On Indexed Data Structures and Functional Matrix Algorithms

Nils Ellmenreich and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau
{nils,lengauer}@fmi.uni-passau.de

Abstract

At first sight, scientific computing seems an application area ideally suited for functional programming: scientific programs are described by a constructive input/output specification. However, scientific programmers still remain reluctant to consider the functional approach.

The difficulties lie in part in missing applications and reluctance to change, but in our view technically in the features and performance properties which current functional programming languages offer. We study two examples: reflexive transitive closure and LU decomposition. For each, we offer what we believe to be a natural functional solution and examine its strengths and weaknesses. From this, we try to draw conclusions about the shape of scientific problems suitable for functional programming and state properties which would make a functional language more suitable for scientific programming.

1 Introduction

Despite the rise in the popularity of functional programming in the last decade, scientific programmers have been steadfast in choosing the imperative programming paradigm for the solution of their problems.

A *functional* program satisfies the *single-assignment property*, which states that every variable is given at most one value during program execution. This property guarantees that equals can be substituted for equals at any time. An *imperative* program contains reassignments, i.e., violates the single-assignment rule.

The choice of a functional over an imperative implementation is often motivated by its greater similarity to the specification of the problem. In the best case, a mathematical specification of a problem describes the result as expressions containing basic functions and relations. In addition, one often wants to take advantage of reasoning capabilities for program development and verification. It has frequently been argued that the functional paradigm offers more support here than the imperative paradigm [Rea89, p.5].

There are basically three different options for how scientific problems are described in textbooks and in the research literature:

1. as a set of equations, often recurrence equations,
2. more constructively, as an algorithm in semi-formal pseudo code, usually with an imperative feel,
3. already by a solution, a program in Fortran or C.

The main reason for the lack of interest of scientific programmers in functional programming is, of course, the current performance advantage of imperative programs. The gap has been closing recently to some extent [HFA$^+$96], especially where compiler-specific optimizations are being offered. The Sisal project [BOCF92] has even claimed to be as good as or better than Fortran on certain applications [Can92]. Be this as it may – we are not addressing the performance gap between the functional and the imperative paradigm here, but rather the advantage of functional array programming for scientific purposes.

Another reason for the unpopularity of the functional programming paradigm among scientific programmers is that the description of scientific problems is usually already biased towards the imperative paradigm. This impairs the derivation of a functional program – and, even worse, the result is likely to be inefficient. To be able to compare the benefits and drawbacks of both paradigms for scientific programming, the starting point, i.e., the description of the problem, must be as abstract as possible. Therefore, we prefer the first of the above options. The latter two options are more popular in the literature but contain often unexplained and, maybe, unnecessary restrictions.

Actually, for problems with an abstract, equational specification, the functional paradigm has an advantage, because the specification is often already similar to a functional program. The simplest case is that the data structure is defined by a single, usually recursive equation and each value is defined exactly once. However, there are some scientific problems which require an incremental definition of the data structure. In a single-assignment language this is problematic since, for any change of a value, a new data object must be created.

2 Arrays in Functional Programming

The dominant data structure in scientific programming is the array. The reason for this is probably the influence of Fortran; it is unlikely that the present dominance of indexing in scientific computations is inherent. However, we are quite happy to base our problem descriptions on indexed data structures, since our ultimate aim, left to future papers, is to port the polytope model [Len93] for the parallelization of repetitive algorithms to the domain of functional programming.

In the case of the array, the maintenance of the single-assignment property is particularly painful since each update results in a copy of the entire array, even though very little information has been altered. Several solutions to this problem have been proposed: arrays as trees [Wis92], version arrays using trailers [Blo89], in-place update analysis [FO95], etc. However, there is no representation whose operations are all of constant time complexity, unless one resorts to special-purpose computer architectures [O'D93]. In imperative programs, arrays are manipulated by in-place updates of individual elements. That is, imperative arrays are *single-threaded*: after an update, the original version of the array is lost. In contrast, a functional program may contain a *multi-threaded* array A: it might update A, naming the result B, and subsequently use both A and B. A compiler which can recognize single-threaded accesses of an array can implement them by in-place updates. For multi-threaded accesses, additional expense in time and space cannot be avoided. In the case of Id, the language simply gives up the property of referential transparency by including "I-structures", which resemble arrays [ANP89].

One can prevent multi-threaded array accesses by updating the array inside a monad [Wad92a, Wad92b], which has the disadvantage of obscuring the algorithm by using an imperative-like programming style within a functional program.

This leads us to conclude that incremental array updates should be avoided. The good news is that they need not be part of the specification, but rather tend to crop up in the imperative program even though they often could be avoided.

In the following we present two examples: one in which incremental updates are inherent, and one in which they are not, but nevertheless appear in solutions in the literature.

3 Case Study 1: Reflexive Transitive Closure

The first example deals with the calculation of the reflexive transitive closure (RTC) of a directed graph, which is expected to be sparse.

3.1 Problem Description

Given a set $V = \{1, \ldots, n\}$ of nodes and n lists $S_1, \ldots, S_n$ of successors for each node, we want to compute the sets R_i, of reachable nodes, for every node i.

One of the key problems is to decide which data structure to use for the representation of the graph. Two obvious choices are the adjacency set and the incidence matrix. In today's functional languages, the former would be implemented as a list, the latter as an array – although one might choose a combination of both. This choice has an influence on the structure and efficiency of the program.

Say we prefer the adjacency set. A possible algorithm is presented in Figure 1. Generally, this algorithm is viewed as the specification of the RTC problem. Clearly, we could look for a more abstract description, but we have not found one that seems more natural, so let us accept Figure 1 as our starting point.

FOR $v \in V$:
    $Todo_v := \{v\}$;
    $RTC_v := \{\}$;
    WHILE $Todo_v \neq \{\}$:
        $x =$ some OLD $Todo_v$;
        $Todo_v =$ OLD $Todo_v \setminus \{x\}$;
        IF $x \notin RTC_v$:
            $RTC_v :=$ OLD $RTC_v \cup \{x\}$
            $Todo_v :=$ OLD $Todo_v \cup S_x$

Figure 1: The list-based algorithm

One important question is whether this problem description is imperative or not. If it is, there is little point in reverting to a functional program. In our view, this algorithm is not biased to any paradigm since it just describes operations on mathematical sets. To avoid name clashes, we distinguished a set from its predecessor in the update history by annotations with `OLD` á la Sisal [BOCF92]. An `OLD` set is the set with the same name from the previous loop iteration. With this convention, all names are unique that the program has the single assignment property. The loop construct is not inherently imperative. It simply prescribes repetition and corresponds to tail recursion, as we shall see in the following section.

3.2 Functional Implementation

This gives us the license to proceed with our implementation in the functional programming paradigm. The nature of the RTC algorithm – namely adding nodes to a set and testing for membership – seems to favor the list data structure. If we choose it, the algorithm in Figure 1 translates directly into a functional program. If we chose Lucid or Sisal, we would have to change little. We are working with Haskell [PHe96], a richer functional language. A Haskell implementation of the RTC algorithm is in Figure 2. The graph is represented as a list of lists of integers. The integer list at position i contains the numbers of the successor nodes of node i. The loops have been replaced by recursion, which eliminates the need for the annotations with `OLD`.

In the central part of `bfsearch`, function `elem` tests for element membership of `actnode` in set `marked`. If `actnode` is not in the set, the node has not been visited yet and a recursive call appends all successors of `actnode` to list `Todo`. In Haskell, `++` denotes list concatenation and `a!!b` returns the `b`th element of list `a`.

This very succinct implementation has one drawback: it requires an additional list, `marked`, to keep track of the nodes already visited. The lookup takes linear time in the length of list `marked`. An imperative implementation would use a boolean array to store a flag for each node. The use of an array in the functional implementation would not only obscure the algorithm, since it would

```
closure :: [[Int]] -> [[Int]]
closure graph = [ bfsearch [i] [] graph  | i<-[0..length graph -1]]

bfsearch :: [Int] -> [Int] -> [[Int]] -> [Int]
bfsearch [] marked graph   = marked
bfsearch (actnode:rest) marked graph
   | actnode `elem` marked = bfsearch rest marked graph
   | otherwise             = bfsearch (rest++(graph!!actnode))
                                   (actnode:marked) graph
```

Figure 2: Closure in concise list form

require additional index calculations, but, even worse, might result in a loss of efficiency, since the array would need to be updated incrementally.

A modification of the list-of-list data structure into a two-dimensional array poses no problems, since the array can be constructed in one step: the elements in each row are defined independently and no element needs to be updated. But we must calculate index positions, as opposed to just appending new elements to a list. Even the data structure `graph` may just as well be an array, although we do not need the full power of array indexing. We only want to access all successors of a node in one operation, as we do using the list indexing operator `!!`. In the next section, we propose a new data structure better tailored to our needs.

In the presence of recursion, a copy of the data structure for an update can only be avoided if the compiler avoids copying the parameters of a recursive call, e.g., by converting tail recursion to iteration.

3.3 A New Data Structure: The *Indexed List*

These observations lead us to a new functional data structure: an *indexed list*, which combines the advantages of the list and the array:

- It has constant-time element access via indexing.
- It does not impose predefined bounds where they are not needed (e.g., in the RTC example).
- A read access to an element requires indexing, but a write access does not – it is realized by a list append instead.

These are snoc lists [GG97]: elements are consed at the end, thus leaving the index positions of previously consed elements unchanged. This does not violate the single-assignment rule if applied elementwise: each element is given a value only once. One proviso is that the program does not refer explicitly to the size of the list – which is, of course, changed with each addition of a new element.

In order to obtain constant-time lookup without having to scan through the list, we require an array-like, contiguous memory layout. The problem here is the unknown upper bound. If we store this data structure on a heap, a growing indexed list requires increasing amounts of space, forcing local memory reorganization. One might consider heuristics such as initially reserving some additional space at the end of the list and, as reorganizations take place, doubling the amount of added free space.

In the implementation of an indexed list, the list entries must be of fixed size in order to avoid unnecessary reorganizations. Dynamic data structures should be placed elsewhere on the heap with a pointer in the indexed list; Nested indexed lists could be implemented this way and mimic dynamic multi-dimensional arrays.

A different issue is that of destructive updates on the indexed list. This calls for an additional indexed write operation. In Haskell, a destructive update of a data structure can only be realized by encapsulating the data structure in a monad. In fact, King [Kin96] proposes the use of monads for functional implementations of graph algorithms. Unfortunately, at present, monadic extensions for an efficient use of arrays are compiler-dependent and cumbersome, and it has been argued that they obscure the functional nature of the program. The motivation for monads is preserving referential transparency while using side-effecting operations (this explanation may vary with a different definition of side effects). Clean [BvEvL+87] provides instead the concept of a unique type, Sisal a means for single-threading accesses to indexed data structures in iterative loop constructs.

The indexed list is related to a couple of other data structures discussed in the literature:

- It is a generalization of the *stream* [ASS96], the classical model of an imperative variable in lazy functional programs. A stream is a list whose last element represents the current value of the imperative variable. In a stream, position i will usually not be accessed once position $i+1$ has been defined. The indexed list does not satisfy this condition.
- It is a special case of another indexed data structure with fuzzy extent, the *data field* [Lis96]. The data field is much more general, and the details of its efficient implementation still need to be worked out.

4 Case Study 2: LU Decomposition

Our second example is the LU decomposition of a non-singular square matrix $A = (a_{ij})$, $(i, j = 1, \ldots, n)$.

4.1 Problem Description

The result consists of one lower triangular matrix $L = (l_{ij})$, with unit diagonal, and one upper triangular matrix $U = (u_{ij})$ whose product must be A.

L and U are defined constructively as follows [Ger78]:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad j \le i, \quad i = 1, 2, \ldots, n \tag{1}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}}, \quad j > i, \quad j = 2, \ldots, n \tag{2}$$

Two aspects of this problem are worth mentioning. First, both arrays are defined constructively without the necessity of additional data structures. This suits our quest for an implementation which avoids incremental array updates. Second, the recurrence equations are mutually recursive, which favors a lazy implementation where values can already be referred to while the evaluation of the data structure is still in progress.

A lazy implementation is very easy to program, since the evaluation order of the array elements is determined at run time, but one pays for this convenience in efficiency. In the case of LU decomposition, the evaluation order could be determined at compile time – with a slightly more complex program, since the order would have to be made explicit.

4.2 Functional Implementation

Therefore, this example can be transformed directly into a Haskell program; see Figure 3. One only has to work out the precise ranges of j for L and i for U since the mathematical description is a bit fuzzy here.

In this case, the transformation from the mathematical specification to the implementation is almost immediate and appears just like a change of notation. A syntactically even closer transformation would be to ALPHA [GMQS89]. However, ALPHA is meant for chip layout, not for scientific programs, which is why it lacks advanced control structures and data types.

Even though we are using Haskell arrays, there is no efficiency penalty with respect to imperative arrays, since the algorithm avoids incremental updates.

```
lu_decomp:: Array (Int,Int) Float -> (Array (Int,Int) Float,
                                      Array (Int,Int) Float)
lu_decomp a = (l ,u)
  where
   l:: Array (Int,Int) Float
   l = array ((1,1), (n,n))
     [ ((i,j), a!(i,j) - sum [ l!(i,k)*u!(k,j)
                             | k <- [1..(j-1)]])
      | i <- [1..n],
        j <- [1..i] ]

   u:: Array (Int,Int) Float
   u = array ((1,2), (n,n))
     [ ((i,j), (a!(i,j) - sum [ l!(i,k)*u!(k,j)
                              | k <- [1..(i-1)]])
               / l!(i,i) )
      | j <- [2..n],
        i <- [1..(j-1)] ]

   (_ , (n, _)) = bounds a
```

Figure 3: Haskell code for LU decomposition

4.3 Imperative Implementation

Why do we prefer this functional to an imperative implementation? Programmers of scientific algorithms are likely to take the position that they do not need any support in deriving a program from the mathematical problem description. The answer is given by the Fortran implementation for LU decomposition from the same source [Ger78] from which Equations 1 and 2 are taken; see Figure 4.

The mere fact that this implementation has not been derived directly from the specification makes it more difficult to obtain. It also raises the possibility for errors in the program development. The need for the first two initializing `DO` loops arise from restrictions in the language, not from the specification. In the Haskell program, the initialization is not separate. Determining all loop bounds in order to meet the restrictions imposed by the data flow is an activity prone to error. The Haskell program simply relies on lazy evaluation and does not specify them. Another restriction of Fortran, the absence of expressions containing control structures, forces the programmer to insert auxiliary variables such as for the intermediate values when accumulating the sum.

All of the above supports our claim that the process of deriving the program in Fortran must have been, compared to Haskell, longer and more error-prone. In addition, the Fortran program lacks similarity with the specification, burdening the programmer with a more difficult correctness proof.

5 Conclusions

Let us summarize our claims and findings:

- Scientific problems are usually described by a repetitive algorithm on an indexed data structure.

- For historical reasons, in the literature, the algorithm is usually presented with a bias towards the imperative programming paradigm, but this already puts it at some distance from the

```
DO FOR I = 1 to n step 1,
   L(I,1) = A(I,1)
ENDDO
DO FOR J = 1 to n step 1,
   U(1,J) = A(1,J)/L(1,1)
ENDDO
DO FOR J = 2 to N step 1,
   DO FOR I = J to n step 1,
      DO FOR K = 1 to J-1 step 1,
         Accumulate sum of L(I,K)*U(K,J).
      ENDDO.
      L(I,J) = A(I,J) - SUM.
   ENDDO.
   U(J,J) = 1.
   DO FOR I = J + 1 to n step 1,
      DO FOR K = 1 to J - 1 step 1,
         Accumulate sum of L(J,K)*U(K,J).
      ENDDO.
      U(J,I) = [A(J,I) - SUM]/L(J,J).
   ENDDO.
ENDDO.
```

Figure 4: LU decomposition in Fortran

abstract problem description, which corresponds to a set of recurrence equations.

- A functional program can be much closer to the abstract description, shortening the development path, offering design alternatives, providing guidance for the design choices and reducing the danger of errors.

- In many cases, the array is the suitable indexed data structure – even in the functional programming paradigm. The list type lacks efficient indexed access. We distinguish three array types:

  1. The *single-assignment* array. Its elements are all defined in one step. Examples are `L` and `U` in our LU decomposition program.
  2. The *single-threaded* array. Its elements must be defined in several steps which, however, can be put into a total order. An example is array `marked` in our functional RTC program. For an efficient internal implementation, this type of array requires destructive updates, which complicate things but can be achieved in the functional programming paradigm.
  3. The *multi-threaded* array. Its updates cannot be put into a total order. This type of array cannot be implemented efficiently on general-purpose computers and need not arise in scientific computing. It does not arise in imperative programming.

  The goal is to avoid unnecessary generality which impairs efficiency. Multithreading, overloaded index types and arbitrary element types – to name a few – are not needed in our problem domain.

- Laziness helps a great deal in keeping the functional program simple and allows us to work with data structures whose extent is not known at compile time. We have proposed a new data structure, the *indexed list*, for the representation of dynamic single-threaded arrays. The imperative programming paradigm does not offer a simple way of expressing laziness.

Our overall conclusion is that functional programming can offer significant support in the development of scientific algorithms. Although the current support for indexed data structures is still not satisfactory, a possible loss in efficiency may be outweighed by the gain in ease and reliability of programming.

6 Acknowledgments

Thanks to Simon Peyton-Jones, Paul Feautrier, John O'Donnell and Sergei Gorlatch for fruitful discussions and John O'Donnell and Noel Winstanley for comments on the first version of this paper. Thanks to Christoph Herrmann for the original code in Figure 2. Financial support was received from the DAAD under its ARC and PROCOPE programmes.

References

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press / McGraw-Hill, 2nd edition, 1996.

[Blo89] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In J. Stoy, editor, *Proc. of the Fourth International Conf. for Functional prog. languages and Comp. Architecture (FPCA'89)*, pages 26–38. ACM press, September 1989.

[BOCF92] A. P. Wim Böhm, Rodney R. Oldehoeft, David C. Cann, and John T. Feo. *SISAL Reference Manual, Language Version 2.0.* Colorado State University – Lawrence Livermore National Laboratory, 1992.

[BvEvL+87] T. Brus, M. van Eekelen, M. van Leer, M.J. Plasmeijer, and H.P. Barendregt. Clean – a language for functional graph rewriting. In *Proc. of the Third International Conf. for Functional Prog. Languages and Comp. Architectures (FPCA '87)*, Lecture Notes in Computer Science 274, pages 364–384. Springer-Verlag, 1987.

[Can92] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8), August 1992.

[FO95] Steve Fitzgerald and Rodney R. Oldehoeft. Update-in-place analysis for true multidimensional arrays. In A. P. Wim Böhm and John T. Feo, editors, *High Performance Functional Computing*, pages 105–118, April 1995.

[Ger78] Curtis F. Gerald. *Applied Numerical Analysis.* Addison-Wesley, 2nd edition, 1978.

[GG97] Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. In M. Hanus, J. Heering, and K. Meinke, editors, *Algebraic and Logic Programming (ALP'97)*, Lecture Notes in Computer Science 1298, pages 46–60. Springer-Verlag, 1997.

[GMQS89] Pierrick Gachet, Christophe Mauras, Patrice Quinton, and Yannick Saouter. A Language for the Design of Regular Parallel Algorithms. In *First European Workshop on Hypercube and Distributed Computers*, pages 189–202. North-Holland, October 1989.

[HFA+96] Pieter Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, and Peter Baumann et al. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[Kin96] David J. King. *Functional Programming and Graph Algorithms.* PhD thesis, Department of Computing Science, University of Glasgow, 1996.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[Lis96] Björn Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 220–251. Springer-Verlag, June 1996.

[O'D93] John O'Donnell. Data parallel implementation of extensible sparse functional arrays. In *Parallel Architectures and Languages Europe (PARLE'93)*, Lecture Notes in Computer Science 694. Springer-Verlag, 1993.

[PHe96] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996.

[Rea89] Chris Reade. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, 1989.

[Wad92a] Philip Wadler. Monads for functional programming. In *Lecture Notes for Marktoberdorf Summer School on Program Design Calculi*, pages 233–264. Springer-Verlag, August 1992.

[Wad92b] Philip Wadler. The essence of functional programming. In *ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14. ACM Press, January 1992.

[Wis92] David S. Wise. Matrix algorithms using quadtrees. In Gaétan Hains and Leonore M.R. Mullin, editors, *ATABLE-92: Second Int'l Workshop on Array Structures*, Technical Report Nr. 841. University of Montreal, November 1992.