J. Functional Programming **7** (1): 103–123, January 1997 © 1997 Cambridge University Press

On the effectiveness of functional language features: NAS benchmark FT

J. HAMMES, S. SUR and W. BÖHM†

Department of Computer Science, Colorado State University, Ft. Collins, CO 80523, USA

Abstract

In this paper we investigate the effectiveness of functional language features when writing scientific codes. Our programs are written in the purely functional subset of Id and executed on a one node Motorola Monsoon machine, and in Haskell and executed on a Sparc 2. In the application we study – the NAS FT benchmark, a three-dimensional heat equation solver – it is necessary to target and select one-dimensional sub-arrays in three-dimensional arrays. Furthermore, it is important to be able to share computation in array definitions. We compare first order and higher order implementations of this benchmark. The higher order version uses functions to select one-dimensional sub-arrays, or slices, from a three-dimensional object, whereas the first order version creates copies to achieve the same result. We compare various representations of a three-dimensional object, and study the effect of strictness in Haskell. We also study the performance of our codes when employing recursive and iterative implementations of the one-dimensional FFT, which forms the kernel of this benchmark. It turns out that these languages still have quite inefficient implementations, with respect to both space and time. For the largest problem we could run (32^3), Haskell is 15 times slower than Fortran and uses three times more space than is absolutely necessary, whereas Id on Monsoon uses nine times more cycles than Fortran on the MIPS R3000, and uses five times more space than is absolutely necessary. This code, and others like it, should inspire compiler writers to improve the performance of functional language implementations.

Capsule Review

Functional programming often seems to be dominated by the rival schools of strict languages implemented through a SECD machine, and lazy languages based on the combinator/graph reduction. As this refreshing and eminently practical paper shows, there is still lots to be learnt from the less well publicised third approach of single assignment languages implemented through dataflow. It is also a sober reminder that lazy and single assignment functional language implementations have a long way to go to match the efficiency of their imperative sisters.

The authors have been investigating the time and space behaviour of Id and Haskell versions of the NAS FT benchmark, a Fast Fourier Transform used to solve a heat equation which involves selecting slices from a three-dimensional array. Their analysis is meticulous, and brings what is, to my mind, a salutary though nonetheless unwelcome conclusion: the more control a programmer has over memory management, the faster and more space efficient functional algorithms become. Thus, the Id versions are improved by explicit space

† This work was supported in part by NSF Grants MIP-9113268 and CDA-9422007, ARPA Contract PO 691964 and a Motorola Monsoon donation from ARPA.

deallocation and the Haskell versions by the use of strict constructs. This is in stark contrast to the promise of functional programming to free users from low level concerns.

The authors have also compared Haskell and Fortran: Fortran won hands down. They are now going to turn their attention to functional languages with eager (pH) and strict evaluation (Sisal). The results will make fascinating reading. Perhaps someone might try the NAS FT benchmark on LISP and Standard ML, and indeed on Miranda. It would be interesting to see how the more mainstream lazy and strict languages compare on this realistic problem, and how strict functional languages measure up against Big Blue's legacy.

1 Introduction

In this paper we study the design of purely functional implementations of the NAS three-dimensional FFT PDE benchmark FT (Bailey *et al.*, 1991), written in Id (Nikhil, 1990) and Haskell (Hudak *et al.*, 1992), in an attempt to assess which declarative language features are of importance when writing efficient, machine independent, parallel scientific codes.

Vectors, matrices and higher dimensional grids play an important role in many scientific codes. Our benchmark is no exception, as it performs three-dimensional Fast Fourier Transforms (FFT) by applying one-dimensional FFTs in all three directions. Hence, when creating or accessing grids, sub-grids need to be selected and targeted. We call this the *slicing problem.* When grid elements are created, substantial sub-computations can be shared by pairs of elements. We call this the *sharing problem.* The representation of the grids influences the expressibility and efficient implementation of slicing and sharing, as well as their resource requirements. We discuss various representations and their effects on time and space efficiency in Id and Haskell.

Two approaches to the slice selection problem are investigated: the creation of a *selection function* that is passed to a higher order function; and the creation of a *copy* of the sub-grid that is passed to a first order function. We also study two approaches to the slice targeting problem: collection of the vectors into a matrix-of-vector structure; and concatenation of sub-grids into a higher dimensional grid. Arrays can be specified using *array comprehensions* and by *association lists*; Haskell has only the latter. We compare the two approaches with regard to their suitability for sub-computation sharing and for slice targeting. We also compare the performance of our codes with recursive and iterative implementations of the one-dimensional FFT, which forms the kernel of the code. Finally, we look at array strictness and its effect of space efficiency in Haskell.

For Id we report the time and space performance of the codes run on a one node Monsoon machine. Monsoon was chosen because it is part of the Id/Monsoon project of MIT and Motorola, and other performance studies for Id programs on this machine have already been performed. An overview of the Id implementation and performance on Monsoon is given in Hicks *et al.* (1993). We measure time performance in the number of machine cycles spent, and we measure our programs' space usage by determining the maximal amount of heap space allocated. We show that the Id heap manager's behaviour is deleterious to the time performance of the recursive codes.

We show that the association list representation, though expressible in Id, does not suit the language implementation well, as most compiler effort has gone in optimizing arrays and loops, and as association lists are much more difficult to explicitly deallocate than arrays. We show this by measuring the performance of an iterative first order version that employs association lists as intermediate structures to solve the sharing problem in the one-dimensional FFT.

For Haskell we use the Glasgow GHC 0.26 compiler and the Chalmers compiler 0.999.7, both run on a SPARC 20/50. We measure time performances using the C shell's `/bin/time` averaged over five runs; space performance is measured using the heap profiling options in both compilers. We use the Chalmers compiler to measure the effects of array strictness by adding annotations to the programs.

The rest of this paper is organized as follows. Section 2 specifies the NAS FT problem. Section 3 discusses programming issues. Sections 4 and 5 detail the various Id and Haskell implementations respectively, and analyse their performances. Section 6 provides conclusions and future research.

2 Problem specification

NAS benchmark *FT* solves the following three-dimensional heat equation:

$$\frac{\delta u(x,t)}{\delta t} = \alpha \nabla^2 u(x,t),$$

where x is a position in three-dimensional space and α a constant describing conductivity. When a Fourier transform is applied to each side, this equation becomes:

$$\frac{\delta v(z,t)}{\delta t} = -4\alpha\pi^2 |z|^2 v(z,t),$$

where $v(z,t)$ is the Fourier transform of $u(x,t)$. This equation has the solution:

$$v(z,t) = e^{-4\alpha\pi^2|z|^2 t} v(z,0).$$

The discrete version of the above problem is solved using FFTs. First a three-dimensional FFT is performed on the original array $u(x,0)$, then the results are multiplied by certain exponentials and lastly an inverse three-dimensional FFT is performed.

In the FT benchmark, the complex array U is initialized using a pseudo-random number generator. Setting V equal to the three-dimensional FFT of U, $\alpha = 10^{-6}$ and $t = 1$, the intermediate value W is computed:

$$W_{j,k,l} = e^{-4\alpha\pi^2(\bar{j}^2+\bar{k}^2+\bar{l}^2)t} V_{j,k,l}$$

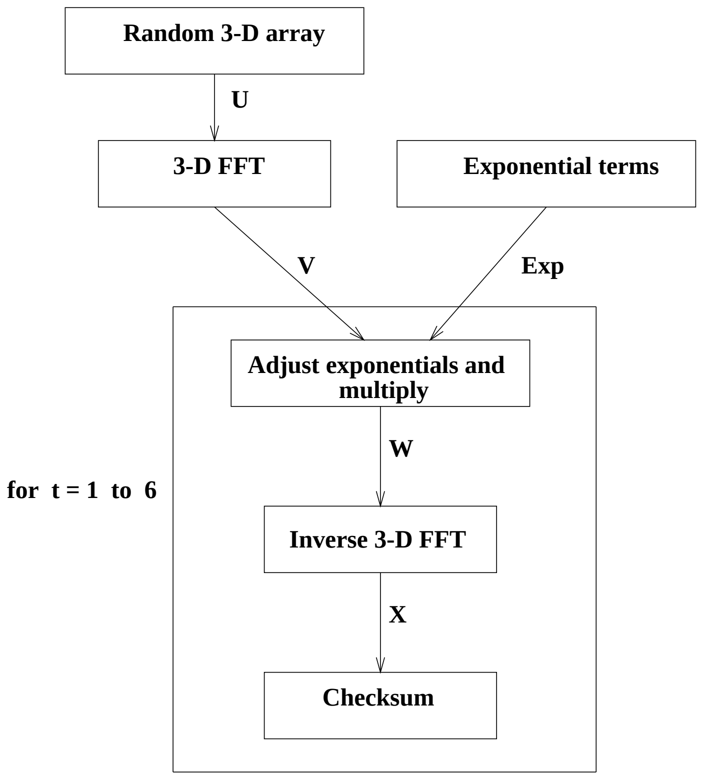

Fig. 1. Flow diagram of the FT benchmark.

where $\bar{j}$ is defined as

$$\bar{j} = j \;\; for \;\; 0 \le j < n_1/2$$

$$\bar{j} = j - n_1 \;\; for \;\; n_1/2 \le j < n_1.$$

The indices $\bar{k}$ and $\bar{l}$ are similarly defined with n_2 and n_3. X, the three-dimensional inverse FFT of W, is then computed. Finally, a checksum is computed and produced as output. The computation of W, X and the checksum, is repeated for values t from one to six. V is computed once. The array of exponential terms for $t > 1$ can be obtained as the t-th power of the array for $t = 1$. Figure 1 shows the flow diagram of the NAS FT benchmark program involving forward and inverse three-dimensional FFTs. A three-dimensional FFT takes a complex array of size $n_1 \times n_2 \times n_3$ and performs $n_2 \times n_3$ n_1-point one-dimensional FFTs in the n_1 direction, followed by $n_3 \times n_1$ n_2-point one-dimensional FFTs in the n_2 direction, followed by $n_1 \times n_2$ n_3 point one-dimensional FFTs in the n_3 direction. Figure 2 sketches the steps performed and the orientation of the intermediate one-dimensional FFTs.

$2n_1n_2n_3$ pseudo-random floating point values are created as specified in the FT benchmark and used to fill the complex array U.

Any algorithm for the individual one-dimensional complex FFT is allowed. We compare a recursive version, performing shuffles and recombinations of two half sized arrays, to an iterative version, which reorders the array using bit-reversal of

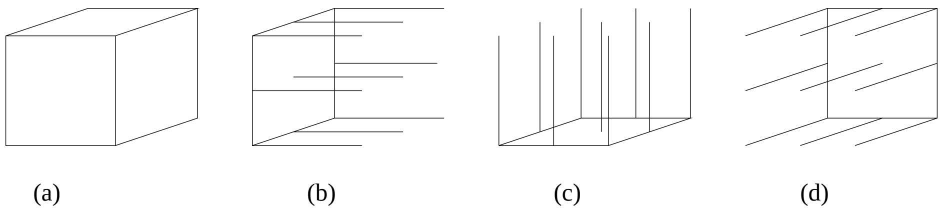

Fig. 2. Steps in performing a three-dimensional FFT.

the index and performs butterfly group recombinations on one array (van Loan, 1992). Bit-reversal is implemented using a table lookup as it is a relatively expensive operation which needs to be done many times for the same size.

A butterfly stage takes two input array elements, which are a distance k away from each other and produces two result array elements, which are targeted at the same positions. Order $log(n)$ butterfly stages are performed where k takes growing powers of two values until $k = n/2$. In our codes the smallest value of k is four, and recombinations for $k = 1$ and $k = 2$ are done in an optimized way. This cuts out the bottom two levels of the FFT call tree, making the program more space and time efficient. The bottom case of four involves multiplications with simple roots of unity ± 1 and $\pm i$. These multiplications can be eliminated, leading to better code than inlining the general case function would achieve, as this function would access the roots of unity from an array and would not eliminate the simple multiplications. Moreover, the bottom level of $k = 1$ merely returns the input array and not going to this level avoids a problem in deallocating intermediate arrays in Id, as a one-dimensional FFT function now always creates a new result array (Böhm and Hiromoto, 1993).

3 Programming issues

Arrays in a functional language can be created using monolithic array constructors called *array comprehensions*, similar in spirit to list comprehensions. While Id has explicit syntax for array comprehensions as distinguished from list comprehensions, Haskell has only list comprehensions, and these are used to specify arrays through association lists. (Though the semantics of Haskell defines arrays through association lists, some programmers may view the composition of the *array* function with a list generator as if it is array comprehension syntax. It is also possible that compiler optimization may elide the list, either as an effect of deforestation or through explicit detection of this common array-building technique.)

An association in Haskell is an (index, value) pair, denoted $(i_1, ..., i_n) := v$, where the index is a tuple of values of the *Ix* type class. Lists of associations are often created using list comprehensions, such as:

```
[(i,j) := f i j | i <- [1..n1], j <- [1..n2]]
```

In the butterfly recombination part of the one-dimensional FFT there are expressions that should be shared among array element computations. Sharing is not directly expressible in the list comprehension syntax, but its effect can be achieved by building a list of sublists, followed by concatenation into one list. Each sublist will contain elements that share a computation. Here is a simple example:

```
concat [ let v = <expr1>
             m = <expr2>
         in [i := m+v, i+1 := m-v] | i <- [1,3..n1]]
```

Arrays in Haskell are defined by two boundary tuples and an association list, where the indices of the association list must be within the boundaries. An index may be undefined or multiply defined in the association list, but an attempt to read such an element from the array will cause a run-time error.

While the list of associations, and the small sublists in the case of sharing, appear to be inefficient, deforestation (Gill *et al.*, 1993) may often eliminate the lists entirely and put the generated elements directly into the array. If this behavior is achieved, then neither expressibility nor efficiency requires a separate syntax for array comprehensions.

Id has an explicit array comprehension syntax that allows an array to be defined in a number of regions of the form | *[target] = expression* || *generator*, where the value of each element of the region is defined by the *expression* and placed in the position specified by the *target*. Both target and expression can be parameterized by the loop variables in the *generator*. The following is an example Id array comprehension:

```
{2D_array((1,n), (1,n)) of
   | [i,j] = i + j || i <- 1 to n-1 ; j <- (i+1) to n
   | [i,j] = i + j || i <- 2 to   n ; j <- 1 to (i-1)
   | [i,i] = 1     || i <- 1 to   n }
```

Id's array comprehensions have no way of expressing shared computations among array elements. However, Id also has list comprehensions that, though different in syntax, are essentially equivalent to those in Haskell, so it is possible in Id to use list comprehensions and association lists in the same way as they are used in Haskell. The Id counterpart of the Haskell *array* function, which builds the array from the association list, is easy to construct. Thus, from the point of view of language expressibility, both Haskell and Id are capable of expressing the sharing of computations among statically known, fixed numbers of array elements via list comprehension and concatenation.

In the three-dimensional FFT we need to go one step further: we need to *select* as well as *target* one-dimensional sub-arrays, or *slices*, of the complete array. There are two approaches to selecting slices. The first approach dynamically creates a *selection function* and passes this function as a parameter to a higher order version of the one-dimensional *fft* function. When provided with an index parameter, the selection function yields an array element from the slice in the three-dimensional object. In a sense, the selection function plays the role of a copy of a one-dimensional slice of the three-dimensional array without the need for an actual copy. The second approach simply creates a copy of the slice and passes this as a parameter to a first order version of the one-dimensional *fft* function.

There are also two approaches to targeting a one-dimensional slice of a three-dimensional object. In the first approach, the representation of the three-dimensional object is a matrix of vectors of complex numbers. This representation allows a vector to be defined as an array element. The second approach uses the association list representation, concatenating association lists together and building the array from the associations.

4 Id implementations and performance

The Id versions that represent intermediate three-dimensional objects by matrices of vectors and select a slice by copying it have the following top level structure:

```
v1 = { matrix ((1,t2),(1,t3)) of
     | [j,k] = fft (get_x j k x)  ro1 fwd || j <- 1 to t2; k <- 1 to t3};
v2 = { matrix ((1, t3),(1,t1)) of
     | [k,i] = fft (get_y k i v1) ro2 fwd || k <- 1 to t3; i <- 1 to t1};
v3 = { matrix ((1, t1),(1,t2)) of
     | [i,j] = fft (get_z i j v2) ro3 fwd || i <- 1 to t1; j <- 1 to t2};
```

where x is a three-dimensional array, $v1$, $v2$ and $v3$ are matrices of vectors, ro_i is a table of roots of unity of the appropriate size, *fwd* a boolean indicating forward or reverse FFT, and the three functions *get_x, get_y* and *get_z* are specialized to deal with the appropriate representation and orientation of the three-dimensional object (see Figure 2), and copy a one-dimensional slice out of it in the proper direction.

In the non copying higher order version a selection function is created and passed to the one-dimensional *fft* function. The keyword *fun* is the Id equivalent of *lambda* in a dynamically defined function.

```
v1 = { matrix ((1,t2),(1,t3)) of
     | [j,k] = fft {fun i =  x[i,j,k]}  t1 ro1 fwd
     || j <- 1 to t2; k <- 1 to t3};
v2 = { matrix ((1,t3),(1,t1)) of
     | [k,i] = fft {fun j = v1[j,k][i]} t2 ro2 fwd
     || k <- 1 to t3; i <- 1 to t1};
v3 = { matrix ((1,t1),(1,t2)) of
     | [i,j] = fft {fun k = v2[k,i][j]} t3 ro3 fwd
     || i <- 1 to t1; j <- 1 to t2};
```

The *fft* function checks whether the input array size is four (the bottom case), and if so, directly performs the transformation. Otherwise it divides the input array into two half-sized arrays containing the odd and even elements of the input array using a shuffle function (van Loan, 1992). After doing a recursive application of *fft* to these two arrays, it combines the results of the recursive function applications into a result array of a size equal to the size of the input array. The higher order version of *fft* gets a selection function rather than an array parameter. Consequently, the shuffle function creates new selection functions instead of arrays.

```
typeof fft = (I -> F) -> I -> Vector F -> I -> Vector F;
def fft fv size RofU fwd =
 if (size == 4) then ... bottom case ...     else
  { (left_fv, right_fv)   =   shuffle fv size;
```

```
      fft_left    =   fft left_fv   (div size 2)  RofU fwd;
      fft_right   =   fft right_fv  (div size 2)  RofU fwd;
       in  combine fft_left fft_right
    };

  typeof shuffle = (I -> F) -> I -> ((I -> F), (I -> F));
  def shuffle fv size =
    { fv_L = {fun i =  fv ((i*2)-1)};
      fv_R = {fun i =  fv (i*2)}
       in (fv_L, fv_R)
    };
```

When the *combine* function needs the array values to compute the recombination, an index value is provided to the selection function, which triggers the application of $O(log(n))$ selection functions to access the appropriate element of the original three-dimensional array. No copying of arrays is needed; only selection functions have been created. Each time a selection function is called, a *closure* is invoked. For each one-dimensional higher order *fft* invocation, there will be $O(n)$ closures and $O(n \times log(n))$ closure invocations. In contrast, for each first order *fft* invocation $O(n)$ intermediate shuffle arrays will be created occupying $O(n \times log(n))$ space.

The association list implementation of the copying iterative version is a direct translation of its Haskell counterpart, using association lists as discussed in section 3. It demonstrates that the sharing expressed in the Haskell programs is also expressible in Id.

4.1 Storage use

The maximal amount of heap memory available on a one node Monsoon machine is four megawords, where a word is an eight byte entity. From Figure 1 it can be seen that in a functional implementation of FT four three-dimensional objects with complex (two word) elements (V, Exp, W, and X) are required simultaneously. Hence, the minimal space requirements for the 8^3, 16^3, 32^3, and 64^3 problems are 32 K bytes, 256 Kbytes, 2048 Kbytes, and 16 Mbytes, respectively. The NAS benchmark size is 64^3. Table 1 provides the heap space usage of our original, non-deallocating codes in K bytes. An 'xxx' indicates that the program ran out of memory. All versions use excessive amounts of space (from 25 to 110 times more than absolutely necessary). For all codes that use the matrix-of-vector representation for intermediate results, the largest problem size we could run on a one node Monsoon machine was 16^3. The association list version performs more than three times worse than its array comprehension counterpart.

We have coded a non-functional Id version of the NAS benchmarks (Sur and Böhm, 1994), which solves the 64^3 sized problem on a one node Monsoon machine. Note that the data structures in these problems are 64 times larger than in a 16^3 sized problem. This code relies on side-effecting in I-structures, and updating in place in M-structures (Barth, *et al.*, 1991).

A main cause of space inefficiency in the current Id implementation is the lack of automatic deallocation and garbage collection. Deallocation must be programmed

Table 1. *Heap space (in Kbytes) used in the Id versions without deallocation*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter	Cop-Iter (Asl)
8^3	1136	1056	832	824	3501
16^3	12120	9200	7080	6888	xxx
32^3	xxx	xxx	xxx	xxx	xxx

explicitly using the *@release* directive. In order to synchronize production, consumption, and release of closures and data structures, barriers (———) are needed. In a block, all expressions preceding a barrier are completely evaluated before any expression following the barrier starts evaluating. A *@release X* directive in a block without an explicit barrier has the effect of releasing X after the whole block has been evaluated. We have implemented explicit deallocation for data structures as well as closures in our matrix of vector codes. We refrained from explicit deallocation in the association list code, as it was only created to show that Id has the same expressive power as Haskell with respect to the sharing problem, but clearly does not compete in terms of efficiency with the matrix of vector codes. With explicit deallocation, the *fft* function from the previous section becomes:

```
typeof fft = (I -> F) -> I -> Vector F -> I -> Vector F;
def fft fv size RofU fwd =
 if (size == 4) then ... bottom case ...     else
  { (left_fv, right_fv)    =    shuffle fv size;
    fft_left     =    fft left_fv    (div size 2)  RofU fwd;
    fft_right    =    fft right_fv   (div size 2)  RofU fwd;
    @release left_fv;   @release right_fv;
    ---
    res = combine fft_left fft_right
    @release fft_left;  @release fft_right
    in res};
  };
```

The heap structures, in this case both closures (*left_fv* and *right_fv*) and arrays (*fft_left* and *fft_right*), are released as soon as possible.

Since any heap structure needs a name if it is to be released, Id programs tend to be written in an incremental style rather than a compositional style. When dynamic functions are created through partial application, each argument uses heap space, and if that space is to be released, each application of an argument must be separately named. Furthermore, sometimes the arguments are not evident in the source code but are introduced by the compiler through lambda lifting. A simple example shows how this can occur and how it can lead to space leaks.

```
typeof test = I -> I -> I;
def test a b =  { @release g;
                 g = {fun i = (a-i)*b}
                 in h g (a+b)};
```

Table 2. *Heap space (in Kbytes) used in the Id versions with deallocation*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
8^3	160	168	176	176
16^3	1320	1248	1320	1296
32^3	22048	9832	22184	10008

The *test* function above allocates two heap closure objects for *g*, one for the partial application of *a* and one for the partial application of *b*, but it releases only the second one. Since the first one is unnamed, it cannot be released. The programmer, in the source program, can release both heap objects through explicit lambda lifting and naming of each application, as follows:

```
typeof test = I -> I -> I;
def test a b =  { @release g'; @release g;
                f = {fun a b i = (a-i)*b};
                g' = f a; g = g' b
                in h g (a+b)};
```

The selection functions exhibit this kind of behavior. For example, in

```
{fun i = x[i,j,k]}
```

the variables *x*, *j*, and *k* each use heap space; the three applications must be explicitly named and eventually released.

Table 2 shows the space requirements of our codes with explicit deallocation. The space use is considerably reduced, and is now between five and ten times the absolute minimum. To reduce the memory requirements further, the Id compiler would need to perform Build and Update in Place optimizations, as is done in the Sisal compiler (Cann, 1992).

For problem size 32^3 we begin to see that the iterative codes are more space efficient than the recursive codes. This is because no intermediate shuffle structures are required, and fewer intermediate butterfly structures (all of the same size) are created.

Work on compiler directed storage management for Id, based on abstract interpretation, is described in Hicks (1993). This approach has the drawback that the compiler cannot always determine whether an object passed into (or returned from) a recursive function has been created anew or is old. If it is old, the object cannot safely be deallocated. This occurs for example in the bottom level of an *fft* function whose base case is size one. In the base case it returns the old object; otherwise it recursively creates a new object, meaning it is safe to release the old object. This problem is discussed in more detail in Böhm and Hiromoto (1993).

Table 3. *Megacycles for COP-ITER, f77 on MIPS R3000 versus Id on Monsoon*

Problem Size	f77 on MIPS R3000	Id on one PE Monsoon
8^3	3.3	26
16^3	25.2	192
32^3	174.2	1583

4.2 Time analysis

The Monsoon machine can be made to count cycles spent in certain parts of the program during execution. It can also report the number of cycles spent in various classes of instructions. Details of the Monsoon and its monitoring system are described in Hicks *et al.* (1993). Here we simply report the total number of Monsoon Megacycles spent in executing a program. The Monsoon is a ten Megahertz machine, so dividing the reported numbers by ten gives the number of seconds spent.

Table 3 compares the number of megacycles spent in the Id copy/iteration version with deallocation running on a one processor Monsoon Machine to the Fortran 77 version of the code that comes with the NAS distribution, running on a DEC Station 5000, which contains a MIPS R3000 processor. This baseline performance comparison of MIPS R3000 and Monsoon is discussed in detail in Hicks *et al.* (1993). The MIPS R3000 requires about nine times fewer cycles than the Monsoon. The Id codes are not optimal because efficiency enhancing non-functional features such as I-structures and M-structures were not used.

In Fortran 77 a three-dimensional object is allocated in a contiguous area in memory, which allows one-, two- or three-dimensional indexing. A three-dimensional FFT is performed in three stages, each stage performing $n_1 \times n_2$ one-dimensional FFTs, just as in the functional code. Between these stages, transpositions are performed by actual copying, which is not done in the functional code. (We assume that the extra copying in Fortran is done for cache performance reasons.) Each one-dimensional FFT is then performed by copying a slice out of the three-dimensional object, performing a one-dimensional FFT on the copy, and copying the result back into the three-dimensional object. A different one-dimensional FFT algorithm (Stockham) is used.

See Tables 4 and 5 for the time performance of our codes. These tables show that deallocation for the iterative codes does not seriously affect time performance. The deallocating recursive codes, however, show an excessive increase in time requirements for the 32^3 problem, and we believe this to be caused by the heap manager. To see the overall effect of the manager, we ran a simple experiment with three simple functions, *up*, *down* and *level*, which allocate space in increasing, decreasing, and unchanging block sizes. For a given value of n, all three functions allocate the same total volume of space.

Table 4. *Megacycles spent in the Id versions without deallocation*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter	Cop-Iter (Asl)
8^3	19	25	23	54	78
16^3	153	199	201	425	xxx
32^3	xxx	xxx	xxx	xxx	xxx

Table 5. *Megacycles spent in the Id versions with deallocation*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
8^3	23	26	25	54
16^3	251	192	232	414
32^3	98769	1583	75895	3300

```
def up n = { s = 0 in
  {for i <- 1 to n do next s = s + f i; finally s }};

def down n = { s = 0 in
  {for i <- n downto 1 do next s = s + f i; finally s }};

def level n = { s = 0 in
  {for i <- 1 to (div n 2) do  next s = s + f n; finally s }};

def f i = { @release a; a = {array (1,i) of | [1] = 1 }; in a[1] };
```

Figure 3 displays the megacycles for these functions for n from 100 to 4800, and shows that there is a large discrepancy between manipulating objects of the same or decreasing size versus objects of increasing size. This behaviour explains the large time performance difference between the iterative and recursive codes. For each one-dimensional *fft* the iterative codes perform $O(log(n))$ cheap allocates of equal sized objects, whereas the recursive codes perform $O(n)$ cheap allocates of objects of decreasing size, and $O(n)$ expensive allocates of objects of increasing size.

5 Haskell implementations and performance

In this section, the following definitions are assumed:

```
type Element = Complex Double
type Vector = Array Int Element
type Threedim = Array (Int, Int, Int) Element
type Matrix_of_Vectors = Array (Int, Int) Vector
type Sel_func = Int -> Element
```

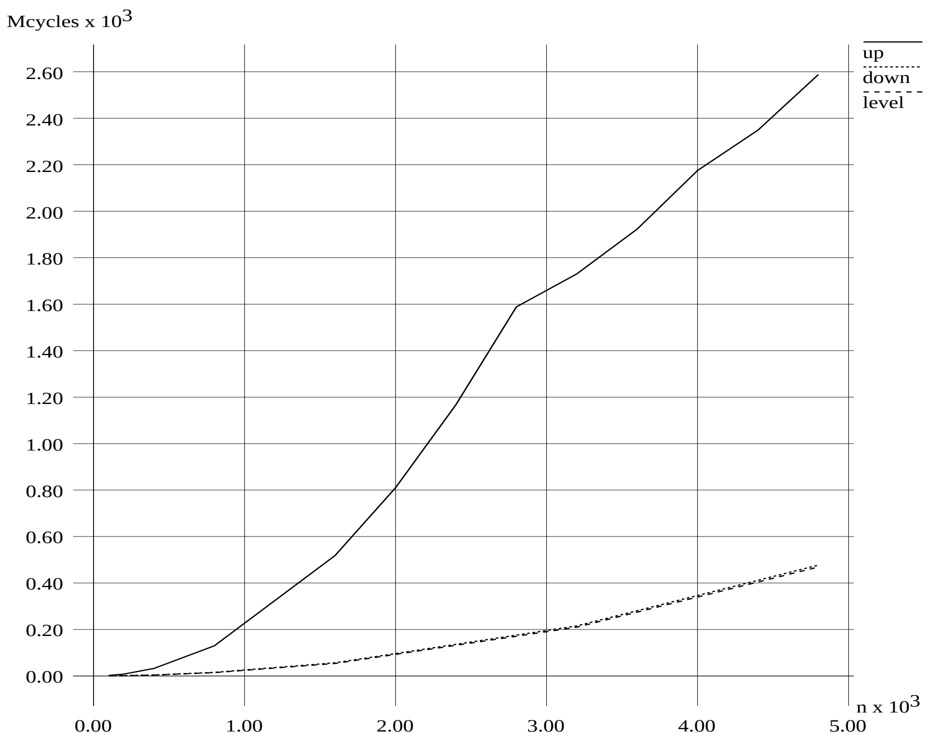

Fig. 3. Megacycles for *up, down* and *level* for varying *n.*

5.1 Three-dimensional arrays as intermediate structures

In the Haskell codes, the most straightforward approach uses three-dimensional arrays as the intermediate structures between the FFTs performed in each dimension. The structure of the three-dimensional FFT function, using the copy technique for slicing, looks like this:

```
cffts_i :: Three_dim -> Three_dim
cffts_i x = array ((1,1),(n2,n3)) [(j,k) :=
    let
        vect = array (1,n1) [i := x!(i,j,k) | i <- [1..n1]]
        g (i := v) = (i,j,k) := v
    in map g (assocs (fft vect rofu_i fwd))
    | j <- [1..n2], k <- [1..n3]]

cffts_j :: Three_dim -> Three_dim
cffts_j x = array ((1,1),(n1,n3)) [(i,k) :=
    let
        vect = array (1,n2) [j := x!(j,k)!i | j <- [1..n2]]
        g (j := v) = (i,j,k) := v
    in map g (assocs (fft vect rofu_j fwd))
    | i <- [1..n1], k <- [1..n3]]
```

```
cffts_k :: Three_dim -> Three_dim
cffts_k x = array ((1,1),(n1,n2)) [(i,j) :=
    let
        vect = array (1,n3) [k := x!(i,k)!j | k <- [1..n3]]
        g (k := v) = (i,j,k) := v
    in map g (assocs (fft vect rofu_k fwd))
    | i <- [1..n1], j <- [1..n2]]

cfft3d :: Three_dim -> Three_dim
cfft3d = cffts_k . cffts_j . cffts_i
```

The *fft* function produces an array, which is transformed into an association list by *assoc*. The *rofu* arrays are the precomputed roots of unity, and *fwd* is a forward/reverse FFT selector. The *vect* arrays are the slices formed by copying. The *cfft3d* function is formed as a composition of the three functions. Each array returned by the one-dimensional *fft* is decomposed to its association list form. Then a one-dimensional-to-three-dimensional function *g* is applied to its elements, and all the lists are concatenated to form a true three-dimensional array. Semantically this creates intermediate lists that will consume space and time. However, deforestation in a compiler may be able to eliminate the intermediate lists and copy the elements directly from the vectors to the three-dimensional array.

Since the vectors from the one-dimensional *fft*s are simply broken down to association lists, it might seem sensible to eliminate the array itself and have the one-dimensional *fft* return an association list instead. However, the one-dimensional *fft* uses arrays internally as it iterates or recurses, and the final array is a natural result of that process. While it is possible to eliminate the arrays within the one-dimensional *fft* and instead to use association lists exclusively, the resulting code is more complicated and obscure because list elements must be ordered so as to bring elements together for use at the right times. This order is implicit and is not evident when looking at the code, making it difficult for a reader to understand how the code works. The advantage of arrays over lists in FFT codes arises out of the ability to use elements in an order that is different from that in which they were produced.

5.2 *Matrix-of-vector intermediate structures*

It is possible to eliminate the three-dimensional intermediate arrays by simply gathering, in a matrix, the vectors returned by the one-dimensional *fft*. The consumer of that matrix of vectors must then be modified to access that structure. This results in the following code:

```
fft :: Vector -> Array Int Element -> Int -> Vector

cffts_i :: Threedim -> Matrix_of_Vectors
cffts_i x = array ((1,1),(n2,n3)) [(j,k) :=
    let   vect = array (1,n1) [i := x!(i,j,k) | i <- [1..n1]]   in
    fft vect rofu_i fwd | j <- [1..n2], k <- [1..n3]]
```

```
cffts_j :: Matrix_of_Vectors -> Matrix_of_Vectors
cffts_j x = array ((1,1),(n1,n3)) [(i,k) :=
    let   vect = array (1,n2) [j := x!(j,k)!i | j <- [1..n2]]   in
    fft vect rofu_j fwd | i <- [1..n1], k <- [1..n3]]

cffts_k :: Matrix_of_Vectors -> Matrix_of_Vectors
cffts_k x = array ((1,1),(n1,n2)) [(i,j) :=
    let   vect = array (1,n3) [k := x!(i,k)!j | k <- [1..n3]]   in
    fft vect rofu_k fwd | i <- [1..n1], j <- [1..n2]]

cfft3d :: Threedim -> Matrix_of_Vectors
cfft3d = cffts_k . cffts_j . cffts_i
```

The *cffts_i* function takes a true three-dimensional array, whereas the *cffts_j* and *cffts_k* functions take a matrix-of-vectors. The vectors that are returned by the one-dimensional *fft* are collected into matrices. Though it is not shown here, the actual code takes the final matrix of vectors and builds a true three-dimensional array as the return value of the *cfft3d* function.

The structure of the selection function version is similar:

```
fft :: Sel_func -> Int -> Array Int Element -> Int -> Vector

cffts_i :: Threedim -> Matrix_of_Vectors
cffts_i x = array ((1,1),(n2,n3)) [(j,k) :=
    let   f i = x!(i,j,k)  in
    fft f n1 rofu_i fwd | j <- [1..n2], k <- [1..n3]]

etc...
```

The selection function *f* and the dimension size *n1* replace the copied vector. The dimension size is needed here because the selection function does not carry bounds information, whereas in the copy version the *fft* function could look at the bounds of the copied array to get the size.

Since the recursive one-dimensional *fft* must shuffle its source array into two subarrays, the selection version of the code must have a shuffle function that takes the selection function and returns two new selection functions, one for each of the two subarrays:

```
shuffle :: Sel_func -> (Sel_func, Sel_func)
shuffle f = ((\i -> f ((i*2)-1)), (\i -> f (i*2)))
```

5.3 Space analysis

Both the Glasgow and Chalmers Haskell compilers allow heap profiling, which can produce snapshots of the heap at various times during the program's execution. The Glasgow profiling can be broken into *cost centres* (Sansom and Peyton-Jones,

Table 6. *Peak heap use (in Kbytes) for Haskell versions with three-dimensional intermediate forms*

	Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
Glasgow	8^3	483	441	464	426
	16^3	4908	4380	4516	4013
Chalmers	8^3	804	834	623	691
	16^3	6300	7038	5115	5828

Table 7. *Peak heap use (in Kbytes) for Haskell versions with matrix-of-vector intermediate forms*

	Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
Glasgow	8^3	432	402	463	413
	16^3	4524	3667	4513	3807
Chalmers	8^3	375	404	356	398
	16^3	3330	3490	3214	3496

1994) that are specified by the programmer and allow attention to be focused on specific areas of interest within the program. All the programs were compiled with `-O` optimization.

Table 6 shows the peak heap consumption of the Haskell codes that use three-dimensional intermediate arrays, and Table 7 shows the same information for the matrix-of-vectors codes. These tables also compare the Glasgow and Chalmers compilers. The Chalmers executables favour selection/recursion, and perform worst for copy/iteration, especially with three-dimensional intermediates. In contrast, the Glasgow executables favour selection/iteration, and perform worst for copy/recursion. It is interesting to note that Chalmers is significantly worse than Glasgow for the three-dimensional intermediates, whereas it tends to be better than Glasgow for matrix-of-vector intermediates.

The result of each benchmark is a set of checksums that are computed by summing selected elements of the three-dimensional fft result arrays. This means that lazy evaluation is probably leaving some elements of the arrays unevaluated if they aren't demanded by the checksum, and it also means that strictness analysis will conclude that the array elements are non-strict. However the intent of the checksums in the FT benchmark is to get a compact 'signature' of the ffts, not to prevent some elements

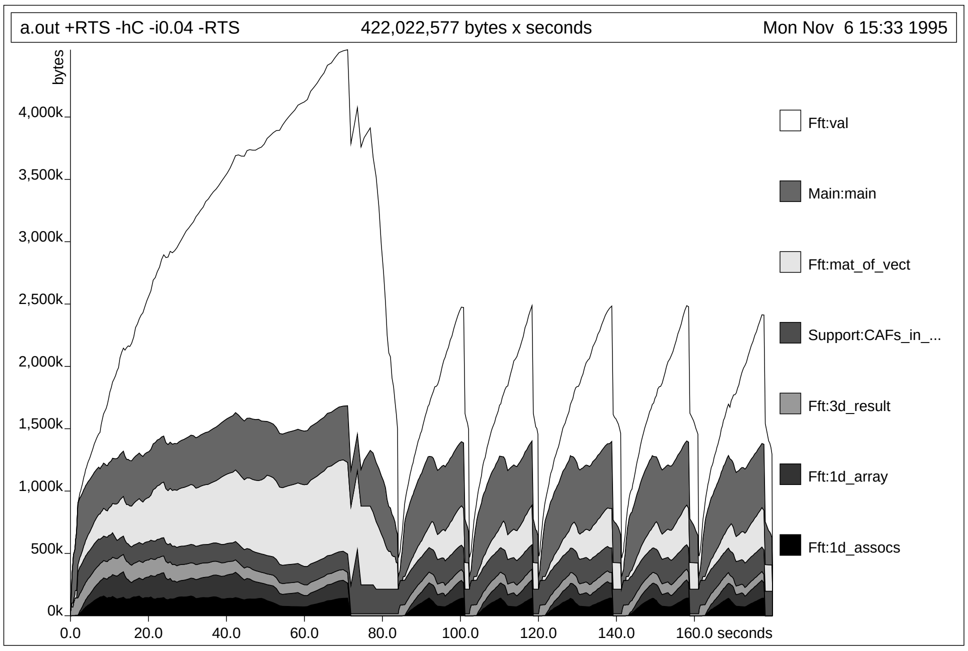

Fig. 4. Heap profile for 16^3 copying, recursive version (Glasgow).

from being computed, so a strict evaluation of all array elements is appropriate and should lead to more efficient heap performance, as will now be shown.

Non-strict arrays require the presence of thunks, which are unevaluated expressions. In Haskell an array is created from an association list that is strict in its indices but not in its values. This allows allocation of space for the array, but requires that the array's elements start out as thunks. When an array element is subsequently demanded, its thunk is evaluated and the value goes into the array in its place. The problem is that thunks take more space than array elements. Figure 4 shows the heap profile for the Glasgow 16^3 copy-recursion code. To help understand the space cost of thunks, the following cost centres were established:

- *mat_of_vect*: the intermediate matrices, but not the one-dimensional vectors they contain
- *1d_in*: the slice copies sent to the one-dimensional *fft*
- *3d_result*: the true three-dimensional result array
- *1d_array*: the arrays returned from the one-dimensional *ffts*
- *1d_assocs*: the association lists for the one-dimensional *ffts*, not including the values in the association lists
- *val*: the values in the one-dimensional association lists
- *main*: everything else

The *1d_assocs* cost centre contains the lists (but not the values) used in the concatenated two-element lists that implement sharing (see section 3). It is not easy to determine exactly where a compiler succeeds in deforesting, but the existence of this band indicates that these association lists have not been deforested by this

Table 8. *Peak heap use (in Kbytes) for strict Haskell versions (Chalmers)*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
8^3	94	86	92	89
16^3	755	713	743	698
32^3	5988	6102	6170	6048

compiler. Nevertheless, it is interesting to note that this cost centre represents a small part of the heap requirements, leading to the conclusion that association list deforestation would not significantly improve the heap performance of the programs.

The *val* cost centre, which is the top white band, represents only the array elements returned by the one-dimensional *fft* (including its intermediate arrays that occur during iteration or recursion), and the profile shows that more than half of the peak heap requirement is attributed to these array values. The white part of the first and biggest peak contains the values used in V and in the first X array (see Figure 1). After the first X and its checksum have been computed, the white space consists only of V's contents, which by then have been evaluated. The white area of the remaining five peaks represents the contents, at first as thunks and then as values, of the remaining X arrays. The fact that the heap use drops so dramatically between peaks suggests that the large *val* area in the first peak must be attributed not only to the values in these arrays, but mainly to the space requirements of the thunks, since the trough between peaks contains the evaluated contents of V. (Its values are needed through all six iterations.)

To measure the effects of strict arrays, the Chalmers compiler was used to compile versions of these codes in which the value in every array association is annotated as strict. Table 8 shows that the peak heap requirements are reduced by as much as 80% over the lazy versions. This made it feasible to run the 32^3 problem that in non-strict evaluation simply consumed too much space. There are no clear winners among these strict codes; their heap requirements are very similar.

5.4 Time analysis

Table 9 shows the time performance of the codes that use three-dimensional intermediate structures. Both Glasgow and Chalmers favour the selection/recursion code, and both show the copy versions performing worse than the selection versions. Glasgow outperforms Chalmers in all four versions.

Table 10 shows the time performance for the matrix-of-vector codes. Both compilers again favour selection/recursion, and perform worst for copy/recursion. Performance improvement over the three-dimensional intermediates ranges from about 20 to 40%.

Table 9. *Execution time (seconds) for Haskell versions with three-dimensional intermediate forms*

	Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
Glasgow	8^3	1.68	1.68	1.38	1.42
	16^3	17.6	15.0	13.5	14.7
Chalmers	8^3	2.04	2.26	1.78	2.24
	16^3	24.9	27.2	17.9	23.1

Table 10. *Execution time (seconds) for Haskell versions with matrix-of-vector intermediate forms*

	Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
Glasgow	8^3	1.22	1.26	0.96	1.12
	16^3	13.9	11.7	10.3	11.1
Chalmers	8^3	1.72	1.88	1.46	1.78
	16^3	16.7	16.5	13.1	16.1

Table 11. *Execution time (seconds) for strict Haskell versions (Chalmers)*

Problem Size	Cop-Rec	Cop-Iter	Sel-Rec	Sel-Iter
8^3	2.04	2.18	1.70	2.02
16^3	16.8	17.3	14.5	16.6
32^3	288.7	240.0	207.4	238.8

Table 11 shows the time performance for the strict array codes compiled with the Chalmers compiler. The performance is slightly worse than the lazy codes. Again, selection/recursion is favoured and copy/recursion is the poorest performer.

The Fortran FT code was also run on the same machine, allowing comparison of its performance with that of Haskell. Table 12 compares the best of the Haskell codes (Glasgow-lazy for 8^3 and 16^3, and Chalmers-strict for 32^3) with Fortran.

6 Conclusion

In writing the NAS FT benchmark in Haskell and the purely functional subset of Id, three important lessons were learned.

Table 12. *Comparison of Haskell and Fortran time performance*

Problem Size	best Haskell	Fortran	ratio
8^3	0.96	0.20	4.8
16^3	10.3	1.60	6.4
32^3	207.4	13.43	15.2

The first lesson is that these languages still have quite inefficient implementations, with respect to both space and time. For the largest problem we could run (32^3), Haskell is 15 times slower than Fortran and uses three times more space than is absolutely necessary, whereas Id on Monsoon uses nine times more cycles than Fortran on the MIPS R3000, and uses five times more space than is absolutely necessary. For Haskell, deforestation might improve the space efficiency slightly, but more serious space efficiency was gained by strictifying the array values. In Id, deallocation needs to be explicitly programmed, and the heap management run time system's implementation can give rise to serious time inefficiencies when increasing sized objects are allocated, as happens in the recursive codes.

The next lesson is that, while slice selection is expressible as a selection function, the slice targeting problem is not as elegantly solved. The most natural intermediate structure in these codes, a three-dimensional array of complex numbers, requires explicitly decomposing the one-dimensional array into its associations and mapping them to three-dimensional associations. While deforestation might eliminate the intermediate lists in this process, it will not allow the *fft* function to put its final values directly into the three-dimensional object. An alternate view is to add slicing constructs to the language, which could allow compilers to do produce more efficient slicing code. The idea of slice selection and assignment in languages is not new: languages such as Algol 68 and APL treat slices as first class array objects.

The third lesson is that, as in all programming languages, the choice of algorithm, data representation, and programming style influence the efficiency of the final code, and conversely, the relative efficiency of certain language features influences the programming style. This encourages Id programmers who are concerned with the efficiency of their codes to use explicit I-structures and M-structures (Sur and Böhm, 1994).

In future work we will investigate the expressiveness and efficient implementation of other functional programming languages such as pH, which will provide parallelism and a more eager evaluation mechanism than Haskell, and Sisal, which allows strict evaluation, efficient array manipulation, and parallel implementation across a large spectrum of parallel machines.

7 Obtaining source code and documentation

The source codes are available by anonymous ftp from *schubert.cs.colostate.edu*, directory *pub/FT_functional*. The NAS FT benchmark specification requires that the program itself generate the array data, so the only required inputs are the three dimensions of the three-dimensional object.

Acknowledgements

Arvind and Nikhil pointed out the selection function approach as an alternative to copying. Simon Peyton Jones pointed out that association lists allow the expression of sharing, and Andy Gill explained the behavior of the deforestation algorithm in the Glasgow Haskell compiler.

References

Arvind, Nikhil, R. S. and Pingali, K. K. (1989) I-structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, **11**(4), 589–632.

Bailey, D. et al. (1991) The NAS Parallel Benchmarks. *Report RNR-91-002 revision 2*, NASA Ames Research Center.

Barth, P. S. and Arvind Nikhil, R. S. (1991) M-structures: Extending a parallel, non-strict, functional language with state. *Proc. FPCA*, Cambridge, MA, August.

Böhm, A. P. W. and Hiromoto, R. E. (1993) Dataflow Time and Space Complexity of FFTs. *J. Parallel and Distributed Computing*, **18**.

Bollman, D., Sanmiguel, F. and Seguel, J. (1992) Implementing FFT's in Sisal. *Proc. Second Sisal Users Conference*, December. LLNL Report CONF-9210270.

Cann, D. (1992) Retire Fortran? A Debate Rekindled. *Communications of the ACM*, **35**(8), 81–89.

Feo, J. and Cann, D. (1993) Developing a high-performance FFT algorithm in Sisal for a vector supercomputer. *Proc. Sisal'93*, October. LLNL Report CONF-9310206.

Gill, A., Launchbury, J. and Peyton-Jones, S. (1993) A Short Cut to Deforestation. *Proc. Functional Programming Languages and Computer Architecture*, Copenhagen, June. ACM Press.

Hicks, J., Chiou, D., Ang, B. S. and Arvind Nikhil, R. S. (1993) Performance studies of Id on the Monsoon dataflow system. *J. Parallel and Distributed Computing*, **18**, 273–300.

Hicks, J. (1993) Experiences with Compiler-Directed Storage Reclamation. *Proc. Functional Programming Languages and Computer Architecture*, Copenhagen, June. ACM Press.

Hudak, P. and Fasel, J. (1992) A gentle Introduction to Haskell. *ACM SIGPLAN Notices*, **27**(5).

Hudak, P., Peyton Jones, S. and Wadler, P. (eds). (1992) Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, **27**(5).

Nikhil, R. S. (1990) Id Version 90.0 Reference Manual. *Computational Structures Group Memo 284-1*, MIT.

Sansom, P. and Peyton-Jones, S. (1994) Time and Space Profiling for Non-Strict Higher-Order Functional Languages. *POPL 1995* (submitted).

Sur, S. and Böhm, A. P. W. (1994) Functional, I-structure, and M-structure Implementations of NAS Benchmark FT. *Proc. PACT'94*, Montreal, Canada, August.

van Loan, C. (1992) Computational Frameworks for Fast Fourier Transform. *SIAM Frontiers in Applied Mathematics*.