary list in linear time only.

Since rev returns a function, we can add an extra parameter to its definition. We find

rev[ ] y = id y = y,

rev(cons a x)y = (rev x · cons a)y
= rev x (cons a y).

In this form, it is clear that rev is the well-known tail-recursive fast reverse function, which uses an 'accumulating parameter' (y) to build up the reversed list!

As another example, consider the function fields that breaks up a line (represented as a list of characters) into a list of words. Fields may be defined naively by

fields[ ] = [ ],

fields (cons c ℓ) = fields ℓ **if** c = space,
= word [c] ℓ **otherwise**,

word w (cons c ℓ) = word(append w [c]) ℓ
if c~ = space,

word w ℓ = cons w (fields ℓ) **otherwise.**

This version is inefficient because characters are constantly appended to the end of the word being built up (w). The cost of building a word in this way is proportional to the square of its length. By using a functional list to represent the word while it is being built, an alternative definition can be derived:

fields[ ] = [ ],

fields(cons c ℓ) = fields ℓ **if** c = space,
= word(cons c) ℓ **otherwise**,

word w (cons c ℓ) = word(w · cons c) ℓ
if c~ = space,

word w ℓ = cons(w[ ])(fields ℓ) **otherwise.**

This version is much more efficient. In this example, it is not possible to achieve the same result by introducing an accumulating parameter. The traditional way to improve fields would be to accumulate the word in reverse order, and reverse it using a fast reverse algorithm once the whole word has been found. At least under Turner's KRC implementation, the 'functional list' approach is more efficient.

We have applied this method to several small KRC examples, namely reverse, a function returning a list of all the atoms in a tree (flatten), and fields. The simplest definition of each of these functions has an $O(n^2)$ complexity, which is reduced to O(n) by using the proposed representation. We also hand-transformed each function into a version with linear complexity for comparison. Differences in efficiency were measured by comparing the number of cells claimed during execution. In each case, the 'functional list' version and the hand-optimised version were about equally efficient, with the naive version being considerably worse. The hand-transformed version of reverse was 25% more efficient than the functional list one, because the explicit presence of the accumulating parameter meant that the composition (rev x · cons a) did not have to be constructed and then applied. The hand-transformed version of fields, on the other hand, was 10% less efficient than the functional list version.

Bird [1] has also applied the functional representation to a text formatting program written in KRC, and reported a 25% improvement in the efficiency of the entire program.

References

[1] R.S. Bird, Transformational Programming and the Paragraph Problem (Oxford University Press, London, 1984).

[2] R. Sleep and S. Holmström, A short note concerning lazy reduction rules of APPEND, Software—Practice and Experience 12 (1982) 1082–1084.

[3] D.A. Turner, Recursion equations as a programming language, in: D.A. Turner, ed., Functional Programming and its Applications (Cambridge University Press, London, 1982).