List Processing Primitives for Parallel Computation*

Tom Axford
School of Computer Science, University of Birmingham
Birmingham B15 2TT, U.K.

Mike Joy
Department of Computer Science, University of Warwick
Coventry CV4 7AL, U.K.

©Copyright TH Axford & MS Joy 1992

Abstract

A new model of list processing is proposed which is more suitable as a basic data structure for architecture-independent programming languages than the traditional model of lists. Its main primitive functions are: *concatenate*, which concatenates two lists; *split*, which partitions a list into two parts; and *length*, which gives the number of elements in a list. This model contains a degree of non-determinism which allows greater freedom to the implementation to achieve high performance on both parallel and serial architectures.

Keywords: data structures, functional programming, list processing, parallel programming.

1 Introduction

Lists have been used as basic data structures within programming languages since the 1950s. The most elegant and successful formulation was in Lisp [9] with its primitive functions *car, cdr* and *cons*, often now referred to by the more meaningful names of *head, tail* and *cons* respectively. Lisp and its model of list processing based on the *head, tail* and *cons* primitives have given rise to a large number of programming languages over the three and a half decades since Lisp was invented; for example, following closely to the pure Lisp tradition are ML[20], Miranda[19] and Haskell[6].

The success of the Lisp model of list processing is due to a combination of its semantic elegance on the one hand and its simplicity and efficiency of implementation on the other.[1] In the context of functional languages particularly, it has given rise to a style of programming which is clear, concise and powerful. This style is well documented in many publications, for example [3].

*Published in *Computer Languages* **19**(1), 1–12 (1993)

[1] In the early development of Lisp, efficiency of implementation was a major concern, while the desire for an elegant and coherent semantic model of list processing was much less pressing. Nevertheless, the reason that Lisp was more successful than its list-processing competitors almost certainly had a lot to do with McCarthy's perceptive choice of the basic routines that operated on lists[10].

Despite the often proclaimed advantages of functional languages for parallel programming [13], there has been very little progress in constructing really worthwhile parallel implementations of them. A large part of the problem lies in the difficulty of obtaining efficient parallel representation of the traditional head-tail-cons model of list processing. Many recent functional languages such as Miranda and Haskell, as well as older functional languages such as List, have this model built intimately into the language. Although most modern languages are quite powerful enough to allow the programmer to define any type of data structure by defining a suitable set of primitive functions, the built-in primitives for *head, tail* and *cons* typically execute approximately an order of magnitude faster than user-defined equivalents. For this reason, almost all programs in these languages use the traditional list-processing model as a matter of course. Hence, if this model cannot be implemented efficiently in parallel, most programs are unlikely to be any better.

There are, however, many high-level list operations for which it is easy to envisage a very efficient parallel implementation, particularly those which operate on the whole list, such as *map* and *reduce* (sometimes called *fold*). Therefore, the problem is not inherent in the semantics of list processing or the concept of a list itself, but rather in the choice of a set of primitive functions, and the (usually implicit) assumption that the implementation executes these in constant time (i.e. independently of the lengths of the lists involved).

In this paper, we propose a new model of list processing based on a different set of primitive functions, chosen to be efficiently implementable on parallel architectures, but preserving the usual semantics of lists. Programs that use pattern matching on lists or explicitly refer to *head, tail* and *cons* will need major rewriting to use the new model, which is best suited to a new style of programming based on the divide-and-conquer paradigm, rather than the usual recursive/iterative style of conventional list processing. On the other hand, programs that do not use the primitives directly, but instead use purely high-level library functions may not require any changes at all!

To achieve satisfactory parallel performance requires that the primitives of the new model be built into the language and implemented directly by the compiler or interpreter. The order of magnitude penalty typically suffered by user-defined primitives means that one has to achieve ideal speedup on approximately ten parallel processors simply to equal the performance of the traditional model on a serial machine. This will usually be an unacceptable price to pay. At the very least, all the primitive functions should be built in, and, preferably, the more commonly used higher level functions as well.

2 The Model

The idea behind our model is a very old one: that of representing a list as a binary tree (e.g. as in some sorting algorithms). There are, however, many different binary trees that can represent the same list. We abstract a set of primitive functions which are suitable for use with binary tree representations, but which define list structures and no more. Although these primitives are well suited for use with binary tree representations of lists, they do not permit the programmer to see the full internal structure of the trees, hence they do require any particular tree representation, or even a tree representation at all. This element of non-determinism in the model is important, and is discussed more fully later.

2.1 Informal Description

The following six functions are chosen as the primitive functions of the model (the first is actually a constant, or a function which takes no arguments).

(i) `[]` is the empty list.

(ii) `singleton x` (or, alternatively, `[x]`) is the list which contains a single element, `x`.

(iii) `concatenate s t` (or, alternatively, `s++t`) is the list formed by concatenating the lists `s` and `t`.

(iv) `split s` is a pair of lists got by partitioning the list `s` into two parts. It is defined only if `s` contains at least two elements. Both lists are non-empty.

If `s` contains more than two elements, the result of applying `split` is non-deterministic, i.e. there is more than one acceptable solution and an implementation is free to choose which of these to give as the result.

(v) `length s` (or, alternatively, `#s`) is the number of elements in the list `s`.

(vi) `element s` is the only element present in the singleton list `s`. This function is undefined for lists which contain either more or less than one element.

The primitive `split` is non-deterministic. This is to allow the implementation to choose the quickest way to implement it that the circumstances permit. This freedom is essential to obtaining good parallel performance, as will be seen later. The reasons for choosing a non-deterministic primitive are discussed later in Section 2.3.

The result of applying `split` is a *pair* of lists (not to be confused with a list of two lists). It is assumed that the language used in examples later in this paper allows the definition of pairs of objects in a single statement, for example:

```
(s,t) = split u
```

Alternatively, if a pair of objects cannot be defined in a single statement in this way, a semantically equivalent approach is to define two primitive functions, `split1` and `split2`:

```
split u = (split1 u, split2 u),
```

so that the definition of `s` and `t` can be carried out separately:

```
s = split1 u
t = split2 u
```

2.2 Algebraic Specification

The algebraic properties of the primitive functions that can be used to specify the semantics of the model are as follows:

1. `#[] = 0`
2. `#[x] = 1`
3. `#(s++t) = #s + #t`
4. `element [x] = x`

5. `s++[] = []++s = s`
6. `s++(t++u) = (s++t)++u`
7. `split([x]++[y]) = ([x],[y])`
8. `#u >= 2,  split u = (s,t)` *implies*
   `s++t = u,  #s >= 1,  #t >= 1`

2.3 Non-Determinism

The primitive function `split` is non-deterministic as defined above. It is easy to modify the specification to make it deterministic, but there are considerable advantages in keeping it the way it is. Consider two obvious ways in which the non-determinism could be removed.

Firstly, we could replace the second line of Axiom 8 by:

```
s++t = u,  #s = #u DIV 2, #t = (#u+1) DIV 2
```

This would mean that `split` always divides the list in half (or as close to that as feasible). The disadvantage with this is that it constrains the implementation unnecessarily. It is very hard to find any representation of lists that makes this easy to do without introducing other inefficiencies elsewhere.

Secondly, we could replace Axioms 7 and 8 by the single axiom:

```
split (s++t) = (s,t)
```

provided `s` and `t` are each non-empty lists. However, while this certainly removes the non-determinism, it changes the specification from one for lists into one for trees: the internal structure is now visible to the programmer, not just the order of the elements. Axiom 6 (associativity of concatenation) would have to be removed as it is contradicts the new axiom. This is certainly not what we want. Two lists must always be equal if they contain the same collection of elements in the same order. The way in which the list was originally constructed should not be significant.

Of course, it could be argued: why not provide trees instead of lists as the basic data structures, after all they are more general and include lists as simply a special case? The answer to this question is more subtle, but equally definite. No commonly-used language has done this, although the argument in favour of trees has always been relevant and has nothing to do with parallelism. The argument against trees is simply that most problems do not need the extra internal structure that can be represented by a tree. Lists are fully adequate in the vast majority of situations. Carrying around the excess baggage of the extra complexity of trees when it is generally unnecessary is highly undesirable. The most successful languages have usually been the simplest ones, not the most complicated ones.

To maintain this simplicity, while at the same time giving sufficient freedom to the implementor to vary the implementation to fit the computer architecture available, it is worth introducing non-determinism, provided that it can be kept strictly under control. With a little extra care on the part of the programmer it is not difficult to write fully deterministic programs using some operators or functions which are non-deterministic. Some of the higher-level functions introduced later illustrate this well (e.g. `reduce`).

As a general principle, it seems very likely that some carefully selected non-deterministic operators will need to be introduced into most architecture-independent programming languages. For example, if it is required to compute

$$a_1 + a_2 + ... + a_n$$

then it does not matter in which order the additions are carried out, the sum will always be the same because addition is associative and commutative. However, a program which specifies that the numbers are added sequentially, one at a time from the left (as is typical in most present-day programming languages) constrains the implementor quite unnecessarily (and highly inefficiently on most parallel machines). The best way to write a summation program that can be run at optimum efficiency on all different types of machine architecture is to use a language in which the order in which the additions are to be performed is undefined. The non-determinism becomes a problem only if such an order-undefined program is written using a non-associative operator instead of addition.

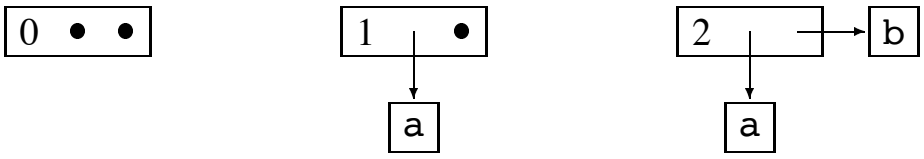
Hence the use of non-deterministic primitives does impose an obligation on the programmer to verify that the operators used have the required properties, such as associativity.

2.4 Representation in the Computer

There are many different ways to represent lists. The representation described below is not claimed to be the only suitable representation, nor the best representation. It is, however, simple, well known, and suitable for high-performance parallel implementation on a shared memory architecture, as well as allowing constant-time execution of all the new primitives. Discussion of parallel execution is left until a later section, and for the moment we simply describe the representation.

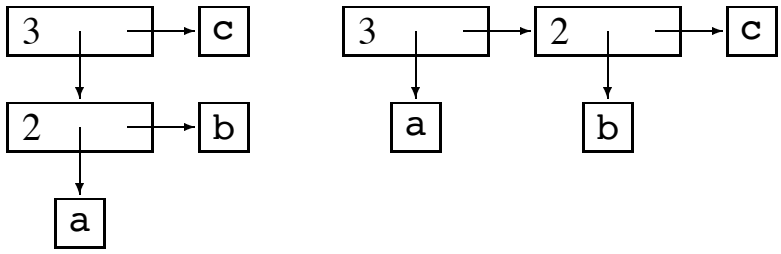
A list is represented as a binary tree. Each node in the tree is either a branch node or a leaf node. Each leaf node contains an element of the list. Each branch node contains two pointers, the left one points to the first part of the list, while the right points to the second part of the list. Ideally these two parts of the list should be approximately equal in length (i.e. the tree should be be balanced), but that is not a requirement for correctness of the representation. It affects only the performance. Each branch node also contains the number of items in its sub-tree (i.e. the length of the list which that sub-tree represents). Again, this is solely to improve performance (so that the length primitive can be computed in constant time).

The representations of the empty list (`[ ]`), a singleton list (`[a]`), and a list of two elements (`[a,b]`) are:



The • denotes a nil pointer and occurs only in lists containing no elements, or just a single element.

Two alternative representations of the list `[a,b,c]` are:



The more elements the list contains, the more different tree structures are possible. All are equally valid and will give exactly the same results, although the performance of a program may depend upon how well-balanced the tree is.

With the above representation, it is easy to program implementations of all six primitive functions that will execute in constant time, irrespective of the lengths of the lists involved. None of these primitives offers any scope for parallelism, however. That comes with the implementation of higher-level list-processing functions.

3 High-Level List-Processing Functions

Most common high-level list-processing functions such as `map` and `reduce` can be easily programmed in terms of the new primitives, using a divide-and-conquer strategy. The structure of divide-and-conquer programs makes them particularly well suited to parallel implementations on a very wide variety of parallel architectures[1, 2].

Most of the functions defined in Section 3.2 are identical in specification to functions in the Haskell standard prelude[6]. The Haskell standard prelude has been chosen as a starting point because it includes a wide variety of useful list-processing functions that are in common use in Haskell and many other functional languages (the standard functions of Miranda, for instance, are very similar). Nearly all of the list-processing functions in the Haskell standard prelude have been included. A few have been omitted and a few have been replaced by similar (but not identical) functions for reasons discussed later.

3.1 The Language Used

All program fragments are expressed in a simple functional language pseudocode which is essentially a very small subset of Miranda and Haskell, but with a few minor syntactic changes. The usual arithmetic operators are used, the relational operators that test for equality and inequality are denoted by == and `!=` (as in C) and a number of other relational and logical operators are borrowed from C also. Conditional expressions are denoted by `IF..THEN..ELSE..FI`. Each program line begins with the symbol > and all other lines are regarded as comments.

As the code for each of the functions is quite brief and easy to understand, the code itself serves as both a formal specification of the function and its implementation. No attempt is made to include separate formal specifications of the functions. The comments preceding the code give a brief informal description of each function, and most of these functions are familiar to programmers of functional languages anyway.

Pattern matching is often used in modern functional programming languages to make programs more readable. It is easy to incorporate the new list primitives into the patterns that can be used. For example, instead of writing the definition of the function `head` as:

```
> head s =
>     IF #s==0 THEN ⊥
>     ELSIF #s==1 THEN element s
>     ELSE head s1 FI
>     WHERE (s1,s2) = split s
```

The same program can be written using pattern matching as:

```
> head [] = ⊥
> head [x] = x
> head (s++t) = head s
```

These two programs would be executed in almost exactly the same way, but the latter form is shorter and clearer. Pattern matching is used throughout this paper for those reasons. Undefined parts of functions will be omitted completely instead of making them explicit with the symbol ⊥.

3.2 Code for the Functions

`head` and `tail` extract the first element and the remaining sub-list, respectively, from a non-empty list. `last` and `init` are the dual functions, extracting the last element and the preceding sub-list.

```
> head [x] = x
> head (s++t) = head s

> last [x] = x
> last (s++t) = last t

> tail [x] = []
> tail (s++t) = tail s ++ t

> init [x] = []
> init (s++t) = s ++ init t
```

`(:)` adds a new element to the beginning of a list.
`append` adds a new element to the end of a list:

```
> x : s = [x] ++ s

> append s x = s ++ [x]
```

`(..)` creates a list made up of a sequence of consecutive integers:

```
> i .. i = [i]
> i .. j = IF i < j THEN (i..mid) ++ ((mid+1)..j) FI
>     WHERE mid = (i + j) DIV 2
```

(`DIV` is integer division)

`s!!i` is the i-th element in the list `s` (counting from 0):

```
> [x] !! 0 = x
> (s++t) !! i = IF i < #s THEN s !! i
>     ELSE t !! (i - #s) FI
```

`balance` is the identity function on lists, but has the useful effect of creating a balanced representation:

```
> balance s = map f (0..(#s-1)) WHERE f i = s!!i
```

`map f s` applies `f` to each element of `s` independently:

```
> map f [] = []
> map f [x] = [f x]
> map f (s++t) = map f s ++ map f t
```

`filter p s` is the list of all those elements of `s` which satisfy the predicate `p`:

```
> filter p [] = []
> filter p [x] = IF p x THEN [x] ELSE [] FI
> filter p (s++t) = filter p s ++ filter p t
```

`partition p s` is the pair of lists such that the first is all elements of `s` satisfying `p`, while the second is all elements of `s` which do not satisfy `p`:

```
> partition p s = (filter p s, filter (not.p) s)
```

`reduce f z s` reduces the list `s`, using the binary operator `f`, and the starting value `z`; while `reduce1` is a variant with no starting value, that must be applied to non-empty lists. The function `f` must be associative for `reduce f` and `reduce1 f` to give deterministic results. `reduce` and `reduce1` replace the Haskell functions `foldl, foldl1, foldr` and `foldr1`.

```
> reduce f z [] = z
> reduce f z [x] = x
> reduce f z (s++t) = f (reduce f z s) (reduce f z t)

> reduce1 f [x] = x
> reduce1 f (s++t) = f (reduce1 f s) (reduce1 f t)
```

`reducemap` is simply the functional composition of `reduce` and `map`:

```
> reducemap f g z s = reduce f z (map g s)
```

`concat`, when applied to a list of lists, gives a single list which is the concatenation of all the element lists:

```
> concat = reduce (++) []
```

`take i s` returns the first `i` elements of `s` (or the whole of `s` if `i` is greater than `#s`), where `i`≥ 0:

```
> take 0 s = []
> take i [] = []
> take i [x] = [x]
> take i (s++t) = IF i <= #s THEN take i s
>     ELSE s ++ take (i - #s) t  FI
```

`drop i s` returns all but the first `i` elements of `s`, where `i`≥ 0:

```
> drop 0 s = s
> drop i [] = []
> drop i [x] = []
> drop i (s++t) = IF i <= #s THEN drop i s ++ t
>     ELSE drop (i - #s) t FI
```

`splitAt i s` does both jobs at once:

```
> splitAt i s = (take i s, drop i s)
```

`takeWhile p s` returns the longest prefix of `s` containing elements satisfying the predicate `p`:

```
> takeWhile = first . span WHERE first (x,y) = x
```

`dropWhile p s` returns the remainder of `s`:

```
> dropWhile = second . span WHERE second (x,y) = y
```

`span p s` is equivalent to `(takeWhile p s, dropWhile p s)`:

```
> span p [] = ([],[])
> span p [x] = IF p x THEN ([x],[]) ELSE ([],[x]) FI
> span p (s++t) = IF #s2 == 0 THEN (s1++t1, t2) ELSE (s1, s2++t) FI
>     WHERE
>     (s1,s2) = span p s
>     (t1,t2) = span p t
```

`break p s` is similar but uses the negation of `p`:

```
> break p = span (not . p)
```

`lines s` breaks the string `s` at each newline character (which is removed) and returns a list of separate lines. A string consisting of a single newline character gives a list of two empty lines. Any string that terminates with a newline character will give a list of lines in which the last line is empty. A string containing no newline characters will give a list of lines containing only one line (which is the original string exactly). The third line of the code below is rather subtle, and best understood by considering the situation in which the last character of `s` is not a newline, nor is the first character of `t`. In that case, the last line of `s` and the first line of `t` are just two parts of the same line of `s++t`, hence the need for the term `[last ss ++ head tt]` which concatenates the last line of `s` and the first line of `t` into a single line. Notice that the other occurrences of the operator ++ in the definition of `lines(s++t)` concatenate *lists of* lines.

```
> lines [] = [[]]
> lines [x] = IF x == newline THEN [[],[]] ELSE [[x]] FI
> lines (s++t) = init ss ++ [(last ss) ++ (head tt)] ++ tail tt
>     WHERE
>     ss = lines s
>     tt = lines t
```

`words s` acts similarly, but splits the string at every occurrence of white space (which is removed), returning a list of words.

```
> words [] = [[]]
> words [x] = IF isSpace x THEN [[],[]] ELSE [[x]] FI
> words (s++t) = init ss ++ join (last ss) (head tt) ++ tail tt
>     WHERE
>     join p q = IF #p==0 && #q==0 THEN [] ELSE [p++q] FI
>     ss = words s
>     tt = words t
```

`unlines` and `unwords` perform the inverse operations:

```
> unlines s = concat (map f s) WHERE f p = p ++ [newline]
> unwords s = concat (map f s) WHERE f p = p ++ [space]
```

`nub s` returns the list consisting of the elements of `s` with all repeated occurrences of the same element removed:

```
> nub [] = []
> nub s = (head s) : nub (filter ((!=)(head s)) (tail s))
```

`reverse s` returns the list got by reversing the order of the elements in `s`:

```
> reverse = reduce (flip (++)) []
>     WHERE flip f x y = f y x
```

`and s` returns the result of logically ANDing together all the elements of `s`.
`or s` performs the similar logical OR operation:

```
> and = reduce (&&) True
> or = reduce (||) False
```

(`&&` denotes the logical AND operator, while `||` denotes logical OR)

`any p s` is true if and only if at least one element of `s` satisfies the predicate `p`.
`all p s` is true if every element of `s` satisfies the predicate `p`:

```
> any p = reducemap (||) p False
> all p = reducemap (&&) p True
```

`elem x s` is true if and only if `x` is an element of `s`.
`notElem x s` is true if and only if `x` is not an element of `s`:

```
> elem = any . (==)
> notElem = all . (!=)
```

where `(.)` denotes function composition: `(f.g)x = f(g x)`

`sum s` returns the sum of all elements of `s`.
`product s` returns the product of the elements.
`maximum s` is the maximum value of the elements.
`minimum s` is the minimum value of the elements:

```
> sum = reduce (+) 0
> product = reduce (*) 1
> maximum = reduce1 max
> minimum = reduce1 min
```

`zip` combines two lists to create a list of pairs:

```
    zip [x1,x2,...] [y1,y2,...] = [(x1,y1), (x2,y2), ...]
```

`zip3` combines three lists to create a list of triples in a similar way:

```
> zip = zipWith f WHERE f a b = (a,b)
> zip3 = zipWith3 f WHERE f a b c = (a,b,c)
```

`zipWith` is a generalisation of `zip` in which corresponding elements are combined using any given function:

```
zipWith f [x1,x2,...] [y1,y2,...] = [f x1 y1, f x2 y2, ...]
```

`zipWith3` combines three lists in a similar way:

```
> zipWith f s t = map g (0 .. (n-1))
>     WHERE
>     g i = f (s!!i) (t!!i)
>     n = min (#s) (#t)

> zipWith3 f s t u = map g (0 .. (n-1))
>     WHERE
>     g i = f (s!!i) (t!!i) (u!!i)
>     n = min (#s) (min (#t) (#u))
```

`transpose`, when applied to a list of lists (interpreted as a list of rows of a matrix), gives that list with rows and columns interchanged:

```
> transpose [] = []
> transpose [[x]] = [[x]]
> transpose [s++t] = transpose s ++ transpose t
> transpose (s++t) = zipWith (++) (transpose s) (transpose t)
```

3.3 Discussion

We set out to implement the full set of list-processing functions defined in the Haskell standard prelude to see if the new list primitives can cope adequately with a wide range of common programming problems. The set of functions included in the previous section is close to the set of functions in the Haskell standard prelude, but some functions have been omitted and some others added. The reasons for these changes are discussed below.

The first six functions (`head, last, tail, init, (:), append`) are the primitives of the traditional model of lists and their duals (operating on the other end of the list). The implementation of each is straightforward. If the list argument of each is balanced, then the result will be very nearly balanced also. However, repeated application of these functions can easily produce unbalanced lists. For example, if `s` is balanced, then `tail(tail(tail(tail(tail s))))` will be badly unbalanced. For this reason, the use of these functions should be avoided if at all possible. Often, this will not be a problem as many things can be done in other ways which do not use these low-level list functions (examples later).

The function (..) which creates a list of consecutive integers is not in the Haskell standard prelude, but is included here because it is used several times in the code for later functions. It has the advantage that the code given always creates balanced lists, and the performance of most of the functions in this list is best for balanced lists, particularly for parallel implementations, as we will see later.

The function `balance` is functionally equivalent to the identity function, but the code given creates a balanced representation no matter how unbalanced the input. Of course, this function does not occur in the Haskell standard prelude as it serves no purpose with the traditional representation of lists.

The next four functions (`(!!), map, filter, partition`) are all functionally the same as in Haskell. Our implementation of each of them uses the divide-and-conquer paradigm, in each case splitting the list into two parts and calling the function recursively on each part, then combining the two results (in the case of `(!!)` alone, only one of the two parts needs to be solved as the other is not required).

The two functions `reduce` and `reduce1` are introduced to replace the Haskell functions `foldl, foldr, foldl1` and `foldr1` which perform the same reduction but in a defined order, rather than in an undefined order as do `reduce` and `reduce1`. This means that both `reduce` and `reduce1` are non-deterministic (i.e. they may give non-unique results) unless the first argument, `f`, is associative. In effect, this means that there is a proof obligation on the programmer to show that `f` is associative, otherwise the program may give unpredictable results. Although programmers are not used to having to cope with such proof obligations, they are generally not too onerous and will become much easier and more generally accepted as formal methods and program verification techniques become more widely used and better supported by appropriate software tools.

The payoff is considerable: `reduce` and `reduce1` are much easier to implement efficiently in parallel than the `fold`–family of functions. The important question, however, is: are they as useful? It is very difficult to find a sound answer to this question short of acquiring years of experience programming with them in a wide variety of applications. The best answer we can give here is to look at all the other functions in the Haskell standard prelude that are programmed using the `fold`–family. There are, in fact, 17 such functions. All but 3 of these are easily programmed using `reduce` instead (because, in each case, the reduction function is associative). This suggests that in the great majority of cases `reduce` is a convenient replacement for `foldl` and `foldr`. The three functions which are not easily programmed using `reduce` are `(\\), sums` and `products`, all of which require that the list elements be scanned in order from left to right. These three functions have been omitted from the previous section because we have not found any better way of implementing them than that given in the Haskell standard prelude, which is serial only.

The Haskell functions `scanl, scanl1, scanr` and `scanr1`, which scan lists either from the left or from the right, are omitted for the same reason.

The function `reducemap` has been introduced as a useful combination of `reduce` and `map`, although it is not included in Haskell.

The functions `concat, take, drop, splitAt, takeWhile, dropWhile, span, break, lines, words, unlines, unwords, nub, reverse, and, or, any, all, elem, notElem, sum, product, maximum, minimum, zipWith, zipWith3, zip, zip3` and `transpose` are all functionally equivalent to the Haskell functions of the same name. In all these cases a simple and efficient divide-and-conquer implementation is possible. The Haskell functions `zip4, zip5,` etc. and `zipWith4, zipWith5,` etc. have been omitted purely to save space. All are very similar to the zip functions that have been included.

4 Parallel Performance

4.1 Parallel Implementation

The most obvious way of implementing the divide-and-conquer style of programming in parallel is to assume a shared memory architecture. In theoretical analysis this usually means

the PRAM model (Parallel Random Access Machine) as it is commonly called in the literature (e.g. [5]). In such a shared memory architecture, after the divide-and-conquer algorithm has divided the problem into a number of independent sub-problems, these subproblems can run concurrently. There is no added overhead for data transmission between processors as all processors can access the shared memory to obtain their input data, while the results written into shared memory are also available (at no added cost) to the processes which need to use them. The performance analysis given in the next section is based on this approach.

A distributed implementation (on an architecture without shared memory) is less easily achieved. Possible approaches are being investigated, but it is too soon to say how successful these will be.

4.2 Theoretical PRAM Performance

Complexity analysis of the programs given in Section 3.2 gives the results shown in the table. The analysis is for asymptotic performance for large n, where n is the problem size, generally the length of the list which is one of the arguments of the function concerned. In the case of the last function, `transpose`, the matrix being transposed is assumed to be $n \times n$. The parallel performance figures are all for the CRCW (Concurrent Read, Concurrent Write) PRAM model.

The first column (after the function itself) is the parallel time for balanced lists. The second column is the parallel time for the worst case (usually for maximally unbalanced lists). The third column gives the number of processors required to achieve maximum parallelism. The fourth column gives the serial time for balanced lists, while the fifth column is the serial time for maximally unbalanced lists. Finally, the last column is the serial time for the conventional head-tail-cons model of lists (as programmed in the Haskell standard prelude, for instance).

Table of Complexity Analysis Results

	Concatenation Model					Cons
	Parallel			Serial		Serial
Function	Balanced	Worst	Procs.	Balanced	Worst	
`s++t`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(n)*
`split s`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(n)*
`#s`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(n)*
`head s`	*O*(log *n*)	*O(n)*	*O(1)*	*O*(log *n*)	*O(n)*	*O(1)*
`last s`	*O*(log *n*)	*O(n)*	*O(1)*	*O*(log *n*)	*O(n)*	*O(n)*
`tail s`	*O*(log *n*)	*O(n)*	*O(1)*	*O*(log *n*)	*O(n)*	*O(1)*
`init s`	*O*(log *n*)	*O(n)*	*O(1)*	*O*(log *n*)	*O(n)*	*O(n)*
`cons x s`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`append s x`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(n)*
`1..n`	*O*(log *n*)	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`balance s`	*O*(log *n*)	*O(n)*	*O(n)*	*O*(*n* log *n*)	*O*(n^2)	—
`s!!i`	*O*(log *n*)	*O(n)*	*O(1)*	*O*(log *n*)	*O(n)*	*O(i)*
`map f s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`filter p s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`partition p s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`reduce f z s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`reduce1 f s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`reducemap f z s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`concat s`	*O*(log *n*)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*

`take i s`	$O(\log n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(i)$
`drop i s`	$O(\log n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(i)$
`splitAt i s`	$O(\log n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(i)$
`takeWhile p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`dropWhile p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`span p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`break p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`lines s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`words s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`unlines s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`unwords s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`nub s`	$O(n \log n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
`reverse s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`and s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`or o`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`any p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`all p s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`elem x s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`notElem x s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`sum s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`product s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

`maximum s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`minimum s`	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`zipWith f s t`	$O(\log n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
`zipWith3 f s t u`	$O(\log n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
`zip s t`	$O(\log n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
`zip3 s t u`	$O(\log n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
`transpose s`	$O(\log^2 n)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n^2)$

It is clear from the table that most of the functions considered fall into one of a small number of categories.

The first category includes the primitives and other very simple functions: `(++), split, #, head, last, tail, init, cons, append`. None of these offer any parallelism. Some are faster than the equivalents in the traditional model of lists, while others are slower. On average they are faster for balanced lists and about the same speed or slower for unbalanced lists.

The second main category includes the functions `map, filter, partition, reduce, reduce1, reducemap, concat, takeWhile, dropWhile, span, break, lines, words, unlines, unwords, reverse, and, or, any, all, elem, notElem, sum, product, maximum` and `minimum`. All of these execute in $O(\log n)$ parallel time for balanced lists and $O(n)$ time for serial execution and for parallel execution of worst-case unbalanced lists. Provided the program is written so that the lists are always approximately balanced, all of these functions offer excellent speedup.

A third category is the 'zip' functions: `zip, zip3, zipWith, zipWith3`. These all have $O(\log n)$ parallel time for balanced lists, $O(n)$ parallel time for worst-case unbalanced lists and for serial execution with the usual head-tail-cons list primitives. For serial execution with the concatenate-split list primitives, however, these functions are slower.

The remaining functions need to be considered separately. `nub` and `transpose` are more complex than most, but both offer excellent parallel speedup for balanced lists. The function `(..)` always produces a balanced list as its output and hence always gives excellent parallel speedup. The function `balance` has no counterpart in the head-tail-cons model, so the comparison cannot be made (although it gives good parallel speedup within the context of the concatenate-split model of lists).

Overall, most of these functions offer very substantial parallel speedups provided that the list representations are reasonably well balanced. This is very dependent on the algorithm design; some functions may cause lists to become unbalanced, while others do not. Much more experience of programming with this model of lists is needed before we can say just how easy it is to write programs which offer large amounts of parallelism in a wide range of typical applications.

Nevertheless, in many common applications highly parallel programs *are* easy to

construct. In a later section, we consider three very different applications: sorting, ray tracing, and finding the convex hull of a set of points. All three can be easily programmed in the new model in ways which are potentially highly parallel.

4.3 Parallel Simulation

The model of list processing has been implemented within a functional programming language and the performance of a simulated parallel implementation investigated. The parallel simulation is for a shared memory architecture that falls within the CRCW PRAM category. The implementation method is combinator graph reduction.

Input to the reducer is the low-level functional language, FLIC [14], which is first of all translated to an acyclic graph representing an equivalent lambda-expression. This graph also contains primitives (integers, reals and sum-products) and operators (which broadly correspond to the FLIC set of operators), together with the Y combinator to indicate recursion.

The standard abstraction algorithm [17] is then performed to translate the graph into one containing the Turner combinators (S, K, I, B, C, S′, B′, C′) and no lambdas. Acyclic graph reduction is then used to evaluate the resulting graph.

The graph is assumed to reside in shared memory so that all available processors can work on reducing different parts of it simultaneously. The performance results are entirely consistent with the analysis given earlier. Details of this simulation work are published elsewhere[7].

5 Three Applications

Three different applications are outlined below to illustrate the use of the model in somewhat more complex situations than those considered previously.

5.1 Quicksort

Hoare's quicksort algorithm can be programmed quite easily in the new model. For simplicity, we assume that the input data is in the form of a list of distinct numbers which are required to be sorted into increasing numerical order. Also for simplicity, a rather crude method is used to estimate the median value. Neither of these simplifications affects the structure of the program for our purposes. A much better and more general quicksort program could be written with exactly the same overall structure, but it would be longer and offer no new insights into the list-processing aspects of the problem.

Our simple version of quicksort is:

```
> quicksort [] = []
> quicksort [x] = [x]
> quicksort s = quicksort s1 ++ quicksort s2
>     WHERE
>     (s1,s2) = reducemap f g [] s
>     g x = IF x <= median THEN ([x],[])
>           ELSE ([],[x]) FI
>     f (t1,t2) (u1,u2) = (t1++u1, t2++u2)
>     median = (head s + last s)/2
```

The use of `reducemap` is safely deterministic because the function `f` is associative (by the associativity of `++`).

The PRAM performance of this program is $O(\log^2 n)$ because there are $\log n$ stages to the whole program and each requires the execution of `reducemap` which is $O(\log n)$. This assumes that the list is well balanced initially and that the estimates of the median are always exactly right. While this is unlikely to be precisely the case, for typical data quicksort behaves nearly as well as for the ideal case. Worst case situations are extremely unlikely to occur for large data sets. As the programmer has full control of the way in which the input list is generated, it should be easy to ensure that it is balanced, and hence optimum parallel performance is attained.

5.2 Ray Tracing

Suppose we have a list of objects and a list of rays (representing physical objects and light rays in three dimensional space). The problem is to compute the first impact of each ray on an object.

Each ray is represented as a starting point and a direction. An impact with an object is simply represented as a distance, which is the distance the ray travels from its starting point before it hits the object. The first impact for a given ray is then the impact represented by the minimum distance. We do not go into the details of how objects are represented, or how the point of impact of a ray with an object is computed, but concentrate solely on the overall structure of the program.

One possible solution is as follows, expressed as a program for the function *impacts* which takes two arguments, a list of rays and a list of objects, and gives a list of impacts as its result (this list is in the same order as the input list of rays):

```
> impacts rays objects = map firstimpact rays
>     WHERE
>     firstimpact ray = reducemap min impact infinity objects
>         WHERE
>         impact object =
                Distance travelled by ray to hit object
>         min x y = IF x<=y THEN x ELSE y FI
```

The program uses `map` to apply the function `firstimpact` to each separate ray in the list. This function takes a single ray as its argument and finds the first impact of that ray with any of the objects. This is done by first applying `impact` to each separate object. The function `impact` takes a single object as its argument and finds the impact of the current ray with that object. When the list of impacts of one ray with all the objects has been found, that list is reduced with `min` to find the first impact of that ray on an object. The value `infinity` denotes a very large value which is used to represent no impact at all (i.e. the distance travelled by the ray is infinite).

An alternative approach is to first find the the list of impacts of all rays with a single object, and do this independently for each of the objects. These results can then be combined to give the first impact of each ray with an object. This program is:

```
> impacts2 rays objects = reducemap (zipWith min) impactson z objects
>     WHERE
```

```
>     impactson object = map impactby rays
>         WHERE
>         impactby ray =
                Distance travelled by ray to hit object
>     min x y = IF x<=y THEN x ELSE y FI
>     z = map noimpact rays
>         WHERE noimpact ray = infinity
```

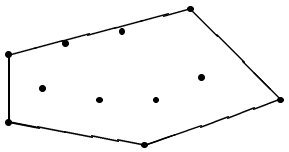
This program is a little more difficult to understand. The function `reducemap` firstly applies `impactson` to each object in the list of objects. This gives a list of lists, one for each object. Each inner list contains the impacts of all rays with that object. These lists of impacts are then combined by finding the first impact for each ray, i.e. the minimum impact distance. The value of `z`, the 'zero' argument of `reducemap` is the result if the list of objects is empty; in which case no ray hits anything, so `z` is simply a list containing the value `infinity` repeated as many times as there are rays.

The first program gives PRAM performance of $O(\log m) + O(\log n) = O(\log mn)$ for balanced lists, where m is the number of rays and n is the number of objects. The term $O(\log m)$ is for the execution of `map` over `rays`, while the term $O(\log n)$ is for the execution of `reducemap` over `objects`. For optimum performance, the programmer must ensure that balanced representations are generated for both the list of rays and the list of objects. This is easy to do and it is hence quite reasonable to assume that the lists are balanced.

The second program gives poorer performance. There are $\log n$ stages to the `reducemap` function in the first line, but each involves the execution of `zipWith` which is $O(\log m)$ (as the lists which `zipWith` operates on contain one element for each ray). So the overall performance is $O(\log m \times \log n)$, which is less good than for the first program.

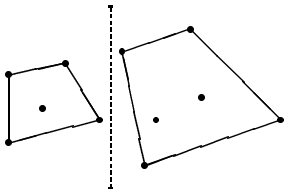
5.3 Convex Hull

The convex hull of a set of points in a plane is the smallest enclosing convex polygon. For example, the set of eleven points shown in the diagram has the convex hull indicated:



A program for finding the convex hull of a set of points can be written using a divide-and-conquer algorithm, using the fact that it is relatively easy to combine two non-overlapping convex polygons into a single convex polygon which encloses the original two[15]. (This involves much less work than combining two overlapping convex polygons.) We can take advantage of this if we divide the original set of points into non-overlapping subsets. An easy way to do this is to order the points in order of their X-coordinates (and if two points have equal X-coordinates, they are ordered on their Y-coordinates). If this ordered list of points is now divided into sublists, each sublist will have a convex hull which does not overlap the convex hull of any other sublist.

For the example given above, the points can be divided into two sets with convex hulls as shown:



A suitable program to do this is shown below. The data is assumed to be in the form of a list of points (called `points`) which have already been ordered in the required manner.

```
> convexhull points = reducemap combine g z points
>     WHERE
>     combine hull1 hull2 =
          Convex hull enclosing hull1 and hull2
>     g point =
          Polygon consisting of that one point
>     z =
          The empty polygon
```

We can easily represent a polygon as an ordered list of points (its vertices) and it is easy to write programs for `g` and `z`. The function `combine` which combines two convex hulls is harder, but the details do not really matter here, as we have sufficient of the structure of the program to show the overall structure and the potential for parallel implementation.

A more efficient program is likely to result if we combine the sorting and the convex hull computation into a single application of divide-and-conquer. This is easy to do, using the quicksort algorithm for the sorting:

```
> convexhull points = polygon
>     WHERE
>     polygon =
>         IF #points == 1 THEN points
>         ELSE combine (convexhull s1) (convexhull s2) FI
>         WHERE
>         (s1,s2) = partition p points
>         p x = (x<=median)
>     median =
          Estimate the median value of the elements of points
>     combine hull1 hull2 =
          Convex hull enclosing hull1 and hull2
```

In this single application of the divide and conquer paradigm, the divide phase effectively does the sorting, while the final recombination phase does the convex hull computation.

Assuming that the two recursive applications (i.e. `convexhull s1` and `convexhull s2`) are always done in parallel, the complete program has $O(\log n)$ stages, each of which requires the execution of `partition` over a list which is $O(n)$ in length, taking $O(\log n)$ time in parallel (assuming the list of points is balanced, which is easy to arrange). Thus, the total time is $O(\log^2 n)$ in the PRAM model.

6 Related Work

It is well known that ordered data (i.e. lists) can be represented by tree structures (e.g. in tree sorting), and also that the divide-and-conquer paradigm applies naturally to tree structures, as well as being very suitable for parallel implementation. Recently, there has been increasing interest in using the divide-and-conquer approach as a basis for the parallel implementation of functional languages[1, 4, 8, 12, 16]. Yet these pieces have not been brought together before in the way proposed in this paper, to suggest an alternative model for lists as basic data structures for parallel (and architecture-independent) programming languages.

George Mou has adopted an alternative approach in his language Divacon[11], in which arrays are used as the basic data structures, but with additional primitives to support the divide-and-conquer style of programming. In Divacon, the primitive operations on arrays are designed primarily to support a distributed representation on the Connection Machine, and the representation does not make use of binary trees.

7 Conclusions

The proposed new model of list processing overcomes one of the major obstacles to achieving good parallel implementations of functional programming languages. It supports a style of programming in which most common list operations can be implemented using divide-and-conquer algorithms which are easily and efficiently parallelised, unlike the conventional style of list processing in which list operations are usually defined to step through the list in strictly sequential order.

For example, the conventional approach to summing a list of numbers is illustrated by the program:

```
> sum [] = 0
> sum (x:s) = x + sum s
```

while the equivalent program in our new model is:

```
> sum [] = 0
> sum [x] = x
> sum (s++t) = sum x + sum t
```

The latter program takes advantage of the associativity of the addition operator, so that the precise order in which the additions are performed does not matter.

At an even higher level, if the function `reduce` is regarded as a basic primitive, `sum` can be programmed in a yet more general form:

```
> sum = reduce (+) 0 s
```

which presupposes no particular model of lists whatsoever and gives even more freedom to the implementor to optimise to suit the architecture on which the program will be run.

Although all three programs above can be written in most common functional languages (and other languages which support general data structures) by simply defining the required data structures from first principles, this will not give acceptable levels of efficiency. Typically, operations on user defined data structures are an order of magnitude slower than the equivalent operations on built-in data structures. This is a very powerful incentive to the programmer

to use the built-in data structures whenever possible. Most programmers would never even comtemplate replacing the built-in model of lists in languages such as Lisp, ML, Miranda or Haskell with their own user-defined model of lists.

With parallel implementation, this difference in efficiency is likely to be even greater, so it is important that the basic model of lists and/or other data structures used in the language is capable of efficient implementation on all the architectures for which the language will be used.

A new generation of much more architecture-independent programming languages is needed in which more suitable models are provided for basic data structures, such as the concatenation model of lists proposed here. This may be either instead of, or in addition to, the traditional head-tail-cons model of lists. Furthermore, languages should be designed to encourage programmers to use even higher-level primitives, such as `reduce`, which do not specify any particular model and so are even more implementation (and hence architecture) independent.

References

[1] T. Axford, An Elementary Language Construct for Parallel Programming, *ACM SIGPLAN Notices* **25**(7) (1990) 72–80.

[2] T. Axford, An Abstract Model for Parallel Programming, Research Report CSR-91-5, School of Computer Science, University of Birmingham, 1991.

[3] R. Bird and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, London, 1988).

[4] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, (Pitman, 1989).

[5] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, (Cambridge University Press, Cambridge, 1988).

[6] P. Hudak and P. Wadler (eds.), Report on the Programming Language Haskell, Internal Report, Department of Computer Science, Yale University, 1990.

[7] M. Joy and T. Axford, Parallel Combinator Reduction: Some Performance Bounds, Research Report 210 (1992), Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K.

[8] P. Kelly, *Functional Programming of Loosely-Coupled Multiprocessors*, (Pitman, 1989).

[9] J. McCarthy, A Micro-Manual for Lisp – Not the Whole Truth, *ACM SIGPLAN Notices* **13**(8) (1978) 215–216.

[10] J. McCarthy, History of Lisp, *ACM SIGPLAN Notices* **13**(8) (1978) 217–223.

[11] Z.G. Mou, Divacon: A Parallel Language for Scientific Computation Based on Divide-and-Conquer, in *Proc. 3rd Symp. Frontiers Massively Parallel Computation* (IEEE, 1990).

[12] Z.G. Mou and P. Hudak, An Algebraic Model for Divide-and-Conquer and Its Parallelism, *J. Supercomputing*, **2** (1988) 257–78.

[13] S.L. Peyton Jones, Parallel Implementations of Functional Programming Languages, *Computer J.* **32**(2) (1989) 175–186.

[14] S.L. Peyton Jones and M.S. Joy, FLIC – a Functional Language Intermediate Code, Research Report 148, Department of Computer Science, University of Warwick, Coventry, 1989.

[15] F.P. Preparata and S.J. Hong, Convex Hulls of Finite Sets of Points in Two and Three Dimensions, *Comm. ACM*, **20**(2) (1977) 87–93.

[16] F.A. Rabhi and G.A. Manson, Experimenting with Divide-and-Conquer Algorithms on a Parallel Graph Reduction Machine, Research Report CS-90-2 (1990), Department of Computer Science, University of Sheffield, Sheffield S10 2TN, U.K.

[17] D.A. Turner, A New Implementation Technique for Applicative Languages, *Software – Practice and Experience*, **9** (1979) 31–49.

[18] D.A. Turner, Recursion Equations as a Programming Language, in *Functional Programming and its Applications* (eds. J. Darlington, P. Henderson and D.A. Turner), (Cambridge University Press, 1982).

[19] D.A. Turner, Miranda: A Non-Strict Functional Language with Polymorphic Types, in Proceedings of FPCA'89 (ed. J-P. Jouannaud), *Lecture Notes in Computer Science* **201**, 1–16, (Springer-Verlag, Berlin, 1985).

[20] A. Wikstrom, *Functional Programming Using Standard ML*, (Prentice-Hall, London, 1987).