Experiments with destructive updates in a lazy functional language

Pieter H. Hartel Willem G. Vree

Department of Computer Systems, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
Email: {pieter,wimv}@fwi.uva.nl

Abstract

The aggregate update problem has received considerable attention since pure functional programming languages were recognised as an interesting research topic. There is extensive literature in this area, which proposes a wide variety of solutions. We have tried to apply some of the proposed solutions to our own applications to see how these solutions work in practice. We have been able to use destructive updates but are not convinced that this could have been achieved without application specific knowledge. In particular, no form of update analysis has been reported that is applicable to non-flat domains in polymorphic languages with higher order functions.

It is our belief that a refinement of the monolithic approach towards constructing arrays may be a good alternative to using the incremental approach with destructive updates.

Keywords: lazy functional languages, array updates, compilation, annotation, measurements.

1 Introduction

In a pure functional language it is difficult to implement an update on an aggregate data structure in an efficient way. The problem is that unless the aggregate is known to be single threaded [1], a copy must be made of the entire aggregate, which may then be destructively updated. Otherwise the vitally important referential transparency property is lost. For the sake of definiteness we will concentrate on the use of a particular aggregate data structure: the array. We study three lazy functional programs that use arrays. In constructing these programs we have tried to apply the techniques proposed in the literature to achieve the best performance that arrays in a lazy functional language have to offer. The focus is on practical aspects. Sometimes an appropriate analysis is able to discover essential information required to achieve a high performance, and sometimes this cannot be achieved without the help of the programmer. Each example is programmed in various styles of using arrays. Measurements are reported in using an implementation that supports a wide range of different array primitives and implementations.

There are three basic operations to be implemented on arrays: create, subscript and update. An array can be implemented in many ways, but our preferred implementation is a *container* [2], which is just a contiguous block of memory. The container holds either the actual array elements, or pointers to the array elements. This depends on the implementation of the functional language. The use of a container of size N permits $O(1)$ subscript time, and $O(N)$ creation time. The problem is to also implement $O(1)$ update time.

There are two basically different ways of using arrays: incrementally or monolithically [3]. When the calculations involved in creating an array follow some kind of regular pattern, a monolithic operation should be appropriate: this creates an entire array in one single operation. When no sufficiently simple regularity can be discerned, an incremental approach is required: this repeatedly updates the current version of the array in one or more places to form the next version. Such updates are impossible with the monolithic approach. Monolithic array operations can be expressed in terms of incremental operations and vice versa, so the expressive power of both approaches is the same [4]. Depending on the implementation, there may be large differences in space and time complexity.

The incremental approach is of a lower level whereas the monolithic approach is of a higher level of abstraction. This favours the monolithic approach, and indeed some researchers have noted that programs using the monolithic approach are clearer [5]. The programming style that should be followed when writing functional programs using arrays should aim for using monolithic array operations. This is consistent with a commonly accepted preference for the use of higher order functions rather than explicit recursions [6].

Having said this, is the aggregate update problem still a major issue? Unfortunately, the answer is yes. Consider a monolithic array operation that transforms an old array into a new one, for instance by applying a certain function to all the array elements. This also presents the array update problem, for the result of the operation requires a container that has the same size as the container holding the input array. So the implementation should attempt to reuse that container [4].

There is a difference between the two manifestations of the aggregate update problem. Suppose that a certain algorithm requires an array of size N to be updated M times in its entirety. A naive incremental implementation, which copies the array upon each update, requires $O(N^2 \times M)$ space, whereas a naive monolithic implementation requires $O(N \times M)$ space. The naive incremental implementation is thus worse than the naive monolithic implementation. For some applications, even a naive monolithic implementation may be acceptable, if the work involved in calculating the $O(N \times M)$ array elements is sufficiently large.

In a non-strict functional language, both the monolithic and the incremental arrays have a further problem, because in such a language also the arrays are non-strict. This means that arrays may have undefined elements. Unless strictness analysis can prove otherwise, which is hard, array elements must be created as suspended computations and unfortunately suspensions are costly. Consider the creation of an array of N integers, as generated by N distinct function calls, say $(f\ 1) \ldots (f\ N)$. In a strict language this requires a container of size N integers. In a non-strict language, a container of the same size is required to store pointers to N suspensions. Each of these suspensions requires space capable of holding at least 2 pointers (one to point at the function and one to point at the argument) but often more. Let us assume that a pointer requires the same amount of space as an integer, then in total the non-strict array requires at least $3N$ space. The time required to manipulate a non-strict array is greater, not only because more space needs to be allocated but also because the storage occupied by $N + 1$ logically separate objects (1 container + N suspensions) must be collected. The time complexity of allocating and recovering a strict array is thus $O(1)$, and the time complexity for handling a non-strict array is $O(N)$.

The problem of efficiently implementing arrays in a non-strict language is further compounded by the requirement that at runtime it must be possible to distinguish evaluated objects from unevaluated objects. Sometimes the compiler already knows that an expression is evaluated, and so it may generate code that avoids testing at runtime whether the expression has or has not been evaluated. The expressions that are known to be evaluated may be represented as *unboxed* objects [7]. Expressions that are not known to be evaluated must be represented by boxed objects. An important disadvantage of boxed objects is that they require some extra information, which takes up space. Boxing analysis is just as difficult as strictness analysis, thus one might expect that the compiler cannot always avoid the creation of boxed objects.

The discussion of boxed and unboxed representations suggests that in an implementation there might be two different representations of an array (of integers say). The first is the fully boxed representation such as it has been described above. A second, more efficient representation allows for unboxed components of the array. This means that the components of the array reside in the container, just as in an implementation of a strict functional language or an imperative language. Such a representation is only safe if the compiler knows that all the array components are evaluated, no matter where and when they are used. Even if only one component cannot be proved to be evaluated, all components must be represented as boxed values. This means that an array with unboxed components must be implemented as a container with pointers to separate heap cells. These cells may then either contain a suspension, or after evaluation and updating, a boxed integer.

The destructive update problem in lazy functional languages has a wide range of aspects, including strictness analysis, boxing analysis and subscript analysis [4], as well as the question of whether to use incremental or monolithic array primitives. The difficulty in implementing destructive updates lies not so much in developing the right kind of analysis for it, but more in the problems raised by combining the many different analyses, which all have to work together in harmony to achieve good results.

The destructive update problem can be avoided in a number of ways. The two most promising approaches are based on monads [8] or on unique types [9]. Both approaches ultimately use the

type system to guarantee that a data structure is single threaded, and that destructive updates are therefore safe. These approaches are interesting because no clever analysis is required to detect single threadedness. It is annotated (but in very different ways) by the programmer and the annotations are verified by the system. These two alternative approaches thus represent a safe way to express imperative programming concepts in a functional context. We shall not consider these alternatives here, because our main interest is in contrasting the imperative and functional style. The former is represented by the incremental approach to array handling and the latter by the monolithic approach. In our view the monadic approach, the unique typing approach, and the analysis approach such as we have used, have the same net effect on the handling of arrays.

Also, because our main interest is in contrasting the imperative and functional style, we do not consider programs that cannot be implemented efficiently in a monolithic style. An example of such a program is in updating a large data base.

We are well aware of the fact that for many algorithms efficient functional implementations exists that work on lists, as opposed to arrays. In our earlier paper [10] we have investigated a number of different versions of the fast Fourier transform. We found that list based implementations can be efficient, but they are beyond the scope of our present focus on arrays. Such list based implementations will not be considered here.

In Section 2, an example program is taken from the literature and two other example problems are presented in some detail. Measurements are reported in Section 3. The last section presents the conclusions.

2 Example problems

The three examples are the ubiquitous quick sort program (`qs`), the fast Fourier transform (`fft`) and a tidal prediction program (`wv`), which simulates the behaviour of an estuary of the North Sea over a number of time steps. The monolithic version of `qs` is taken from Wadler's paper [11], the incremental version originates from Hudak's paper [12]. The `fft` and `wv` programs will be described in the following sections. The array primitives being used are borrowed from the languages Tale [13] and Haskell [14]. Angular brackets are used here to denote an array thus: $\langle a_l \ldots a_u \rangle$. All arrays are accompanied by a descriptor pair (l, u), which holds the lower bound l and the upper bound u of the array. Here are two examples of array primitives. The first is the function *tabulate*, taken from the language Tale, which allocates a block of memory with $u-l+1$ references to applications of the function f thus: $(f\ l), (f\ (l+1))$ etc:

$$\begin{array}{l} \mathit{tabulate}\ ::\ (\mathit{int} \to \alpha) \to (\mathit{int} \times \mathit{int}) \to \mathit{array\ of}\ \alpha \\ \mathit{tabulate}\ f\ (l,u)\ =\ \langle (f\ l) \ldots (f\ u) \rangle \end{array}$$

The second example function *accum* is from Haskell:

$$\begin{array}{l} \mathit{accum}\ ::\ (\alpha \to \beta \to \alpha) \to \mathit{array\ of}\ \alpha \to \mathit{list\ of}\ (\mathit{int} \times \beta) \to \mathit{array\ of}\ \alpha \\ \mathit{accum}\ f\ \langle a_l \ldots a_u \rangle\ vs\ = \langle (\mathit{foldl}\ f\ a_l\ [v|(i,v) \leftarrow vs; i = l]) \ldots (\mathit{foldl}\ f\ a_u\ [v|(i,v) \leftarrow vs; i = u]) \rangle \end{array}$$

2.1 The discrete Fourier transform

The discrete Fourier transform of N complex data items is defined as follows:

$$x'_j = \sum_{k=0}^{N-1} x_k \times w^{jk} \qquad \text{for } j = 0 \ldots N-1$$
$$\text{where } w = e^{2\pi i/N}$$

The N input elements x_j are transformed into N output values x'_j. A straightforward algorithm to compute the discrete Fourier transform would require $O(N^2)$ steps. Cooley and Tukey [15] published the `fft` algorithm, which completes the required calculation in $O(N \times \log N)$ steps, provided that N is a power of 2. Figure 1 illustrates the data flow when an array of 4 elements is processed by the `fft`.

The input $\langle x_0 \ldots x_{N-1} \rangle$ is transformed to $\langle x'_0 \ldots x'_{N-1} \rangle$ in $\log_2 N$ (= 2) steps. Each step calculates an intermediate array from its input values, shown as levels 0 and 1 in Figure 1. On each level, the input array is fed in pairs into $N \div 2$ independent calculations, shown as cells in Figure 1. The computation

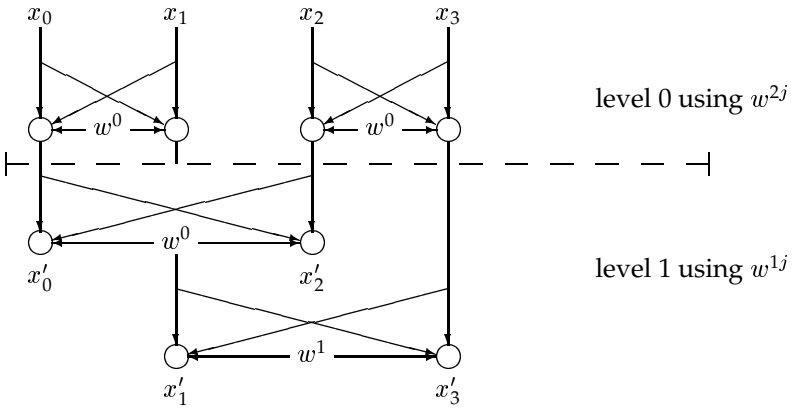

Figure 1: The data flow in a 4-point fast Fourier transform.

within each cell, called a butterfly because of its graphical appearance, is the function *bfly* shown at the bottom of Figure 2. In Figure 1, a butterfly is shown as two circles, each connected by three arrows. What changes from level to level is the span of each butterfly ($k = j + 2^{level}$) and the positions where butterflies are applied. The input elements at level 0 must appear in a particular order, but we will not be concerned with this aspect of the program here (See [10]).

The data flow of Figure 1 indicates that the left inputs of all butterflies are connected in a regular way to the outputs on the next higher level. The regularity is the following: the left wing of a butterfly at level m is connected to x_j at level $m-1$, when the binary representation of j has a zero in the m-th bit position. In Figure 1 for example, the left inputs of the butterflies at level 1 are connected to x_0 and x_1, because bit 1 is zero in 0 and 1.

The implementation in Figure 2 is based on the observed regularity. The first argument n of the function *fft* is used to compute the current level m. All levels will have been handled as soon as n reaches the value 0 and the recursion is then terminated. There are two implementations $level_i$ and $level_m$ of the function that expresses the butterfly applications on one level m. These two functions have the same semantics but differ in the way the array x is handled.

Both versions of *level* use the auxiliary functions *bflylist*, *bfly* and *bit* to perform the butterfly calculation. The function *bit* j m calculates the value of the m-th bit position of the integer j. The mentioned regular connections of the left inputs x_j of the butterflies are expressed in the program by the restriction *bit* j $m = 0$. The span of the butterflies at level m is 2^m, so the index for the right inputs x_k is $k = j + 2^m$.

The function $level_i$ is based on the incremental approach towards array construction. At each invocation of $level_i$, elements at positions j and k of the input array x are replaced by two new values v_j and v_k, which are destined for the same positions j and k. The way in which the replacement is effectuated (i.e. destructively or non-destructively) will be considered later.

The monolithic approach towards array construction is embodied in the function $level_m$, which replaces the entire array by a new one rather than piecemeal as in the incremental approach. The contents of the new array are produced by the list comprehension under the keyword *where* as the list l. This list is actually a list of two-element lists, which must therefore be flattened by the function *concat*. The association pairs thus produced will appear in some particular order that is not the index order of the array. This explains why the function *array* has been used, which builds an array out of a list of index value pairs. The index computations are based on the regularity inherent in the `fft` algorithm. Without such a regularity, it would not have been possible to use the monolithic approach at all.

The monolithic implementation creates a considerable amount of intermediate list structure, which a good compiler should be able to avoid completely [4, 16]. Unfortunately, the compiler that we have been using (FAST [17]) does not have this capability.

The definition of the function *reorder* has been omitted from the program of Figure 2 as it does not play a role in the discussion on updating. The interested reader is referred to our paper [10] for a complete description of the program.

$$
\begin{array}{l}
\textit{main} :: \textit{array of complex} \\
\textit{main} \quad = \textit{fft}\ (N \div 2)\ (\textit{reorder}\ N\ \textit{input}) \\
\\
\textit{fft} :: \textit{int} \to \textit{array of complex} \to \textit{array of complex} \\
\textit{fft}\ 0\ x \quad = x \\
\textit{fft}\ n\ x \quad = \textit{fft}\ (n \div 2)\ y \\
\qquad \text{where} \\
\qquad m = \log_2\ (N \div (n \times 2)) \\
\qquad y = \textit{level}\ 0\ m\ n\ x \qquad || \text{ Choose either } \textit{level}_m \text{ or } \textit{level}_i \\
\\
\textit{level}_i, \textit{level}_m :: \textit{int} \to \textit{int} \to \textit{int} \to \textit{array of complex} \to \textit{array of complex} \\
\textit{level}_i\ j\ m\ n\ x \quad = x, \quad \text{if } j = N \\
\qquad = \textit{level}_i\ (j+1)\ m\ n\ z, \quad \text{if } \textit{bit}\ j\ m = 0 \\
\qquad = \textit{level}_i\ (j+1)\ m\ n\ x, \quad \text{otherwise} \\
\qquad \text{where} \\
\qquad y = \textit{update}\ x\ j\ v_j \\
\qquad z = \textit{update}\ y\ k\ v_k \\
\qquad [(j', v_j), (k, v_k)] = \textit{bflylist}\ j\ m\ n\ x \\
\\
\textit{level}_m\ j\ m\ n\ x \quad = \textit{array}\ (\textit{bounds}\ x)\ (\textit{concat}\ l) \\
\qquad \text{where} \\
\qquad l = [\textit{bflylist}\ j\ m\ n\ x \mid j \leftarrow [0 \ldots N-1];\ \textit{bit}\ j\ m = 0] \\
\\
\textit{bflylist} :: \textit{int} \to \textit{int} \to \textit{int} \to \textit{array of complex} \to \textit{list of}\ (\textit{int} \times \textit{complex}) \\
\textit{bflylist}\ j\ m\ n\ x = [(j, v_j), (k, v_k)] \\
\qquad \text{where} \\
\qquad k = j + 2^m \\
\qquad (v_j, v_k) = \textit{bfly}\ (n \times j)\ (\textit{subscript}\ x\ j)\ (\textit{subscript}\ x\ k) \\
\\
\textit{bfly} :: \textit{int} \to \textit{complex} \to \textit{complex} \to (\textit{complex} \times \textit{complex}) \\
\textit{bfly}\ j\ x_j\ x_k \quad = (x_j + z, x_j - z) \\
\qquad \text{where} \\
\qquad z = x_k \times w^j
\end{array}
$$

Figure 2: Two implementations of the `fft`: $\textit{level}_m$ follows the monolithic approach and $\textit{level}_i$ follows the incremental approach towards array construction.

2.2 Can destructive update be made safe?

Depending on the choice made in the function *fft* in Figure 2, either the monolithic $level_m$ or the incremental $level_i$ is used. The time complexity of the destructively updating (incremental) and the monolithic implementations are the same: $O(N \times \log N)$. The non-destructively updating implementation can be ruled out, because it has a time complexity of $O(N^2 \times \log N)$. A selection can thus be based purely on the space behaviour of the two implementations.

The monolithic implementation of the `fft` makes $\log_2 N$ calls to *array* and therefore allocates $O(N \times \log N)$ space. The destructively updating implementation uses only $O(N)$ space. So the incremental approach is best, but only if the updates can be performed destructively. If this cannot be guaranteed, the implementation will require $O(N^2 \times \log N)$ space.

The question is: can this guarantee be given? Unfortunately, the answer is no, unless some precautionary measures are taken. Let us first investigate why the required guarantee is difficult to give, and then come back to the measures that have been proposed in the literature.

Without special measures, a compiler for a lazy functional language may generate code similar to that shown on the left of Figure 3 as the function $level_i$. The generated code has been expressed as a C program; functions are thus called by value. Only the essential parts of the C code have been shown, which include the array manipulations.

As one would expect in a lazy language, we have assumed that *update* is strict in its first two arguments (the array and the index) and non-strict in its third argument (the new value). The strictness analysis of the compiler is then able to discover the following fact: since the *update* function is strict in its first two arguments (the array and the index), both j and k are needed and will therefore represent integers, rather than pointers to suspensions in the heap. Similarly, the variables x, y, and z represent proper arrays rather than suspensions, which only produce arrays as soon as they are evaluated. The two *update* functions are therefore called rather than embedded in suspensions. On the other hand, the variables v_j and v_k are not needed and therefore point at suspensions in the heap, because of the non-strictness of the third argument (the value) of *update*. The functions *fst* and *snd* select the first, respectively the second component of the tuple returned by *bfly*. The variables x_j and x_k cannot be proved to be needed, so suspensions should be made of the *subscript* functions, as shown in the body of $level_i$. These two suspensions make it impossible to use destructive updates, because pointers emanating from these suspensions will refer to the array x. This problem has been described by Stoye [18] and some others. We will present a solution that works for the `fft` program, but unfortunately this solution does not always apply.

```
1.  level_i(j, m, n, x)
2.  {
3.      if (j = N) {
4.          return x;
5.      } else if (bit(j, m) = 0) {
6.          k   = j + 2^m;
7.          x_j = suspend(subscript, x, j);
8.          x_k = suspend(subscript, x, k);
9.          v_jk = suspend(bfly, n × j, x_j, x_k);
10.         v_j = suspend(fst, v_jk);
11.         v_k = suspend(snd, v_jk);
12.         y   = update(x, j, v_j);
13.         z   = update(y, k, v_k);
14.         return level_i(j + 1, m, n, z);
15.     } else {
16.         return level_i(j + 1, m, n, x);
17.     }
18. }
```

```
level'_i(j, m, n, x)
{
    if (j = N) {
        return x;
    } else if (bit(j, m) = 0) {
        k   = j + 2^m;
        x_j = subscript'(x, j);
        x_k = subscript'(x, k);
        v_jk = suspend(bfly, n × j, x_j, x_k);
        v_j = suspend(fst, v_jk);
        v_k = suspend(snd, v_jk);
        y   = update(x, j, v_j);
        z   = update(y, k, v_k);
        return level'_i(j + 1, m, n, z);
    } else {
        return level'_i(j + 1, m, n, x);
    }
}
```

Figure 3: The $level_i$ procedure taken from the C-code generated for the incremental version of the `fft`, showing on the left the code before and on the right the code after the cheap eagerness optimisation.

2.3 Possible solutions to the suspended subscript problem

The update function as used by Bloss [19] is strict in all three arguments, rather than in just the first two as is the case here. Depending on the sophistication of the strictness analyser, this may or may not avoid the problem we are presently facing. Should the strictness analysis be capable of reasoning about structured data, such as the tuple returned by the *bfly* function, the values of v_j and v_k will be known to be needed and so will the values of x_j and x_k. It is possible to build a sufficiently sophisticated strictness analyser to prove the neededness of x_j and x_k, but the FAST compiler does not offer this.

An alternative approach has been implemented in the FAST compiler [20]. Even with a fairly simple strictness analyser, and when using a version of *update* that is non-strict in its third argument, it is still possible to use a destructive update. To achieve this, an optimisation called *cheap eagerness* is used: instead of building a suspension for a selector function and using the pointer to the suspension, cheap eagerness selects the required item and uses a pointer to that item. These two approaches are schematically shown in Figure 4. Here a points at the suspension of the *subscript* function and b points at the item to be selected. The arguments of *subscript* are an array x and an index i. The cheap eagerness optimisation is also used in the LML compiler [21].

Assume that both the array and the index are head normal forms, that is, the index is an integer and the array is a block of memory containing pointers to the actual elements. Whether the array elements themselves are (head) normal forms or not is irrelevant. It is now possible to use the pointer b instead of the pointer a and thus avoid building the suspension of *subscript*. This amounts to selecting a certain element from a container while is is not known whether the resulting value will ever be needed. This is thus non-lazy. However, in the present case the amount of work involved in selecting the element, is less than the amount of work required to build the suspension for *subscript*. So even though the optimised code is not lazy, it is arguably more efficient and therefore desirable to have. It should be noted that the optimisation does not evaluate the selected item (in the dashed box), as that may involve an unlimited amount of computation, which would defeat the whole objective.

The cheap eagerness optimisation can only be applied if the compiler is able to prove that a limited form of eager evaluation requires less work than lazy evaluation. In the example of Figure 4, the compiler must thus know that both the index and the container of the array (but not necessarily the actual array elements) have indeed been evaluated sufficiently by the time the subscript operations at lines 7 and 8 in Figure 3 are executed. The cheap eagerness optimisation can be applied to all selector functions and most arithmetic functions.

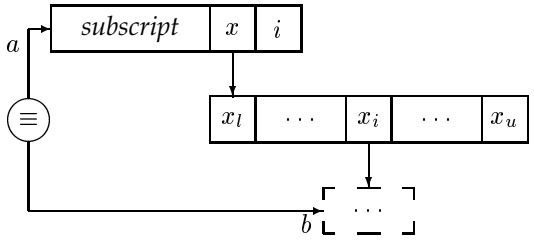

Figure 4: Applying the cheap eagerness optimisation to array subscription. The boxes represent heap cells.

Returning to the C-code of Figure 3 it becomes apparent that the cheap eagerness optimisation can be applied to the two subscriptions $x_j = \ldots$ and $x_k = \ldots$. During strictness analysis, the compiler has proved that both the array x and the two indices j and k are indeed evaluated by the time the suspensions of *subscript* are made (see lines 7 and 8 in Figure 3). The optimisation thus enables the two statements that build the suspensions of *subscript* to be repaced by calls to a special function *subscript*$'$. The latter differs from the regular *subscript* in that it does not guarantee to return a head normal form; *subscript*$'$ merely returns (a pointer to) the selected item as found. The optimised version $level'_i$ is shown on the right of Figure 3.

After applying the cheap eagerness optimisation to the `fft` program, destructive updates are safe, because the array of complex numbers is now single threaded. We have thus achieved our goal, without having to modify the incremental version of the `fft` program. However, this particular version had been prepared with the capabilities of the compiler in mind. This gave the function $level_i$ its present

form and helped the strictness analyser to discover that $level_i$ is strict in its first and its last argument, which happens to be just the information needed to make the cheap eagerness optimisation work. Cheap eagerness is a general optimisation, which applies many primitive as well as user defined functions. However, because it is an optimisation, cheap eagerness cannot be guaranteed to solve the problem of the suspended subscripts in other cases. As many researchers have observed, a small change to either the program or the compiler and the single threadedness is lost again, so that the performance of an incremental program deteriorates considerably because destructive updates are no longer safe.

The monolithic version of the `fft` can also be made to destructively update the array. This requires the call to *array* in $level_m$ to be replaced by a call to *accum*, so that access to the old array x is granted, yielding:

$$
\begin{aligned}
&level'_m\ j\ m\ n\ x = accum\ (\lambda xy.y)\ x\ (concat\ l) \\
&\qquad\qquad \text{where} \\
&\qquad\qquad l = [bflylist\ j\ m\ n\ x \mid j \leftarrow [0 \ldots N-1];\ bit\ j\ m = 0]
\end{aligned}
$$

For the new function $level'_m$ to indeed reuse the container of the array x, the elements of the array must be read out two at a time. The two values must then be run through the *bfly* function to deliver the two new elements, which are then written in place of the old elements. The subscript analysis described by Anderson and Hudak [4] should be capable of achieving this. However, this requires a very powerful compiler and a programmer who knows exactly how to express a problem such that it can be recognised. These are both hard to achieve.

2.4 Tidal prediction

The second example that we will study is a tidal prediction. This program is also based on previous work [22, 23], which explains in some detail the physical background of the application. To make the present paper reasonably self contained, we discuss the essential aspects of the application domain, which leads to a formulation of an almost single threaded implementation. For a comprehensive treatment of the application domain, the reader is referred to Heemink's thesis [24].

The prediction of the tides in the North Sea and its estuaries are based on the *shallow water equations*. The linearised and slightly simplified versions of these equations [24] are shown below:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} - fv + \lambda\frac{u}{D} - \gamma\frac{V^2\cos\psi}{D} = 0 \tag{1}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial y} + fu + \lambda\frac{v}{D} - \gamma\frac{V^2\sin\psi}{D} = 0 \tag{2}$$

$$\frac{\partial h}{\partial t} + \frac{\partial (Du)}{\partial x} + \frac{\partial (Dv)}{\partial y} = 0 \tag{3}$$

h =small variations in the water height
D=depth of the water as a function of x and y
f =Coriolis parameter($1.25 \times 10^{-4} s^{-1}$)
γ =wind friction($\approx 3.2 \times 10^{-6}$)
ψ =wind direction
u =water velocity in the x direction
v =water velocity in the y direction
g =acceleration of gravity($9.8 ms^{-2}$)
λ =bottom friction($\approx 2.4 \times 10^{-3} ms^{-1}$)
V=wind velocity

Equations 1 and 2 represent the conservation of momentum and Equation 3 represents the conservation of mass. The effects of the earth rotation (f), the friction with the bottom (λ) and the friction with the wind (γ) are taken into account. The effect of the atmospheric pressure has been ignored because it is relatively small.

A numerical approximation for the partial differential Equations 1–3 is given in the finite difference scheme of Equations 4–6 below. For a formal derivation see van der Houwen [25]. The variables u, v and h in Equations 1–3 are approximated in Equations 4–6 by values $u^k_{i,j}$, $v^k_{i,j}$ and $h^k_{i,j}$ on a spatial grid at discrete points in time. The grid coordinates are i and j and the superscript k represents the discrete time steps. The depth D does not vary in time ($h^k_{i,j}$ is varying), hence no superscript k is required. When $\triangle x$ and $\triangle y$ are the distance steps in the grid, the relations between the true and approximated height and velocities are:

$$u^k_{i,j} \approx u\ ((2i-1)\triangle x, 2j\triangle y, \quad k\triangle t)$$
$$v^k_{i,j} \approx v\ (2i\triangle x, \quad (2j-1)\triangle y, k\triangle t)$$
$$h^k_{i,j} \approx u\ (2i\triangle x, 2j\triangle y, k\triangle t)$$
$$D_{i,j} = D\ (2i\triangle x, 2j\triangle y)$$

The spatial grids for u, v and h are slightly shifted with respect to each other, in about the same way as the red, green and blue dots that correspond to the pixels of a colour monitor. This alignment of the matrices is called a space staggered grid, and it has important advantages for the stability of the computations. See van der Houwen [25] for further details. Because of the choice of a space staggered grid the equations for $u^k_{i,j}$ and $v^k_{i,j}$ are asymmetrical:

$$\begin{aligned} u^{k+1}_{i,j} &= u^k_{i,j} - g\frac{\triangle t}{2\triangle x}(h^k_{i,j} - h^k_{i-1,j}) + f\frac{\triangle t}{4}(v^k_{i-1,j} + v^k_{i-1,j+1} + v^k_{i,j} + v^k_{i,j+1}) \\ &\quad -2\triangle t\frac{\lambda u^k_{i,j} - \gamma V^2\cos\psi}{D_{i,j} + D_{i,j+1}} \end{aligned} \tag{4}$$

$$\begin{aligned} v^{k+1}_{i,j} &= v^k_{i,j} - g\frac{\triangle t}{2\triangle y}(h^k_{i,j} - h^k_{i,j-1}) - f\frac{\triangle t}{4}(u^{k+1}_{i,j-1} + u^{k+1}_{i+1,j-1} + u^{k+1}_{i,j} + u^{k+1}_{i+1,j}) \\ &\quad -2\triangle t\frac{\lambda v^k_{i,j} - \gamma V^2\sin\psi}{D_{i,j} + D_{i+1,j}} \end{aligned} \tag{5}$$

$$\begin{aligned} h^{k+1}_{i,j} &= h^k_{i,j} - \frac{\triangle t}{4\triangle x}\left\{(D_{i+1,j} + D_{i+1,j+1})u^{k+1}_{i+1,j} - (D_{i,j} + D_{i,j+1})u^{k+1}_{i,j}\right\} \\ &\quad -\frac{\triangle t}{4\triangle y}\left\{(D_{i,j+1} + D_{i+1,j+1})v^{k+1}_{i,j+1} - (D_{i,j} + D_{i+1,j})v^{k+1}_{i,j}\right\} \end{aligned} \tag{6}$$

The use of the finite difference scheme and the particular choice for a space staggered grid has three important advantages for the structure of the implementation:

1. The calculations in each grid point only require neighbouring values, both in space and in time. For instance, the value of v in Equation 1 is approximated in Equation 4 by an average of four neighbours $(v^k_{i-1,j} + v^k_{i-1,j+1} + v^k_{i,j} + v^k_{i,j+1})/4$. The finite difference scheme thus leads to the locality that is required to enable efficient use of computing resources.

2. The finite difference schemes gives an order in which new approximations can be calculated from previous values: first compute all the $u^{k+1}_{i,j}$, then compute all the $v^{k+1}_{i,j}$ and finally all the $h^{k+1}_{i,j}$. This sequence must be repeated until the required time frame has been reached. This sequence has a good stability, which is very important for numerical approximations.

3. The choice of a space staggered grid makes it possible to store the velocities and heights using only three arrays: one for each of the $u^k_{i,j}$, $v^k_{i,j}$ and $h^k_{i,j}$. These arrays may be updated in place, provided the computations are sequenced as discussed above.

These advantages lead to the monolithic and incremental implementations shown in Figure 5. The constant N represents the number of times the *transform* function has to be repeated to compute the heights and velocities during the N-th time frame. The constants X and Y specify the size of the grid. The initial state of the grid is the 3-tuple (u_0, v_0, h_0). Because a space staggered grid is used, the sizes of the three matrices u, v and h are different. This difference is a source of complication for the implementation of both the monolithic and the incremental versions of *transform*.

Given a coordinate pair, the functions *fu*, *fv* and *fh* calculate the initial states of the matrices u_0, v_0 and h_0. The three functions gu', gv' and gh' compute the new value of a point according to the Equations 4–6.

The monolithic version $transform_m$ uses *tabulate2* to generate three new matrices u', v' and h', based on the descriptor *dxy*, where *tabulate2* is merely a two dimensional form of the function *tabulate* as discussed at the beginning of Section 2. The descriptor *dxy* covers exactly the areas of the three matrices for which new values must be calculated during an iteration. As shown in the definitions of u_0 and v_0, the matrix u_0 and therefore also u' must have an extra column to the right and similarly the matrix v' must have an extra row on top. This extra row/column is not delivered by the two applications of *tabulate2* in $level_m$. Instead, an auxiliary function *reshape* is used, which cuts the extra column/row off its first argument and pastes it onto its second argument. This gives the new matrices u' and v' the right shape. The shape of the matrix h' is properly described by *dxy* so there is no need for a call to *reshape* in the definition of h'.

There are other, more efficient solutions possible to give the matrices u' and v' the right shape. We could have hidden the special treatment of the extra column/row in the functions *gu* and *gv*. We have chosen to make the reshaping of the results of the *tabulate2* functions explicit at this level because it confirms that the monolithic array operations as they have been used are not quite powerful enough.

```
descr == (int × int) × (int × int)
triple == (matrix of real × matrix of real × matrix of real)

main :: triple
main                = repeat N (u0, v0, h0)

u0, v0, h0 :: matrix of real
u0                  = tabulate2 fu ((0, X + 1), (0, Y))
v0                  = tabulate2 fv ((0, X), (0, Y + 1))
h0                  = tabulate2 fh dxy

dxy :: descr
dxy                 = ((0, X), (0, Y))

repeat :: int → triple → triple
repeat 0 t          = t
repeat n t          = repeat (n − 1) (transform t)

transform_i, transform_m :: triple → triple
transform_m (u, v, h) = (u', v', h')
                        where
                        u' = reshape u (tabulate2 (gu u v h) dxy)
                        v' = reshape v (tabulate2 (gv u' v h) dxy)
                        h' = tabulate2 (gh u' v' h) dxy

transform_i (u, v, h) = force(u', v', h')
                        where
                        u' = aimap2 (gu' v h) u dxy
                        v' = aimap2 (gv' u' h) v dxy
                        h' = aimap2 (gh' u' v') h dxy

gu, gv, gh :: matrix of real → matrix of real → matrix of real → int → int → real
gu u v h i j        = gu' v h (subscript2 u i j) i j
gv u v h i j        = gv' u h (subscript2 v i j) i j
gh u v h i j        = gh' u v (subscript2 h i j) i j

aimap2 :: (α → int → int → α) → matrix of α → descr → matrix of α
aimap2 f mat ((l_x, u_x), (l_y, u_y))
                    = aimap g mat l_x u_x
                      where
                      g vec i = aimap h vec l_y u_y
                                where
                                h elem j = f elem i j

aimap :: (α → int → α) → array of α → int → int → array of α
aimap f vec k u     = vec, if k > u
                    = aimap f (update vec k x) (k + 1) u, otherwise
                      where
                      x = f (subscript vec k) k
```

Figure 5: Two implementations of `wv`: $transform_m$ follows the monolithic approach and $transform_i$ follows the incremental approach towards array construction.

A function such as *tabulate2* should be capable of applying different functions to different areas of the matrix. This problem is addressed to some extent by the work of the Sisal group [26], and we think that the monolithic array primitives from Haskell are not adequate.

The incremental version $transform_i$ uses *aimap2* to generate three new matrices u', v' and h'. Because of the incremental approach, this time there is no need for something like *reshape*. The reason is that *aimap2* calculates new values for a certain submatrix, and retains the remaining elements as they are. The shape of the matrix argument thus determines the shape of the result, whereas the shape of the descriptor argument determines which matrix elements will be updated. This naturally fits the problem being solved. So for once the incremental approach seems "nicer" than the monolithic approach.

To make the incremental version work properly, an annotation is required to make the three matrices u, v and h single threaded. This annotation is the *force* function in the body of $transform_i$, which evaluates its argument to a full normal form. To see why the matrices are not single threaded without the *force* application, consider the graph shown in Figure 6. This graph represents the evaluation of the third state of the matrix u, which is noted as u''. A box represents a function application, whose name is shown in the box. An arrow represents a pointer to a argument. The pointers to the descriptor argument *dxy* have been omitted to avoid unnecessary clutter. What is shown is thus the essential structure as it is built by the graph reduction implementation. The demand driven evaluation implied by lazy graph reduction starts on the edge marked u''. The demand propagates to the edges marked u' and u. The initial matrix u_0, is thus updated once to form the second state u' and then again to form the third state u''. As the graph shows, there are several pointers to the second state u', which have not been resolved because of pending computations required to calculate v' and h'.

The effect of the *force* annotation is to guarantee that all computations required by the current step are indeed completed before the next step is started. The *force* annotation causes the components of the tuple to be evaluated from left to right, so that first u' is evaluated, then v' and finally h'. Forcing the components in a different order would not solve the problem at all! The *force* annotation makes the matrices single threaded and therefore enables the use of destructive updates. After inserting the *force* annotation, we noticed a significant reduction in the space requirements of the `wv` simulation. For this reason alone, we should thus have been using the *force* annotation at this particular place and not only in the incremental version, but also in the monolithic version of *transform*.

We should like to point out that the FAST compiler is not capable of verifying the safety of destructive updates. It is entirely up to the programmer to work this out. We do not propose to present arbitrary destructive updates as a tool to the casual programmer, but instead use such unsafe facilities as an object of study.

3 Measurements

The three programs `qs`, `fft` and `wv` have been compiled by the FAST compiler [17] and executed on a SUN SPARC 4/690, with 64Mb of memory, 64KB caches, running under SunOS 4.1.2. Various statistics are collected by the runtime system, of which Table 1 presents the most important ones, as shown in the first column.

The three programs have been implemented using incremental arrays (see the columns marked *Incr*) as well as monolithic arrays (see the columns marked *Mono*). All incremental versions have been executed using a destructive implementation of the array update (columns marked *destr*). The `wv` program has also been executed using the naive copying implementation of the array update (column marked *copy*). With a naive copying implementation of the array update, the asymptotic complexity of the other programs is so bad that there is no point in presenting precise measurements. That this is not the case for the `wv` program can be explained as follows. Unlike the other two programs, `wv` uses $N \times N$ matrices, which are implemented as arrays of arrays. When naively updating a point in the matrix, only an array of size N has to be copied, not an entire matrix of size $N \times N$. Because of this implementation choice, even the incremental copying version of this program has an acceptable performance.

Two different array implementations have been used. The first implementation uses strict arrays, which means that every array primitive evaluates all array elements to head normal form when creating or updating an array. This implementation does therefore not allow arrays with undefined elements to be created. The array elements are always boxed. The measurements pertaining to this implementation may be found in the first section of the table (under the heading *Strict arrays*). The second implementation is based on non-strict arrays, for which the measurements may be found in the second section of the

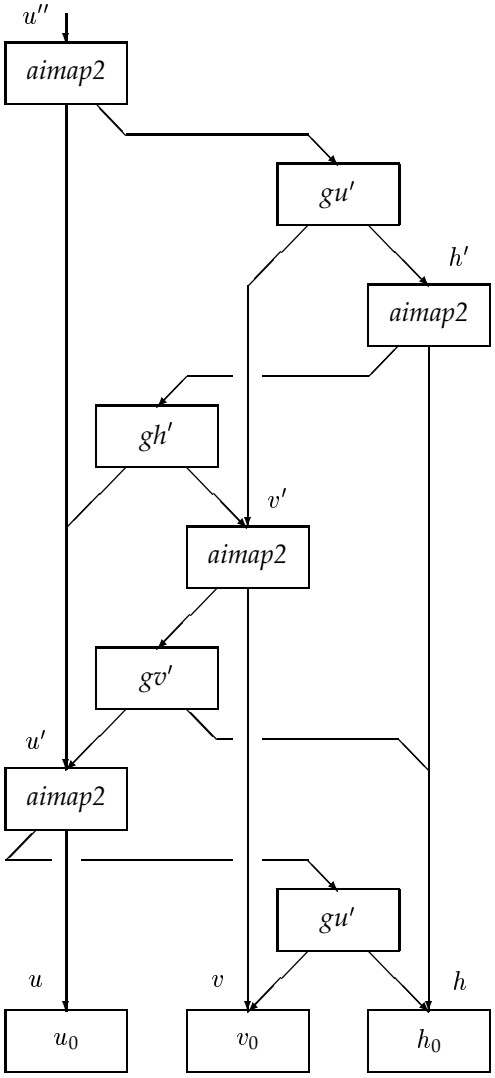

Figure 6: Graph reduction of the function $transform_i$ indicating why single threadedness is not automatic.

table (under the heading *Non-strict arrays*). These latter measurements are shown as a percentage of the corresponding statistic for the strict array implementation. A positive percentage corresponds to an increase, i.e. a worse performance. A negative percentage represents a decrease, i.e. a better performance.

The third section of the table presents a break down of the number of heap cells claimed with respect to the different ways in which the heap cells are used. The break down applies to the implementation of strict arrays only, non-strict arrays are not too different in this respect.

program	`qs`		`fft`		`wv`		
version	Incr destr	Mono	Incr destr	Mono	Incr copy	Incr destr	Mono
input	32767	32767	8192	8192	10	10	10
Strict arrays							
calls	4155K	8009K	2645K	4038K	5262K	5262K	7201K
cells	360K	1657K	977K	2386K	2171K	2046K	2261K
reduce	5429K	9305K	2446K	6730K	8965K	8965K	12010K
seconds	7	15	6	20	24	15	19
Non-strict arrays							
calls			+10%				
cells			+36%	+1%	+7%	+7%	+10%
reduce	+23%		+48%		+13%	+13%	+15%
seconds	+4%	+2%	+88%	+3%	+10%	+14%	+28%
Strict arrays, break down of cell claims							
<SUSP>	36%	47%	3%	37%	17%	18%	18%
<CONS>	9%	40%	1%	11%			
<ARR>				5%	6%		
<NUMB>	27%	6%	61%	32%	12%	13%	15%
DBL			16%	7%	63%	66%	60%

Table 1: Function calls, cell claims, reducer activity and execution time for the different versions of each program. The blank entries in the second and third sections of the table indicate "no change for this entry".

The row marked *input* for `qs` gives the length of the list to be sorted. For `fft`, the input parameter is the number of points transformed and for `wv` the input parameter gives the number of simulated time steps.

The row marked *calls* in the first section presents the total number of function calls, including all primitive and user defined functions. Even a simple addition is counted as a function call, to give an impression of the amount of work involved in each of the computations. The row marked *calls* in the second section gives the increase/decrease in the number of function calls due to the use of non-strict instead of strict arrays. The total number of function calls for the incremental version of the `fft` program is 10% higher when using non-strict arrays. The two other programs are not affected by the choice of (non)-strict array primitives with respect to this statistic.

The row *cells* gives the number of heap cell claims required during the execution. Heap cells are an expensive resource so the larger this number, the larger the expected execution time. The number of cell claims is worse affected by the choice of strict/non-strict arrays than the number of function calls.

The row *reduce* gives a measure of the costs involved in implementing lazy evaluation. The statistic is incremented each time the reducer is called, and also when certain subfunctions of the reducer are called (for example a step during the unwind of the spine). Most versions show a significant increase, up to 48% with respect to this statistic when moving to non-strict arrays.

The row *seconds* gives the execution time of the different versions of the programs. The incremental versions, implemented with destructive updates are always the fastest. There is a relatively large error, of perhaps 50% in time measurements on a complex architecture, such as the SUN SPARC processor with caches [27]. Such large errors arise because small variations in heap size or code size may cause large variations in the effectiveness of the cache and thus in the execution time. Therefore it is safe only to conclude that the destructively updating incremental implementation of `fft` is significantly faster with strict than with non-strict arrays.

In the third section, a breakdown is presented of the number of cell claims as a percentage of the statistic *cell* in the first section. The measurements in the first as well as the third section apply to the implementations based on strict arrays. The following categories of cells appear in the table: <*SUSP*> represents suspended function applications <*CONS*> represents boxed list constructor cells, <*ARR*> represents boxed array containers and <*NUMB*> represents boxed numbers. The category *DBL* represents unboxed double precision numbers. There are other cell categories, such as array descriptor pairs, but they do not occur frequently and have therefore been omitted. This is also the reason that the percentages in each column of the third section do not add up to 100%.

For the `wv` program, the statistic *DBL* accounts for over 66% of the cell claims. This is entirely due to a problem in the implementation that we have used, which is not able to store unboxed double precision numbers directly in a stack frame. Instead, for each unboxed double, a heap cell is allocated, and a pointer to that heap cell is stored in a stack frame. There are implementation techniques to solve this problem but we have not implemented those. The `fft` program also suffers from this deficiency but to a much lesser extent, because at most 16% of its cell claims are unboxed doubles. The list structure being built by the `fft` can be avoided almost entirely, if better methods for compiling array comprehensions are used [4, 16].

The number of cells claimed by even the fastest version of each program is large when compared to the problem size. In particular, it should be possible to run the destructively updating versions of each program using a fixed number of cells, related to the size of the problem being solved. For `qs` and `fft`, the required number of cells is equal to the value of the input parameter. For `wv`, the number of cells does not depend on the input parameter, but only on the size of the three matrices involved. The destructively updating version of `fft` uses $977K/8192 \approx 119$ times more cells than we would like. It should come as no surprise that an implementation in C [28] is about 100 times faster than our best implementation. To solve this problem, it will be necessary to treat unboxed data, such as complex numbers, as first class citizens. This means that an array of complex numbers should be represented not as an array of pointers to cells but exactly as in C, by an array of structs that contain the real and imaginary parts. To achieve such a result requires both powerful boxing analysis as well as new techniques for supporting unboxed values at runtime, and in particular during garbage collection.

4 Conclusions

Using destructive updates in a lazy language is hard, even though the programs that we have studied are relatively small. The programmer requires intimate knowledge of the implementation and even worse, a slight change in the program may make it impossible for the compiler to prove the uniqueness of a pointer to an array. As we have seen in sections 2.2 and 2.4, this has a disastrous effect on the performance of the program and therefore relying on destructive updates is unsatisfactory.

Although the safety of programs using monads or unique types is guaranteed, still the programmer is required to have the same intimate knowledge of the underlying implementation. Otherwise the program cannot be written in such a way that it will be accepted by the static checks of the compiler.

In the programs that we have used, destructive updates were found to be safe except in one case, where an annotation by the programmer proved essential. This annotation (in the `wv` program) turned out to be required also to curtail the space consumption of the program.

The three programs that have been studied admit solutions based on the use of monolithic arrays, which do not have the problem that a small change may influence the performance dramatically. The performance of these monolithic versions is at most a factor of 2 worse than that of the incremental versions. This performance difference may however change as soon as we correct the two shortcomings of our implementation described earlier (the use of heap cells to store unboxed double precision numbers and the generation of redundant intermediate list structure for array comprehensions).

The monolithic programs are neater than the incremental programs, and they have the advantage that even a simple implementation will give a reasonable performance. A sophisticated implementation should give a performance comparable to that of incremental implementations.

The monolithic array operations that we have used are relatively primitive. It would be worthwhile to extend the primitives to support operations over certain index patterns or index ranges. This would improve programs such as `wv` that reshape matrices. Such an extension would improve the appearance of programs using arrays and it would also offer the compiler more scope for optimisations. This route

looks more promising than to try to build several complex analyses into the compiler that must operate in perfect harmony to achieve good results.

Acknowledgements

We thank Marcel Beemster, Bob Hiromoto, David King, Simon Marlow, John O'Donnell and the referee for their comments on a draft version of the paper.

References

[1] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM transactions on programming languages and systems*, 7(2):299–310, Apr 1985.

[2] C. McCrosky, K. Roy, and K. Sailor. Falafel: Arrays in a functional language. In L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao, editors, *1st Arrays, functional languages, and parallel systems (ATABLE)*, pages 107–123, Boston, Massachusetts, Jun 1990. Kluwer Academic Publishers.

[3] P. L. Wadler. A new array operation. In J. F. Fasel and R. M. Keller, editors, *Graph reduction, LNCS 279*, pages 328–333, Santa Fé, New Mexico, Sep 1986. Springer-Verlag, Berlin.

[4] S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Programming language design and implementation*, pages 137–149, White Plains, New York, Jun 1990. ACM SIGPLAN notices, 25(6).

[5] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: A functional perspective. In J. F. Fasel and R. M. Keller, editors, *Graph reduction, LNCS 279*, pages 312–327, Santa Fé, New Mexico, Sep 1986. Springer-Verlag, Berlin.

[6] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM computing surveys*, 21(3):359–411, Sep 1989.

[7] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 636–666, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.

[8] S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. In *20th Principles of programming languages*, pages 71–84, Charleston, South Carolina, Jan 1993. ACM, New York.

[9] S. Smetsers, E. Barendsen, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. Technical report 93-04, Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, 1993.

[10] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. In G. Hains and L. M. R. Mullin, editors, *2nd Arrays, functional languages, and parallel systems (ATABLE)*, pages 52–66. Publication 841, Dept. d'informatique et de recherche opérationelle, Univ. de Montréal, Canada, Jun 1992.

[11] P. L. Wadler. The concatenate vanishes. Internal report, Dept. of Comp. Sci, Univ. of Glasgow, Scotland, Dec 1987.

[12] P. Hudak. A semantic model of reference counting and its abstraction. In *Lisp and functional programming*, pages 351–363, Cambridge, Massachusetts, Aug 1986. ACM, New York.

[13] H. P. Barendregt and M. van Leeuwen. Functional programming and the language Tale. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current trends in concurrency: overviews and tutorials, LNCS 224*, pages 122–208, Noordwijkerhout, The Netherlands, Jun 1985. Springer-Verlag, Berlin.

[14] P. Hudak, S. L. Peyton Jones, and P. L. Wadler (editors). Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):R1–R162, May 1992.

[15] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, Apr 1965.

[16] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *6th Functional programming languages and computer architecture*, pages 223–232, Copenhagen, Denmark, Jun 1993. ACM, New York.

[17] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. *Software—practice and experience*, 24(2):127–173, Feb 1994.

[18] W. R. Stoye. *The implementation of functional programming languages using custom hardware*. PhD thesis, Univ. of Cambridge, England, Dec 1985. Technical report 81.

[19] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 26–38, London, England, Sep 1989. ACM, New York.

[20] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, England, Jun 1991. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England.

[21] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The computer journal*, 32(2):127–141, Apr 1989.

[22] W. G. Vree. The grain size of parallel computations in a functional program. In E. Chiricozzi and A. d'Amico, editors, *Parallel processing and Applications*, pages 363–370, L'Aquila, Italy, Sep 1987. Elsevier Science Publishers.

[23] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1989.

[24] A. W. Heemink. *Storm surge prediction using Kalman filtering*. PhD thesis, Twente technical Univ., Sep 1986.

[25] P. J. van der Houwen. Finite difference methods for solving partial differential equations. Mathematical centre tracts 20, Mathematical Centre, Amsterdam, 1968.

[26] J. T. Feo. Arrays in Sisal. In L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao, editors, *1st Arrays, functional languages, and parallel systems (ATABLE)*, pages 93–106, Boston, Massachusetts, Jun 1990. Kluwer Academic Publishers.

[27] K. Hammond, G. L. Burn, and D. B. Howe. Spiking your caches. In K. Hammond and J. T. O'Donnell, editors, *Functional programming*, pages 58–68, Ayr, Scotland, Jul 1993. Springer-Verlag, Berlin.

[28] W. H. Press, B. P. Flannery, S. A. Tekolsky, and W. T. Vetterling. *Numerical recipes – The art of scientific computing*. Cambridge Univ. Press, Cambridge, England, 1986.

Biographical sketches

Pieter H. Hartel received the Master's degree in Mathematics and Computer Science from the Free University of Amsterdam in 1978 and the Ph.D. degree in Computer Science from the University of Amsterdam in 1989. He has worked at CERN in Geneva, the University of Nijmegen (The Netherlands) and the University of Southampton (England). He is currently a Senior Lecturer at the University of Amsterdam, in the Department of Computer Systems. His research interests are in the theory of programming languages, and the design of compilers, operating systems and architectures for functional languages.

Willem G. Vree studied applied physics in Amsterdam where he obtained the Master's degree in 1973. He worked for 5 years on pattern recognition at CERN in Geneva. Next his interest shifted to distributed

real time systems at the Dutch Waterboard. Later he joined the University of Amsterdam and completed a Ph.D. thesis on parallel reduction machines in 1989. Currently he is head of strategic research in information technology at the Dutch Waterboard. His research interest is in declarative systems and their applications.