Programming with Higher Order Functions

Andreas Maaßen
Lehrstuhl für Informatik II, RWTH Aachen
Ahornstraße 55, D-5100 Aachen

July 1989

Abstract

Recursion is the most important control structure in applicative languages, but the extensive use of recursive definitions tends to spoil readability and understandability of programs. Besides, many programmers don't like to use recursion. We suggest a choice of higher order functions which enable programmers to write concise and (almost) recursion-free functional programs. Our approach is heuristical: we developed our higher order functions by observing what kind of programming patterns appeared repeatedly when we wrote functional programs for a number of problems. Most of our higher order functions are tightly related to data structures. We suggest a choice of problem-oriented applicative data structures and discuss aspects of an efficient implementation.

1 Introduction

The possibility to write concise and readable programs in functional languages is one of the most often cited advantages of this class of programming languages. Much of this advantage is due to the existence of higher order functions which enable the programmer to define powerful control structures of his own, according to the kind of problem he wants to solve. It is particularly this feature of functional programming languages that makes them satisfy J. Backus demand that programming languages should "encourage us to think in terms of the larger conceptual units of the task at hand" [3].

We have observed that there are two major odds against understandability of functional programs:

- Most languages lack problem-oriented data structures. Lists, which are the standard data structure in most applicative languages, are easy to implement, but quite insufficient when it comes to writing programs that reveal the logical structure of the problems to be solved. We need data structures which are as "high" and comfortable as the other language constructs of functional languages.

- Most applicative languages lack a reasonable set of predefined higher order functions, particularly higher order functions on data structures. The programmer is left alone with recursion as the most important control structure, which discourages many programmers from the very beginning. The others tend to use recursive function definitions extensively, which spoils readability.

It is the intention of this paper to demonstrate that a sensible choice of data structures with a corresponding set of higher order functions enables us to write very concise and (almost) recursion-free functional programs for a lot of applications. We believe that these language constructs should be offered to the programmer by any functional programming system. If they are not part of the language, they should at least be present as a library module.

Most of the higher order functions presented in this paper are defined on data structures. We try to avoid making particular assumptions about the characteristics and the implementation of these objects. In general, we treat them as abstract data types whose internal structure is unknown. This implies that we do not make use of the well known applicative data structure *List* with its constructors *Cons* and *Nil*, because this is no abstract data type.

Throughout this paper, we use the syntax of L. Cardelli's version of ML [6], [7] for the representation of programs. (A brief explanation of some aspects of ML - syntax can be found in the appendix.) This does not mean that everything we present in this paper can be programmed in ML straightforward. An implementation of these language constructs in Miranda[1] is in preparation, so we shall be able to do more practical research on this topic soon.

The remainder of this paper consists of 6 sections. In section 2, we present our data structures and higher order functions. In sections 3, 4, and 5, we explain how we developed functional programs for a number of non-trivial standard problems. We examine some sorting algorithms, the subset construction to generate a deterministic finite automaton, and a program which tests a context-free grammar for LL(1)-property. In section 6, we discuss some implementation aspects, and section 7 summarizes out observations.

2 Higher Order Functions on Data Structures

In this section we introduce the abstract data types and the primitives used in the following sections.

2.1 Data Structures

There will be three different data structures, each of them representing characteristical properties of three distinct classes of data structures which exist in most

[1]Miranda is a trademark of Research Software Ltd.

programming languages:

- sequence:
  Sequences are list-like structures with an implicit indexing. The index of an entry is a natural number, namely its position in the sequence. Inserting an element into a sequence at any position increments the indices of all subsequent elements.

- table:
  Tables are array-like structures with explicit indexing. As indices we permit any data type whose elements can be tested for equality effectively, hence no function types are permitted as index types. We choose an incremental approach for the generation of tables. This means that we start from empty structures which are filled out (updated) incrementally. We do not require to name the maximum index range for a table when generating an empty structure, instead we permit to extend the index range dynamically by using new indices. Of course, new indices must be of the same type as the old ones.

  This approach to array-like structures is more liberal than what is found in most programming languages, and we admit that we do not know an elegant and satisfying way how to translate constructions like, e. g. , tables with tables as indices into conventional programming languages. Nevertheless, there are ways how to do this, e. g. via hash tables which contain natural numbers as substitute indices for tables we wish to use as logical indices. We think that the concept of array-like structures with arbitrary index types is very helpful and important for the design of programs which reveal the true logical and mathematical structure of the algorithms we want to encode. Especially for applicative languages it is important that the data structures have the same expressive power as the rest of the language. This paper contains examples which support this claim.

- set:
  Sets – as their name indicates – represent indexless structures. The most important element in this class is the set in the mathematical sense, but there are other ones, e. g. multisets (Bird [4] calls them *bags*). Only types that can be tested for equality effectively may be used as base types for sets.

For each of these structures we assume that the following primitives are provided by the system (with the wild card '*' replaced by 'seq', 'tab', and 'set' for sequences, tables, and sets, respectively). The structure of the parameterlists varies slightly according to the data types, e. g. there are no indices used for the insertion of new elements into sets. In general, 'i' denotes an index, 'x' an entry, an 'd' a structured data object.

- **insert*** i x d
  Inserts a new entry into the structure.
  For sequences, the index must be an appropriate integer, for sets, it is omitted.

- **delete*** i/x d
  Deletes the entry with index i (or the element x from a set, respectively).

- **sel*** i/x d
  Returns the entry under index i.
  selset tests whether x is an element of d.

- **append*** d1 d2
  Returns the natural unification of the two structured objects.
  For tables, we need a tie-breaking rule for the case that d1 and d2 contain different values under the same index: let us say that d1 wins.

- **split*** d
  Splits the object d into two objects of (approximately) half the size of d.

- **make*** i x
  Returns a structured object of size 1 with entry x under index i. For sets and sequences, the index is omitted.

- **empty***
  Returns an empty initial structure.

- ***size** d
  Returns the number of entries in the object d.

These primitives are the canonical ones that everybody would expect. In addition, we introduce the following higher order functions ('f' and 's' denote functions):

- **fold*** f s d
  Applies s to all entries in the structured object d and connects the sub-results by the function f.

  E. g. , for a list l = [a1, ... , an] **foldseq** f s l computes

  $$f\ (s\ a_1)\ (f(s\ a_2)\ (\ldots f(sa_{n-1})\ (s\ a_n)\ \ldots))$$

  or, in infix-notation

  $$(s\ a_1)\ f\ (s\ a_2)\ f\ \ldots\ f\ (s\ a_{n-1})\ f\ (s\ a_n)$$

  'inserting' f between the entries of l. In contrast to what the prefix-notation might suggest, we must not assume any particular order for the evaluation of the f's [2], so f should be associative. We can take for granted that **fold*** does not change the order of the entries. Nevertheless, for tables and sets, the internal order of the indices or the elements, respectively, is not known.
  For tables, s takes both the index of an entry and the entry itself as parameters.

  The idea of **fold*** is nothing new, but most versions we have encountered (cf. [5], e. g.) differ from our version in some respects:

[2] This supports parallel evaluation.

- Usually **fold*** evaluates the invocations of f from left to right (or vice versa). This is sensible for classical list structures, but severely restricts parallelism. Our version is suitable for other implementations of data structures, e. g. implementations based on (height-balanced) trees. It turns out that for many applications f is associative, anyway. Thus no particular evaluation order is needed.
- **fold*** does not require a parameter which gives the result of **fold*** for an empty data object. This is different from most other functional languages, yet sensible. For a number of applications of **fold***, an empty data object means an erroneous situation (e. g. searching an empty table). The strict semantics of ML does not permit to use error-messages as parameters for function calls, hence the programmer has to check for this himself. **fold*** *always* considers an empty structure as an erroneous situation. In an non-strict language, **fold*** might take an additional parameter which yields the result for an empty data object.
- **fold*** requires two function parameters: one to compute the value for each entry, one to combine the subresults. Only the latter is standard, and this standard suffices to do things like summing up a list of numbers. We have observed that the number of possible applications for **fold*** increases significantly if we add the power to perform separate computations on every entry before combining the subresults. The classical technique regards the whole entry as a subresult. This approach can not be expected to be produce concise and reasonable programs when dealing with real-size problems and data structures which contain structured entries (e. g. complex records).

- **map*** f d
  Returns a modified version of object d with each entry x replaced by f(x).

- **connect*** f d1 d2
  Connects the entries of d1 and d2 pointwise by the function f.
  - For sequences, d1 and d2 must be of equal size;
  - for tables, indices and the corresponding entries with no matching pair in the other structured object are just copied;
  - **connect*** does not exist for sets.

For pattern-matching, we allow the use of the patterns ***leaf**(i, x) (or **leaf**(x), respectively) and **empty*** for structures of size 1 and the empty structure, respectively. The index i is used with tables only. Objects with more than one entry are only matched by the wild card '_'. A more liberal approach would be likely to violate the principles of data abstraction. E. g. ,we can not expect that there will be unique representations of data objects as (balanced) trees. Thus pattern-matching the root of such a tree would not necessarily yield the same subtrees for semantically identical data objects.

Before we continue the presentation of the language constructs, we want to motivate our choice of higher order functions on data structures by presenting examples for characteristical applications of them.

Anybody who is familiar with functional programming probably knows **map***. The application of a function to all entries of a structure occurs rather frequently, so we mention just one example. Multiplication of a matrix with a scalar can be expressed easily in terms of **maptab**. We define a type *matrix* as

```
type matrix =
    mat of (int * int , int ) table;
```

A matrix is a table which maps pairs of indices (**int** * **int**) to values (**int**). Unfortunately, ML does not permit type synonyms, so we have to use an additional constructor *mat*.

```
val matrix_times_scalar c (mat(m)) =
    mat(maptab ((op *) c) m);
```

The basic idea of **fold*** is that something is computed by folding a structure into a single value. E. g. , the length of a sequence can be expressed in terms of **foldseq**:

```
val length s =
    if s = emptyseq then
        0
    else
        foldseq (op +) (fun x . 1 ) s;
```

(In Cardelli's ML, **fun**-expressions denote lambda-expressions, i. e. nameless functions. In Standard ML [12], the keyword is **fn**.)

Searching a table for an entry can be expressed in terms of **fold***, too:

```
val searchtab entry t =
    if t = emptytab then
        false
    else
        let val s i x =
            {*** i is a dummy parameter required by foldtab ***}
            entry = x
        in
            foldtab (op or) s t
        end
```

But the value into which the data structure is folded can be structured, too. A suitable example for such an application are filterfunctions:

```
val filtertab condition t =
    if t = emptytab then
        emptytab
    else
        let val s index entry =
            if condition entry then
                maketab index entry
            else
                emptytab
        in
            foldtab appendtab s t
        end ;
```

A useful application of fold* for sets is the computation of a (finite) transitive closure. Assume that a function f computes the set of successors (w. r. t. the one step derivation) for one element. We can describe the computation of the closure as the least fixed-point of a transformation which is expressed in terms of **foldset**:

```
val closure f =
    fixed_point (foldset appendset f);
```

(The function **fixed_point** is introduced in the following subsection.)

Our last two examples illustrate the use of **connect***. If we represent vectors as tables with integers as indices, we can easily define a function for vector addition:

```
val VectorSum =
    connecttab (op +);
```

Another problem that can be solved in an elegant manner is the well-known histogram problem, i. e. the task of counting the occurences of certain information in the entries of a data structure. We assume that a given function 'info' extracts the information to be counted from the entries of the table.

```
val histogram info t =
    if t = emptytab then
        emptytab
    else
        let val s i entry =
            {*** i is a dummy parameter required by foldtab ***}
            maketab (info entry) 1
        in
            foldtab (connecttab (op +)) s t
        end ;
```

This concludes our list of examples.

2.2 More higher order functions

In addition to the higher order functions on data structures which we have presented in the previous subsection, we introduce some more primitives which are not defined on data structures. One function we shall use is **foldint**, an analogon to **fold*** defined for intervalls of integers. We give its semantics in ML-syntax:

```
val foldint f s (i, j) =
    if i > j then
        escape "foldint: range error"
    else if i = j then
        s i
    else
        let val m = (i + j) div 2
        in
            f (foldint f s (i, m)) (foldint f s (m + 1, j))
        end ;
```

As in the case of **fold***, we should assume no particular evaluation order for the invocations of the function f.

foldint is an analogon to the for-loop we know from PASCAL or MODULA-2. We would like to have something like while-loops, too. Thus we define

```
val rec iterate transformation condition object =
    {*** iterates the transformation of the object as
    long as the condition does not hold ***}
    if condition object then
        object
    else
        iterate transformation condition (transformation object);
```

The computation of a (least) fixed-point is of particular interest. It can be implemented in terms of **iterate**, but for the sake of effieciency we choose a different implementation.

```
val rec fixed_point transformation object =
    let val next_object = transformation object
    in
        if next_object = object then
            object
        else
            fixed_point transformation next_object
    end ;
```

An implementation in terms of **iterate** would cost superfluous evaluations of the transformation, because next_object would be computed twice upon every iteration:

```
val fixed_point transformation =
    let val condition object =
        object = (transformation object)
    in
        iterate transformation condition
    end ;
```

The set of primitives presented so far falls short of what is needed for a general purpose language. For the manipulation of sets, e. g. , we need operations like intersection and set difference. We shall introduce additional functions whenever they are needed for the programs we present. Most of them can be expressed in terms of the functions introduced so far, but for a real programming system we should choose more efficient implementations.

3 Sorting

Implementing sorting algorithms is one of the standard problems of computer science which are used to demonstrate the expressive power of programming languages. Thus we shall deal with this problem, too. We give programs for two of the most often used sorting algorithms, namely mergesort and quicksort.

3.1 Mergesort

In its commonly used version, mergesort is a list-oriented sorting algorithm. Its basic idea can be described as follows:

- To sort a list, we split it into two lists of equal length and sort them separately, then merge the two lists into a sorted big list.
- To merge two sorted lists, we compare the first entries of the lists, choose the smaller one, merge the rest, and put the chosen entry in front of the merged rest.

The sorting phase can also be described in a non-recursive way: To sort a list, we split in into sorted lists (of length 1, in general), and merge them subsequently until there is only one list left.

This rough outline of the algorithm can already be transformed into a program:


```
val rec merge comp s1 s2 =
    {*** comp x y = true iff x ≤ y ***}
    if s1 = emptyseq then
        s2
    else if s2 = emptyseq then
        s1
    else
        let val x, y = selseq 1 s1, selseq 1 s2
        in
            if comp x y then
                insertseq 1 x (merge (deleteseq 1 s1) s2)
            else
                insertseq 1 y (merge s1 (deleteseq 1 s2))
        end ;

val mergesort comp =
    foldseq (merge comp) makeseq ;
```

If we compare this program with conventional implementations of mergesort (e. g. [15]), we wonder where the alternating distribute-and-merge phases have gone. In fact, the distribution phases are hidden in **foldseq** which produces something like a perfect decomposition of a data structure. The merge phases are hidden in **foldseq**, too. The merge phases are the repeated invocations of *merge* which combine the subresults into the final result of **foldseq**.

The reader may object that sequences look like a sort of data structure which is to be stored in the main memory, whilst mergesort is meant to sort amounts of data which are so large that they have to be stored in the background. But the concept of *fold** can be applied to data stored on background media, too. In this context, it is important that we do not require any particular evaluation order for the invocations of *merge* (or, in general, the function which combines the subresults).

3.2 Quicksort

Quicksort is well-known as a sorting-algorithm for arrays [15]. Nevertheless, it is much easier to understand as a sorting algorithm for list-like data structures:

- To sort a list, we take one of its entries, split the rest into two lists containing the entries smaller and greater than the chosen entry, sort these two lists independently and join the sorted lists with the chosen entry into a single, sorted list.

This description can be transformed into a functional program straightforward:

```
val partition comp x =
    {*** splits a sequence into the sequences of entries smaller
    and greater than x ***}
    let val one_y y =
        {*** handles one entry ***}
        if comp x y then
```

```
            emptyseq, makeseq y
        else
            makeseq y, emptyseq;
    val append (s1, s2) (t1, t2) =
        appendseq s1 t1, appendseq s2 t2
    in
        foldseq append one_y
    end ;

val rec quicksort comp s =
    if seqsize s < 2 then
        s
    else
        let val hd, tl = selseq 1 s, deleteseq 1 s;
        val l, r = partition comp hd tl
        in
            appendseq (quicksort comp l) (insertseq 1 hd (quicksort comp r))
        end ;
```

But what about sorting tables or other array-like structures? The array-oriented versions of quicksort and mergesort can be programmed in functional languages, too, but we should think twice before we start to write a program that simulates the imperative versions of our sorting algorithms. The most important advantage of quicksort with imperative arrays is space efficiency (besides the fact that quicksort is fast, of course). This advantage will almost surely be lost when simulating the algorithm in a functional language, because each update is likely to create garbage cells, and a long sequence of updates will create lots of garbage. The easiest way to sort a table seems to be transforming it into two sequences (one for the indices, one for the values), sorting the values separately, and connecting them with the indices finally. A program for quicksort might read like this:

```
val quicksort_tables comp t =
    let val indices, values =
        let val one_entry i v =
            {*** handles one entry of the table ***}
            makeseq i, makeseq v;
        val unify (i1, v1) (i2, v2) =
            (appendseq i1 i2), (appendseq v1 v2)
        in
            foldtab unify one_entry t
        end
    in
        connectseq maketab indices (quicksort comp values)
    end ;
```

Implementations of quicksort and mergesort for tables which do not use sequences can be found in [11]. They show that it can be done without using sequences, but it is not recommended to write such programs.

4 Subset Construction

In this section we develop a program for the well-known subset construction which transforms a nondeterministic finite automaton into a deterministic one. This method is one of the fundamentals of automata theory, hence there is no need to present it in detail. We just introduce our notation:

A non-deterministic finite automaton (NFA) is a quintuple

$$A = < Q, \Sigma, q_0, \delta, F >,$$

with Q denoting the set of states, Σ the alphabet, q_0 the initial state, δ the transition function, and F the set of final states. The type of δ is

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q),$$

with ε denoting the empty word and $\mathcal{P}$ the power set.

For a deterministic finite automaton (DFA), δ must be of type

$$\delta : Q \times \Sigma \rightarrow Q.$$

These types can be described in our functional language straightforward:

```
type 'sigma or_eps =
    {*** Σ ∪ {ε} ***}
    epsilon |
    letter of 'sigma;

type ('states, 'sigma) NFA =
    NA of 'states set * 'sigma set *
    'states * ('states * ('sigma or_eps) , 'states set) table * 'states set;

type ('states, 'sigma) DFA =
    DA of 'states set * 'sigma set * 'states * ('states * 'sigma) table * 'states set;
```

Before we present the program, we give a survey of its main functions: ***eps_closure*** computes the ε-closure for a set A of states, i. e. the set of states reachable from A by ε-moves. For a set s of states and a letter a, ***transition s a*** computes the set of states that can be reached from elements of s by reading a, ε-transitions included.

The construction of the transition table for the DFA is the most complex part of the program. It is built up step by step, with each intermediate stage characterized by a triple (PQ, pdelta, NQ), where

- PQ denotes the set of DFA-states constructed so far;

- pdelta denotes the part of the DFA transtion table constructed so far;
- NQ denotes the set of elements from PQ whose outgoing transitions have not been computed yet.

The initial stage is characterized by the tuple

$$(Q_0, emptytab, \{Q_0\}),$$

where

$$Q_0 = epsclosure\{q_0\}$$

denotes the initial state of the DFA. The iteration halts when NQ is empty.

```
val subset_construction (NA(Q, Sigma, q0, delta, F)) =
    DA(PQ, Sigma, Q0, pdelta, PF)
    where
    val eps_closure =
        let val eps_succ q =
            {*** set of successors for one state q ***}
            seltab (q, eps) delta ? emptyset
            {*** default value is emptyset ***}
        in
            closure eps_succ
        end ;

    {************************************************}
    {************************************************}

    val transition s a =
        let val succ_one_state q =
            seltab (q, letter a) delta ? emptyset
        in
            eps_closure(foldset appendset succ_one_state s)
        end ;

    {************************************************}
    {************************************************}

    val Q0 = eps_closure (makeset q0);
    val (PQ, pdelta, dummy) =
        iteration next_tuple stop_condition (makeset Q0, emptytab, makeset Q0)
        where
        val unify ((s1, t1), (s2, t2)) =
            {*** this is an auxiliary function to combine results ***}
            (appendset s1 s2, appendtab t1 t2);

        {************************************************}
```



```
        val stop_condition (PQ, pdelta, NQ) =
            NQ = emptyset;

        {************************************************}

        val next_tuple (PQ, pdelta, NQ) =
            let val (NPQ, npdelta) =
                {*** the part of the DFA which is determined by
                the outgoing transitions of NQ ***}
                let val constr_one_state s =
                    {*** construction for one s ∈ NQ ***}
                    let val constr_one_letter a =
                        let val t = transition s a
                        in
                            (makeset t, maketab (s, a) t)
                        end
                    in
                        foldset unify constr_one_letter Sigma
                    end
                in foldset unify constr_one_state NQ
            in
                (appendset PQ NQ, appendtab pdelta npdelta,
                    setdifference NPQ PQ)
            end ;

    {************************************************}
    {************************************************}

    val PF =
        let val cond s =
            intersection s F <> emptyset
        in
            subset cond PQ
        end
    end ;
```

5 Testing for LL(1)-Property

This section is divided into four parts. In the first part, we give a formal characterization of LL(1)-property and present a set of rules which can be applied to compute the required information. In the second and third part, we present the programs which do the most difficult work: compute the so-called ***first***- and ***follow***-sets. In the fourth part, we present the remainder of the program.

5.1 Formal Description

One of the most often used techniques for parsing is recursive descent [14]. The advantages of recursive descent are that it is easy to understand and that the parser can be adopted to minor changes of the grammar without problems - at least, in general. One major drawback of recursive descent is that not all grammars of practical interest can be parsed by this method.

In this section we present a method for checking whether a context-free grammar (CFG) can be parsed by recursive descent without problems. The test we perform is also known as test for LL(1)-property. Intuitively, it means that for the left analysis of a derived word, knowing the next nonterminal and the next terminal symbol suffices to decide which rule of the grammar was applied next.

We introduce some notation before we start the description of the algorithm.

A CFG is a quadruple

$$G = < NS, TS, R, S >,$$

with NS denoting the finite, non-empty set of nonterminal symbols, TS the finite, non-empty set of terminal symbols, R the finite set of rules, and $S \in NS$ the start symbol. NS and TS must be disjoined. R can be interpreted as a function which maps each nonterminal to its set of right hand sides, thus we can say that R is of type $NS \rightarrow \mathcal{P}((NS \cup TS)^*)$. The empty word is denoted by ε, and $\mathcal{P}$ denotes the power set. We write $\alpha \rightarrow \beta$ for any two phrases α, β if there is a rule transforming α to β, and $\Rightarrow$ for the transitive and reflexive closure of $\rightarrow$. In general, we use capital roman letters to denote nonterminals, small roman letters for terminals, and small greek letters for phrases. For simplicity, we assume that we deal with reduced grammars, i. e. with grammars in which every nonterminal appears in some phrase that can be derived from the start symbol and in which every nonterminal can be derived to some word, i. e. some phrase consisting of terminal symbols only.

We need an exact characterization of what it means that a grammar is LL(1). Obviously, this property depends on the terminals that may be at first position in words derived from nonterminals:

Definition (first-set)

For any phrase α, $\underline{\text{fi}}\,\alpha$ denotes the set of terminals that can be at first position in words derived from α.

$$\underline{fi}\,\alpha = \begin{cases} \{a_1 \mid a_1...a_n \in TS^*, \alpha \Rightarrow a_1...a_n\} \cup \{\varepsilon\} & if\ \alpha \Rightarrow \varepsilon \\ \{a_1 \mid a_1...a_n \in TS^*, \alpha \Rightarrow a_1...a_n\} & otherwise \end{cases}$$

Note: $\underline{\text{fi}}\,\varepsilon = \{\varepsilon\}$.

We must pay attention to the fact that $\varepsilon \in \underline{\text{fi}}\,\alpha$ may be true for some $\alpha \neq \varepsilon$. E. g. , if we have $\alpha A \beta \rightarrow \alpha\sigma\beta$ and $\varepsilon \in \underline{\text{fi}}\,\sigma$, the application of the rule A $\rightarrow \sigma$ might be recognised only by taking into account the first letter derived from β.

This motivates the next definition:

Definition (follow-sets)

For any nonterminal A, $\underline{\text{fo}}$ A denotes the set of terminals (or ε) which may appear immediately behind words that were derived from A in any phrase.

$$\underline{fo}\,A = \bigcup_{S \Rightarrow \alpha A \beta} \underline{fi}\,\beta$$

Note: $\varepsilon \in \underline{\text{fo}}$ S.

In terms of these definitions, we are now able to define the set of terminals which indicate that a particular rule was applied to a given nonterminal.

Definition (look-ahead-set)

For any rule A $\rightarrow \beta$, $\underline{\text{la}}$ (A,β) denotes the set of terminals that may appear next if in any left derivation A $\rightarrow \beta$ was the next rule which was applied.

$$\underline{la}(A, \beta) \;=\; \bigcup_{x \in \underline{fo}\,A} \underline{fi}\,\beta x$$

Definition (LL(1)-property)

We say that a grammar is LL(1) iff for any two distinct rules A $\rightarrow \alpha$, A $\rightarrow \beta$

$$\underline{la}(A, \alpha) \cap \underline{la}(A, \beta) \;=\; \emptyset$$

holds.

This characterization of the LL(1)-property is mathematically exact, yet it is not constructive, because the set of derivations to be taken into account for the computation of the follow-sets may (and, in general, will) be infinite. But we can give a small finite set of rules which enables us to compute the desired information effectively (and efficiently):

- **first-sets:**

$$\begin{array}{llcll}
(fi\,1) & \underline{fi}\,\varepsilon & = & \{\varepsilon\} & \\
(fi\,2) & \underline{fi}\,a & = & \{a\} & for\ a \in TS \\
(fi\,3) & \underline{fi}\,\alpha & \subseteq & \underline{fi}\,A & if\ A \rightarrow \alpha \in R \\
(fi\,4) & \underline{fi}\,\alpha\beta & = & (\underline{fi}\,\alpha \setminus \{\varepsilon\}) \cup \underline{fi}\,\beta & if\ \varepsilon \in \underline{fi}\,\alpha \\
(fi\,5) & \underline{fi}\,\alpha\beta & = & \underline{fi}\,\alpha & if\ \varepsilon \notin \underline{fi}\,\alpha
\end{array}$$

- **follow-sets:**

$$\begin{array}{llll} (fo\,1) & \varepsilon & \in \underline{fo}\,S & \\ (fo\,2) & (\underline{fi}\,\beta \setminus \{\varepsilon\}) & \subseteq \underline{fo}\,B & if\ A \rightarrow \alpha B\beta \in R \\ (fo\,3) & \underline{fo}\,A & \subseteq \underline{fo}\,B & if\ A \rightarrow \alpha B\beta \in R,\ \varepsilon \in \underline{fi}\,\beta \end{array}$$

  Note that (fo 3) includes the case that $\beta = \varepsilon$.

- **look-ahead-sets:**

$$\begin{array}{lll} (la\,1) & \underline{la}(A,\beta) = (\underline{fi}\,\beta \setminus \{\varepsilon\}) \cup \underline{fo}\,A & if\ \varepsilon \in \underline{fi}\,\beta \\ (la\,2) & \underline{la}(A,\beta) = \underline{fi}\,\beta & if\ \varepsilon \notin \underline{fi}\,\beta \end{array}$$

These rules induce a continous transformation for vectors of first- and follow-sets, and the least fixed-point of this transformation is the result we are looking for.

The following example demonstrates how such a fixed point is computed: Let G = <NS, TS, R, S> a CFG with NS = {A, B, C, S}, TS = {a, b, c, d}, and R the function which is described by the table

N	R(N)
S	{ABC, Ad}
A	{ε, aS}
B	{ε, b}
C	{A, c}

We need the first-sets for

- the right hand sides of all nonterminals (cf. (fi 3))

$$righthands = \{\mathrm{ABC}, \mathrm{Ad}, \varepsilon, \mathrm{aS}, \mathrm{b}, \mathrm{A}, \mathrm{c}\}$$

- the phrases which appear behind nonterminals in rules (cf. (fo 2), (fo 3))

$$phrases = \{\mathrm{BC}, \mathrm{C}, \mathrm{d}, \varepsilon\}$$

The following table shows the approximations of the first-sets which are generated while computing the fixed point.

	0	1	2	3	4	5
ABC	$\emptyset$	$\emptyset$	{a, b, c}	{a, b, c, ε}	{a, b, c, ε}	{a, b, c, ε}
Ad	$\emptyset$	$\emptyset$	{a, d}	{a, d}	{a, d}	{a, d}
ε	$\emptyset$	{ε}	{ε}	{ε}	{ε}	{ε}
aS	$\emptyset$	$\emptyset$	{a}	{a}	{a}	{a}
b	$\emptyset$	{b}	{b}	{b}	{b}	{b}
A	$\emptyset$	$\emptyset$	{ε}	{a, ε}	{a, ε}	{a, ε}
c	$\emptyset$	{c}	{c}	{c}	{c}	{c}
BC	$\emptyset$	$\emptyset$	{b, c}	{b, c, ε}	{a, b, c, ε}	{a, b, c, ε}
C	$\emptyset$	$\emptyset$	{c}	{c, ε}	{a, c, ε}	{a, c, ε}
d	$\emptyset$	{d}	{d}	{d}	{d}	{d}

The next table shows the development of the follow-sets:

	0	1	2	3
S	{ε}	{ε}	{a, b, c, d, ε}	{a, b, c, d, ε}
A	$\emptyset$	{a, b, c, d, ε}	{a, b, c, d, ε}	{a, b, c, d, ε}
B	$\emptyset$	{a, c, ε}	{a, c, ε}	{a, b, c, d, ε}
C	$\emptyset$	{ε}	{ε}	{a, b, c, d, ε}

The computation of the look-ahead-sets is left to the reader.

Up to now, we have given our explanations and definitions on a rather abstract mathematical level. The reader will be surprised to see that the description given above is all we need to start writing our program! We have outlined an algorithm which is elegant, concise, and is expressed in 'large terms of the task at hand' (cf. [3]). Now we shall demonstrate how this high-level description of an algorithm can be encoded in a functional language, provided the language has the characteristics we have outlined in section 2.

Before we start presenting the program which computes the first-sets, we discuss how to represent a CFG in a program. The type ***grammar*** we introduce has two type parameters, one for the type of the nonterminals, one for the type of the terminals. These two types must be distinguished, but must be mixed into phrases, too. Thus we define

```
type ('n, 't) symbol =
    N of 'n |
    T of 't;

type ('n, 't) grammar =
    CFG of 'n set * 't set * ('n, ('n,'t) symbol sequence set) table * 'n;
```

Note how this definition reveals the logical structure of a grammar: R is a function that, for any given nonterminal A, denotes the set of right hand sides of the rules for A, which is a set of symbol sequences (phrases).

The first-, follow-, and look-ahead-sets we wish to compute contain terminals or ε. This type is represented as

```
type 't or_eps =
    eps |
    L of 't;
```

5.2 Computation of the first-sets

Before we present the program for the computation of the first-sets, we give a short outline of the main functions appearing in this program.

The rules (fo 1) - (fo 3) and (la 1) - (la 2) require the first sets for a certain set of phrases, and we want to compute a table containing these first-sets. We need

- the right hand sides of all rules (cf. (la 1), (la 2));
- the phrases which appear in right hand sides of rules behind nonterminals (cf. (fo2), (fo 3)).

We call these two sets of phrases ***right_hands*** and ***phrases***, compute them separately, and unify them into a set called ***all_phrases***.

The computation of ***right_hands*** is straightforward. To compute ***phrases***, we need something like a loop which tests all positions in a right hand side whether the corresponding symbol is a nonterminal. Whenever this is the case, we extract the part of the phrase that begins immediately behind this position. For this purpose, we assume that we have a function

$$\mathbf{rightseq} : \mathbf{N} \rightarrow ('a\ \mathbf{sequence} \rightarrow' a\ \mathbf{sequence})$$

which returns the sequence of the last n entries of s when called with parameters n and s. (The implementation of this function in terms of our given functions is straightforward.) The sets of phrases we thus extract from the right hand sides of all rules must be unified (in the set-theoretical sense) for all right hand sides of all nonterminals. With ***all_phrases*** we can easily generate the initial table ***init_fi*** for the computation of the first-sets. The function ***transform*** computes the next approximation of the fixed-point (the vector of first-sets) we are interested in.

```
val first_sets(CFG(NS, TS, R, S)) =
    {* * * computes the table of first-sets needed for testing the LL(1)-property * * *}
    fixed_point transform init_fi
    where
    val all_phrases =
        {* * * The set of all phrases whose first-sets are needed * * *}
        appendset phrases right_hands
        where
        val phrases =
            {* * * the set of phrases appearing behind nonterminals in
            the right hand sides of rules * * *}
            foldtab appendset phrss_one_symb R
            where
            val phrss_one_symb (A, right_hands_A) =
                {* * * handles the right hand sides of one nonterminal * * *}
                let val phrss_one_rule alpha =
                    {* * * handles one right hand side of A * * *}
                    let val pos i =
                        {* * * tests i-th symbol in alpha * * *}
                        case selseq i alpha of
                            T(a) .
                                emptyset |
                            N(B) .
                                let val l = seqsize alpha;
```



```
                                    val beta = rightseq ( l - i ) alpha
                                in
                                    makeset beta
                                end
                    in
                        {* * * body for phrss_one_rule * * *}
                        if alpha = emptyseq then
                            {* * * α = ε * * *}
                            emptyset
                        else
                            foldint appendset pos ( 1, seqsize alpha)
                    end
                in
                    {* * * body for phrss_one_symb * * *}
                    foldset appendset phrss_one_rule right_hands_B
                end
        end ;

    {* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}

    val right_hands =
        {* * * right hand sides of all rules * * *}
        let val s( B, right_hands_B ) =
            right_hands_B
        in
            foldtab appendset s R
        end
    end

{* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}
{* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}

val init_fi =
    {* * * intial table of (empty) first-sets for all_phrases * * *}
    let val s phrs =
        maketab phrs emptyset
    in
        foldset appendtab s all_phrases
    end ;

{* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}
{* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}

val transform fi =
    {* * * yields the next approximation for fi according to (fi 1) - (fi 5) * * *}
    foldset appendtab next_fi all_phrases
    where
    val next_fi sym =
```

```
{*** computes the next approximation for the single phrase sym ***}
let val fi_one_symb sy =
    {*** evaluates one symbol of sym ***}
    case sy of
        T(a) .
            makeset (L(a)) |
        N(A) .
            {*** uses ( fi 3 ) ***}
            let val fi_one_rule alpha =
                seltab alpha fi
            in
                foldset appendset fi_one_rule (seltab A R)
            end ;

{***********************************************}

val collect_fi fi_alpha fi_beta =
    {*** unifies two first-sets according to (fi 4) and (fi 5) ***}
    if selset eps fi_alpha then
        {*** if ε ∈ fi α ***}
            appendset (deleteset eps fi_alpha) fi_beta
    else
        fi_alpha

{***********************************************}

in
    {*** body for next_fi ***}
    let val new_fi =
        {*** next approximation for the first-set of sym ***}
        if sym = emptyseq then
            {*** sym = ε ***}
            makeset eps
        else
            foldseq collect_fi fi_one_symb sym
    in
        maketab sym new_fi
    end
  end
end ;
```

5.3 Computation of the follow-sets

The base for the computation of the follow-sets is provided mainly by (fo 2) and (fo3). The structure of the program is similar to the one we used for the computation of ***phrases*** in the preceeding program: For any rule $A \rightarrow \sigma$, a function called ***fo_tab*** computes a table with approximations for the follow-sets of the nonterminals

occurring in σ. To do this, we compute the next approximation of the follow-set for any nonterminal B in σ according to (fo 2) and (fo 3) and unify these initial tables of size 1 into one for the whole phrase σ. We collect and unify these tables for all right hand sides of all nonterminals.
For simplicity, we do not use (fo 1) as a transformation rule, instead we take it into account when initializing the table for the follow-sets.

```
val unify_follow_tabs =
    {*** this auxiliary function yields the unification of two tables with follow-sets ***}
    connecttab appendset;

{***********************************************}
{***********************************************}

val follow_sets fi (CFG (NS, TS, R, S)) =
    {*** computes the table of follow-sets in dependency from the appropiate
    table of first-sets ***}
    fixed_point transform init_fo
    where
    val init_fo =
        {*** initial approximation of the table with the follow-sets ***}
        let val one_nonterminal A =
            if A = S then
                maketab S (makeset eps)
            else
                maketab A emptyset
        in
            foldset appendtab one_nonterminal NS
        end ;

    {***********************************************}
    {***********************************************}

    val transform fo =
        {*** computes the next approximation for the follow-sets ***}
        foldtab unify_follow_tabs one_nonterminal R
        where
        val one_nonterminal A right_hands_A =
            {*** handles one nonterminal A and its right hand sides ***}
            let val one_rule A alpha =
                {*** handles a rule A → α ***}
                let val pos i =
                    {*** evaluates the i-th symbol in α ***}
                    case selseq i alpha of
                        T(_) .
                            {*** terminal symbols have no effect ***}
                            emptytab |
                        N(B) .
```

```
                  {*** apply (fo 2) and (fo 3) ***}
                  let val beta = rightseq (seqsize alpha - i ) alpha;
                  val fi_beta = seltab beta fi;
                  val fo_B_new =
                     let val if_eps =
                        if selset eps fi_beta then
                           seltab B fo
                        else
                           emptyset
                     in
                        appendset if_eps (deleteset eps fi_beta)
                     end
                  in
                     maketab B (appendset (seltab B fo) fo_B_new)
                  end
            in
               {*** body for one_rule ***}
               if alpha = emptyseq then
                  {*** α = ε ***}
                  emptytab
               else
                  foldint unify_follow_tabs pos (1, seqsize alpha)
            end
         in
            {*** body for one_nonterminal ***}
            foldset unify_follow_tabs (one_rule A) right_hands_A
         end
      end
   end ;
```

5.4 Computation of the look-ahead-sets and test for LL(1)-property

With the computation of the first- and follow-sets we have solved the main problem that arises when checking for LL(1)-property. What is left to be done is nothing difficult to encode in a programming language, yet, for sake of completeness, we add the remaining functions.

For every nonterminal we generate a table with the look-ahead-sets of all respective right hand sides. We unify these tables in another big table, each of them being indexed by the corresponding nonterminal. The actual test for the LL(1)-property is performed by checking the local tables for every nonterminal and connecting the results by the logical *AND*.

```
 val look_ahead_sets (CFG (NS, TS, R, S)) =
    let val fi = first_sets (CFG (NS, TS, R, S));
```



```
    val fo = follow_sets fi (CFG (NS, TS, R, S));
    val one_nonterminal A right_hands_A =
       {*** computes the look-ahead-sets for the nonterminal A
       and its right hand sides ***}
       let val one_rule A alpha =
          {*** generates a table with la(A, α) under index α ***}
          let val fi_alpha = seltab alpha fi;
          val look_ahead =
             if selset eps fi_alpha then
                appendset (deleteset eps fi_alpha) (seltab A fo)
             else
                fi_alpha
          in
             maketab alpha look_ahead
          end
       in
          {*** body for one_nonterminal ***}
          maketab A (foldset appendtab (one_rule A) right_hands_A)
       end
    in
       {*** body for look_ahead_sets ***}
       foldtab appendtab one_nonterminal R
    end ;

{**********************************************}
{**********************************************}

val LL_1_test G =
    foldtab (op &) check_one_nonterminal (look_ahead_sets G)
    where
    val check_one_nonterminal A la_table_A =
       {*** checks the look-ahead-sets of the nonterminal A; A is allright iff the
       cardinality of the union of all its look-ahead-sets equals the sum of the
       cardinalities ***}
       cardinality_of_union = sum_of_cardinalities
       where
       val cardinality_of_union =
          let val one_set alpha la_alpha =
             la_alpha
          in
             setsize (foldtab appendset one_set la_table_A)
          end ;

    {**********************************************}

       val sum_of_cardinalities =
          let val one_card alpha la_alpha =
             setsize la_alpha
```

```
        in
          foldtab (op +) one_card la_table_A
        end
      end
    end ;
```

The reader should note how clearly the program we presented reveals the logical structure and the different phases of the high-level algorithm we outlined. Note also that the whole program does not contain any function definition which uses recursion explicitly. We claim that we made good on our promise that the higher order functions we presented provide a powerful means to replace recursive function definitions.

6 Implementation Aspects

This paper is about programming methodology, but it is worthwhile to discuss some implementation aspects of the things we have presented:

- Which consequences do arise for the implementation of functional languages from our observations?
- How efficiently can our data structures be implemented?

Our examples illustrate that we can avoid many recursive function definitions by using a small fixed set of higher order functions. The formal definitions for the semantics of these functions were given in ML-notation, and most of these definitions were recursive. Hence, we can not expect that an implementation of these functions in library modules will increase run time efficiency. The only effect would be that recursion would be hidden (which is worthwhile, anyway).

Run time efficiency could be increased if these higher order functions would become part of a functional language. An implementation in terms of a (abstract or physical) machine could exploit the special features of our functions, saving both time and space. Saving space (and thus reducing garbage and the need for garbage collection) would be possible particularly with structured data objects. E. g. ,there would be no need to store the intermediate versions which are created when computing a fixed point, and updates of data structures might be made in-place. Furthermore, one would try to find non-recursive implementations of our functions, which would save additional time and space.

As we mentioned before, our data structures are not easy to implement. Various techniques have been studied [8], [9], [11], and it seems that an implementation technique which is based on trees is suitable. (It is an open question whether it pays off to use balanced trees or whether it fits to trust in a well-natured average behaviour

of unbalanced trees.) Using this technique, pointwise operations like **sel*** run in O(log n) (with n denoting the size of a data object), and functions like **fold*** and **map*** can be implemented efficiently (O(log n) when using unrestricted parallelism).

One fact of particular interest is that implementations based on trees support parallel execution of programs. This aspect seems to be of great importance for the future development of functional programming. Note that our higher order functions **fold***, **map***, and **connect*** avoid the bottleneck-problem usually associated with trees [9]: we do not create a large number of processes which request data from one data structure. Instead, we create a tree of processes, and each of these processes gets direct access (rather than indirect access via the root) to its data, i. e. the subtree it handles.

7 Summary

We have developed and discussed programs for a number of non-trivial standard problems. The most striking observation is that a combination of high-level data structures and higher order functions enables us to write programs which are rather concise and almost free from recursive function definitions. The possibility to avoid explicit recursion may help to convince programmers who are uncomfortable with applicative languages that functional programming is worthwhile.

Recently there has been remarkable progress in the development of parallel machines and parallelizing compilers for applicative languages (cf. [10], [9], e. g.). The programming language we used in this paper makes only few assumptions about implementation techniques. Nevertheless, we defined the semantics in a way that supports parallel evaluation. Particularly **fold*** was designed with regard to this aspect. The price we have to pay is a certain degree of nondeterminism, namely the programmer's uncertainty about the evaluation order of the binary function *f*. But we believe that this is no real problem, because in most cases the evaluation order does not matter, anyway. The programming examples in this paper support this claim.

We expect that further research will yield the result that our programming paradigm is suitable for many applications of functional programming. Therefore we suggest that more higher order functions and more powerful data structures should be included in functional languages and that there should be more research on the their efficient implementation.

8 Appendix

This paper takes for granted that the reader is familiar with the principles of functional programming, hence, we do not explain the basic language constructs of applicative languages. Nevertheless, we present the syntax of some ML-constructs in order to help readers who are not familiar with this particular language. As we mentioned in the very beginning of this paper, we use Cardelli's version of ML [6], [7]. It differs slightly from Standard ML [12].

- **type variables**
  ML permits the usage of type variables, e. g. for defining polymorphic data types. The syntax for type variables is an identifier which is preceeded by an apostroph.

- **user-defined data types**
  User-defined data types in ML are algebraic data types. A data object consists of a constructor which is applied to a number of arguments of various types. Recursive definitions are permitted. A constructor definition consists of the name of the constructor, the keyword **of**, and the sequence of the argument types. The argument types are separated by '*'. There may be various constructors for one data type. The alternatives are separated by '|'.

- **local definitions**
  There are two ways how to define objects which are local to an expression:

  - **let** *definitions* **in** *expression* **end**;
  - *expression* **where** *definitions* **end**;

  Their semantics are identical, but the choice influences readability and understandability of programs, particularly for large programs with nested local definitions.

- **infix-to-prefix conversion**
  Prefix notation for functions is standard in ML. Nevertheless, function names can be declared to be used as infix-operators (we do not make use of it in this paper), and some predefined functions are infix. Putting the word **op** in front of such a function name converts it to its curried prefix version. E. g. , **op** + is of type $\mathbf{N} \rightarrow (\mathbf{N} \rightarrow \mathbf{N})$.

- **comments**
  Comments are enclosed in '{ }'.

References

[1] Aasa, Holmström, and Nilsson: *An Efficiency Comparison of Some Representations of Purely Functional Arrays,* Programming Methodology Group Report No. 33, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg (1987)

[2] Adelson-Velskij and Landis: *An Algorithm for the Organisation of Information,* Soviet Math. Dokt. 3 (1962)

[3] Backus: *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,* Comm. ACM, Vol. 21, No. 8 (1978)

[4] Bird: *Constructive Functional Programming,* International Summer School on Constructive Methods in Computing Science, Marktoberdorf 1988

[5] Bird and Wadler: *Introduction to Functional Programming,* Prentice Hall (1988)

[6] Cardelli: *ML under UNIX,* Bell Laboratories (1983)

[7] Kuchen: *Funktionales Programmieren in ML,* Lehrstuhl für Informatik II, Aachen Technical University (1986)

[8] Kuchen: *Parallele programmierte Graphreduktion in der abstrakten Maschine PAM, insbesondere Realisierung applikativer Datenstrukturen,* 4. Workshop Alternative Konzepte für Sprachen und Rechner Bad Honnef 1987, Angewandte Mathematik und Informatik 2/87 – I, Münster University (1987)

[9] Kuchen: *Parallele Implementierung einer funktionalen Sprache auf einem OCCAM-Transputersystem unter Berücksichtigung applikativer Datenstrukturen,* Ph. D. Thesis, Aachen Technical University (1989) (in preparation)

[10] Loogen:
Parallele Implementierung funktionaler Programmiersprachen, Ph. D. Thesis, Aachen Technical University (1989)

[11] Maaßen: *Entwicklung paralleler Algorithmen unter Verwendung spezieller applikativer Datenstrukturen,* Diploma Thesis, Lehrstuhl für Informatik II, Aachen Technical University (1988)

[12] Milner: *The Standard ML Core Language,,* Polymorphism, Vol. 2, No. 2, (1985)

[13] Myers: *Efficient Applicative Data Types,* ACM Symposion on Principles of Programming Languages 1984

[14] Waite and Goos: *Compiler Construction,* Springer Verlag, Berlin (1984)

[15] Wirth: *Algorithmen und Datenstrukturen,* Teubner Studienbücher Informatik Bd. 31, Teubner Verlag, Stuttgart (1975)

[16] Wise: *Matrix Algebra and Applicative Programming,* LNCS No. 274, Springer Verlag Berlin, (1986)