Implementing the Conjugate Gradient Algorithm in a Functional Language

Pascal R. Serrarens

Computer Science Institute
University of Nijmegen, The Netherlands
e-mail: pascalrs@cs.kun.nl

Abstract. This paper evaluates the elegance and efficiency of functional programming in numerical scientific computing, an interesting area because time and space efficiency are important: many scientific programs work with large data sets and run for a long time. The example we use is the conjugate gradient algorithm, an iterative method to solve systems of linear equations. We investigated various implementations of the algorithm in the functional languages Clean and Haskell and the imperative language C. Good results are obtained when comparing the algorithm written in Clean with the same algorithm in C and Haskell.

The expressive power of functional programming languages seems to equal to that of traditional imperative languages, while many current compilers produce time- and space-efficient code, using various optimisations including the important update-in-place. Two traditionally weak points of functional language implementations are speed and memory usage, which are very important in numerical scientific computing. The "pseudoknot" paper [H^+96] showed that, for a numerical algorithm, only a couple of functional-language compilers could produce code comparable with to that produced by C compilers.

In this paper we investigate writing the conjugate gradient algorithm in the general purpose lazy functional language Clean [PvE93] and compare it against similar implementations in C and Haskell [HPJW92]. In Section 1 we will look at the algorithm itself. We look how the choice of data structure influences the performance in Section 2. We describe some optimisations that yield an efficient implementation in Section 3. In Section 4 we compare our implementation in Clean with implementations in the languages C and Haskell, while Section 5 concludes.

1 The Conjugate Gradient Algorithm

An algorithm often used in scientific computing for solving a system of linear equations is the conjugate gradient algorithm [CSJ95] [EO94] [JDG92] [HS93]. The algorithm itself is an iterative approximation method to solve the linear equation $Ax = b$. A direct method is the gauss elimination. But, only when working with symbolic computation, which provides arbitrary precision integers, can we get an exact solution. Functional languages might be well suited for that. The conjugate gradient algorithm is used often because it is very fast and gives a good approximation of x.

The algorithm can be written down easily in Clean: it is close to what a mathematician would write. One limitation making the code less elegant is that the standard library defines that the overloaded *-symbol must have two arguments of the same type. Matrix-vector and scalar multiplications take the combination matrix-vector and scalar(real)-vector. To overcome this problem, we use the symbols #* and .* for matrix-vector and scalar multiplication respectively.

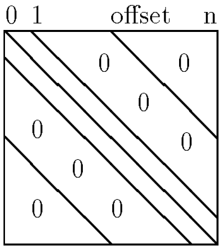
```
CG :: Matrix Vector Vector → Vector
CG a x b
    | delta0 < Epsilon = x
    | otherwise        = CGloop a x g delta0 d
    where
        g      = a #* x - b
        delta0 = InnerProd g g
        d      = ~ g


CGloop :: Matrix Vector Vector Real Vector → Vector
CGloop a x g delta0 d
    | delta1 < Epsilon = x`
    | otherwise        = CGloop a x` g` delta1 d`
    where
        h      = a #* d
        tau    = delta0 / InnerProd d h
        x`     = tau .* d + x
        g`     = tau .* h + g
        delta1 = InnerProd g` g`
        beta   = delta1 / delta0
        d`     = beta .* d - g`
```

Fig. 1. The conjugate gradient algorithm

The algorithm is called with the matrix A, the vector b and a rough approximation of vector x. After some initialisation in function CG, the recursive function CGloop runs until delta1 drops below the value of Epsilon, for which good values are between 10^{-8} and 10^{-12}. The returned x is a good approximation of the real value of x. A more detailed description of the algorithm can be found in [KGGK94].

1.1 The Matrix

The representation of the matrix we have chosen in this paper is not straightforward. The conjugate gradient algorithm is used often to solve large sparse systems of linear equations. This implies two things: the first is that the matrix can be very large: matrices of size $100,000 \times 100,000$ for storing the data are not uncommon. With a direct representation it will take more than 74 gigabytes to store a matrix of that size.



Sparse matrix

The second assumption is that the matrix is sparse, meaning that many matrix elements are zero. In this paper we use banded diagonal matrices with all non-zero elements aligned along five diagonals (see the picture on the left), just storing the diagonals is a good idea. We also assume that the matrix is symmetric, so we only have to store 3 diagonals. Now we need only a little more than 2 megabytes memory to store the matrix above.

The special representation of the matrix has its influence in the implementation of the matrix-vector multiplication. Normally, every value in the resulting vector is the result of an inner product of the vector element and the corresponding row of the matrix. In our situation we need only sum the products of the diagonals with the vector.

1.2 The problem sizes

We have tested the various implementations of the algorithm for up to five problem sizes, which are determined by the array size (see table 1).

Array size	Matrix + Vectors
625×625	35 Kb
2500×2500	140 Kb
10000×10000	560 Kb
40000×40000	2240 Kb
160000×160000	8960 Kb

Table 1. The problem sizes and their data sizes

The first column gives the array size. The second column lists the minimum data size of the matrix and vectors for every problem size. The minimum data size can be calculated easily: we need 3 vectors for the diagonals of the matrix and 4 vectors to store intermediate results during the computation: x, g, d and h. As we use 8-byte reals we need $s * (3 + 4) * 8$ bytes of memory, where s is the size of the array in one dimension.

2 Implementing the vectors and matrices

2.1 Vectors and matrices from lists

Why consider lists when time and space efficiency are so important? We have a number of reasons for that:

- Functional languages are designed with the list as the main linear structure, with list comprehensions, pattern matching and an extensive set of standard functions.
- In our algorithm lists can be used quite easily, because most of the time we simply walk over the structure. Indexing and updating can be avoided.
- We wanted to see the importance of good implementation of arrays in functional languages.

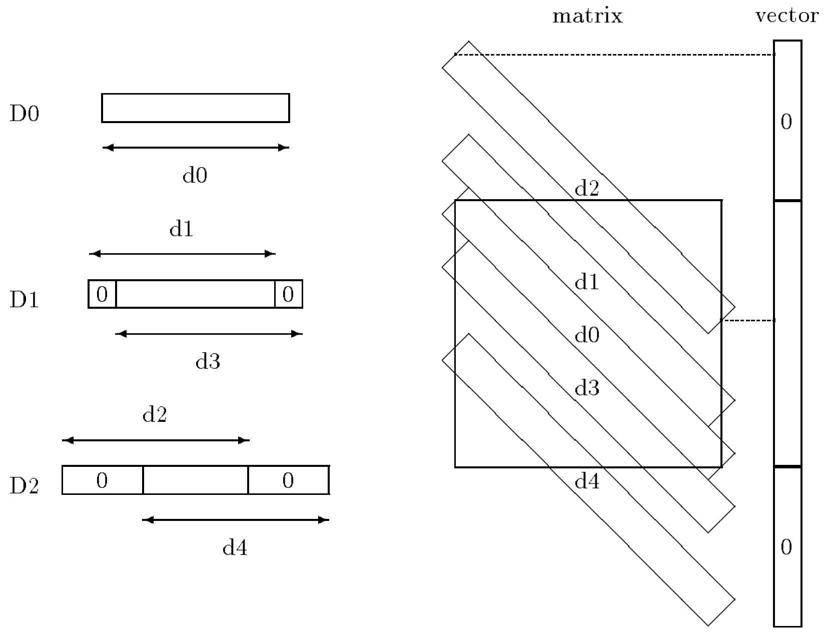
A number of operations are defined on vectors, like addition and inner product. Having list comprehensions, we can express those operations easily: (In Clean, when generators are separated with &, both lists are traversed simultaneously; each generated element is only combined with that at the same position in the other list. In Haskell, you would use zipwith)

```
instance + [ Real ]
where
    (+) v1 v2 = [ e1 + e2 \\ e1 ← v1 & e2 ← v2 ]

InnerProd :: [ Real ] [ Real ] → Real
InnerProd v1 v2 = foldl1 (+) [ e1 * e2 \\ e1 ← v1 & e2 ← v2 ]
```

The matrix-vector multiplication #* is the hardest part of the implementation, because of the representation of the matrix. First we have to extract the real five diagonals. The stored diagonals D0, D1 and D2 are the three upper diagonals of the matrix with respectively 0, 1 and *offset* zeros added to the front and back. From this all five real diagonals of the matrix can be created easily, by taking the correct ranges (see figure 2a).

We also add *offset* zeros to the front and back of the vector. Generating the result vector is now relatively easy: we multiply every diagonal of the matrix with the corresponding range of the vector (see figure 2b). Finally, we sum the results up for the result vector. The code for lists is shown below:

```
(#*) infixl 7 :: [[ Real ]] [ Real ] → [ Real ]
(#*) (offset, D0, D1, D2) v =
    [ d0e * v0e + d1e * v1e + d2e * v2e + d3e * v3e + d4e * v4e \\
```



(a) Creating the five matrix diagonals dx from the stored three diagonals Dx

(b) Multiplying the diagonals with the vector. The dashed lines indicate which part of the vector is taken.

Fig. 2. The Matrix' diagonals

```
    d0e ← D0 &
    d1e ← drop 1 D1 &
    d2e ← drop offset D2 &
    d3e ← take 1 D1 &
    d4e ← take offset D2 &

    v0e ← v &
    v1e ← drop (offset + 1) v' &
    v2e ← drop (offset + offset) v' &
    v3e ← drop (offset - 1) v' &
    v4e ← v' ]
where
  v' = repeatn offset 0.0 ++ v ++ repeatn offset 0.0
```

2.2 Arrays

Clean provides three different kinds of arrays:

Lazy arrays { x } are basically a block of pointers to elements, which may represent closures.

Strict arrays { ! x } are represented the same way as lazy arrays, but now the elements are always evaluated.

Unboxed arrays { # x } are stored in the most efficient way. All elements are stored unboxed [PJL91] in a continuous block of memory.

The overloading mechanism of Clean makes it possible to use the same function names for creating, selecting and updating of the three kinds of arrays. Therefore we can write the program independent of the array representation.

The arrays of Clean are similar to lists, as we have ZF-expressions and pattern matching for arrays. The ZF-expressions are called array comprehensions and differ from list comprehensions in two things: to generate an array curly braces { } are used instead of square brackets [ ] and ←: is used to draw an element from an array, while lists use ←. As an example we give the map function on lazy arrays:

```
map_array :: ( a → b ) { a } → { b }
map_array f a = { f e \\ e ←: a }
```

Pattern matching on arrays can be done with the pattern a=:{[i] = x} , where x is taken from the i-th position of array a.

Besides generating and indexing of arrays, updating array elements is also possible in some cases. When the array type has the uniqueness attribute *, it is safe to perform an update in place. Uniqueness typing [BS93] is able to derive, or enforce, that functions have exclusive access to arguments with the uniqueness attribute. The expression { a & [i] = x } is an update of array a at position i with x.

2.3 Vectors and matrices from arrays

In [H+96] the remark was made that arrays were not well supported in most of the languages. Now that Clean supports arrays, we can see the importance of it. In this section we do not make use of update-in-place, with one exception for the matrix-vector multiplication. Update-in-place will be used in Section 3.2.

For the inner product we use indexing, which is more efficient than a combination of a map and a fold, because now we use only one loop (see also Section 3.1).

```
InnerProd :: { Real } { Real } → Real
InnerProd v1 v2 = InnerProd` (size v1 - 1)
where
    InnerProd` :: Int → Real
    InnerProd` 0 = v1.[0] * v2.[0]
    InnerProd` i  = v1.[i] * v2.[i] + InnerProd` (i - 1)
```

The matrix-vector multiplication with the array implementation is different from the list version, because we use indexing and updates. Going from the middle to the outside diagonals, we first multiply the middle diagonal with the given vector. Of the next two diagonals, the first $n-1$ elements of the upper upper diagonal are multiplied with the vector and added to the result vector. The treatment of the lower diagonal is similar, but uses the last $n-1$ elements. Next, we go to the outermost diagonals, where we take the first and last $n-offset$ elements of the vector argument respectively. This results in the following code:

```
(#*) infixl 7 :: {{ Real }} { Real } → { Real }
(#*) (offset, d0, d1, d2) v =
    Mul1 (n - offset) n d2 ( Mul1 (n - 1) n d1 ( Mul0 v ) )
where
    n = size v - 1

    Mul0 :: Vector → *Vector
    Mul0 v = { de * ve \\ de ←: d0 & ve ←: v }

    Mul1 :: Int Int Diagonal *Vector → *Vector
    Mul1 0 i2 d=:{[0]=de} u=:{[0]=e1, [i2]=e2} =
        { u & [0] = e1 + de * v.[i2],
              [i2] = e2 + de * v.[0] }
    Mul1 i1 i2 d=:{[i1]=de} u=:{[i1]=e1, [i2]=e2} =
        Mul1 (i1 - 1) (i2 - 1) d { u & [i1] = e1 + de * v.[i2],
                                       [i2] = e2 + de * v.[i1] }
```

2.4 Comparing lists and arrays

We were able to reuse the algorithm code presented in Section 1 for all list and array implementations. However, the implementation of simple operations like addition of vectors, must be implemented separately for arrays and lists. Multi-parameter type classes could avoid this [PJ96], but are not implemented at the moment.

For every problem size 10 test-runs were made, of which the best time was taken. All tests ran on a Sun Sparcstation 20 at 50MHz with 64Mb of memory and 1Mb cache running under Solaris 2.4. The compiler for Clean was Clean 1.1 for Solaris. For most implementations we tested for all problem sizes which could run efficiently in 32 Mb or less memory. The times were obtained by the built-in timer of Clean.

Table 2 lists from left to right the problem size and the execution times in seconds for lazy and spine-strict lists and lazy, strict and unboxed arrays. The spine-strict lists are user-defined and were to test the advantage of them over lazy lists. It showed that using spine-strict instead of lazy lists does not give a big speed advantage. The most remarkable thing is that lazy arrays are less efficient than all other implementations, including lists. This is because access to the array elements is much more expensive, as the elements may be unevaluated. Much time is also spent in the garbage collector, due to the high memory requirements.

	Lists		Arrays		
Array size	Lazy	Strict	Lazy	Strict	Unboxed
625^2	1.09	0.70	1.38	0.28	0.18
2500^2	9.06	7.51	15.10	3.23	1.45
10000^2		73.01		31.21	14.57
40000^2				318.45	159.13

Table 2. The execution times in seconds of the various list and array implementations

		Lists		Arrays		
Array size	Real Data	Lazy	Strict	Lazy	Strict	Unboxed
625^2	35 Kb	850 Kb	120 Kb	3790 Kb	80 Kb	60 Kb
2500^2	140 Kb	6170 Kb	480 Kb	29960 Kb	310 Kb	210 Kb
10000^2	560 Kb		1900 Kb		1220 Kb	810 Kb
40000^2	2240 Kb				4900 Kb	3230 Kb

Table 3. The minimum heap sizes for the various list and array implementations

Table 3 shows for every implementation and problem size the minimum heap size: we experimentally determined the size of heap just big enough for the program to run, by gradually reducing the heap size. Next to the problem size is the real data size we deduced in Section 1.1. The memory requirements of the strict and unboxed implementations of the list and the array are reasonable close to the minimal possible size of the data. The lazy versions have a bigger overhead, but again the lazy array is an exception: it uses a huge amount of memory to run. This is mainly because the elements in the array are closures pointing to other (older) arrays which also contain closures. The array elements are not evaluated when updated, but the closures created instead are not evaluated until needed, which is at the end of the program. In this way almost all intermediate vectors are kept in memory.

3 Optimisations

3.1 Map fusion

Among the inefficiencies in the implementation of the previous section are the loops and intermediate vectors introduced by expressions like `tau .* d + x`. With a straightforward implementation, this expression is built with two functions: + and .*. These operators are implemented with array comprehensions. The expression can then be reduced as follows:

```
      tau .* d + x
(+)   { ye + xe \\ ye ←: tau .* d & xe ←: x }
(.*)  { ye + xe \\ ye ←: { tau * de \\ de ←: d } & xe ←: x }
```

Now it is clear that the expression will be evaluated by the execution of two loops, represented by the two array comprehensions. This is not the most efficient way to implement the expression. One loop applying a function to the vector elements is better:

```
{ tau * de + xe \\ de ←: d & xe ←: x }
```

To test the significance of this we replaced three functions of the conjugate gradient specification by more efficient ones:

```
CGloop :: Matrix Vector Vector Real Vector → Vector
CGloop a x g delta0 d
  | delta1 < Epsilon = x'
  | otherwise        = CGloop a x' g' delta1 d'
  where
    h      = a #* d
    tau    = delta0 / InnerProd d h
    x'     = SAXPY tau d x
    g'     = SAXPY tau h g
    delta1 = InnerProd g' g'
    beta   = delta1 / delta0
    d'     = SAXMY beta d g'

SAXPY :: Real Vector Vector → Vector
SAXPY a x y = { a * xe + ye \\ xe ←: x & ye ←: y }
```

From table 4, we see that this change had a positive effect on the heap consumption. The minimal heap size decreased by 20% and the garbage collection times went down, because we had less intermediate vectors. The efficiency increased by 20-40%, because we did not have to create the extra vectors.

	Min	Test run		
Array size	Heap size	Heap size	Exec time	GC time
625^2	50 Kb	128 Kb	0.15	0.00
2500^2	170 Kb	512 Mb	1.16	0.05
10000^2	650 Kb	2 Mb	10.68	1.14
40000^2	2580 Kb	8 Mb	110.17	11.08

Table 4. Unboxed arrays with map fusion

Unfortunately, map fusion must be done completely by hand at the moment. There are a number of theories to do it automatically [Wad90] [SF93], but it is very hard to implement them. Only one implementation doing cheap deforestation exists at the moment [GLPJ93]. It only works for lists (and Haskell arrays, but those are based on lists, see Section 4.2) and cannot deforest more than one function argument.

3.2 Update in place

In our implementation of the conjugate gradient algorithm, we did not make an extensive use of the uniqueness typing and update-in-place of Clean. Instead of updating old ones, we created new vectors in every function. Using uniqueness typing for update-in-place has one disadvantage: one has to return every unique argument used, because it may be updated later on. If we want to introduce update-in-place with uniqueness typing in the version with map fusion of the previous section, we have to alter the specification again, losing the elegance of using the standard overloaded operators:

```
CGloop :: Matrix *Vector *Vector Real *Vector *Vector → *Vector
CGloop m x g delta0 d h
    | delta1 < Epsilon = x1
    | otherwise        = CGloop m x1 g3 delta1 d4 h3
    where
        (h1, d1)         = MatMulVec h m d
        (ip_dh, d2, h2)  = InnerProd d1 h1
        tau              = delta0 / ip_dh
        (x1, d3)         = SAXPY tau d2 x
        (g1, h3)         = SAXPY tau h2 g
        (delta1, g2)     = InnerProd2 g1
        beta             = delta1 / delta0
        (g3, d4)         = SAXMY g2 beta d3
```

The vector and matrix functions change too. As an example, the implementation of SAXPY is given here:

```
SAXPY :: Real v:Vector *Vector → (*Vector, v:Vector)
SAXPY a x y = SAXPY' 0 x y'
where
    (n, y') = usize y

    SAXPY' :: Int v:Vector *Vector → (*Vector, v:Vector)
    SAXPY' i x y
        | i == n   = (y, x)
        | otherwise= SAXPY' (i + 1) x' { y' & [i] = a * xe + ye }
        where
            (xe, x') = uselect x i
            (ye, y') = uselect y i
```

Table 5 shows the performance when using update-in-place and map fusion. The code produced is approximately 2 times faster than the unoptimised version. We were able to set the heap size to its minimum, while garbage collection times are negligible, because when only using updates and tail recursions, no closures need to be built. Thus, except for the initialisation, the garbage collector is not used.

Another remarkable fact is that the minimal heap size is equal to that of the previous results with map fusion. The amount of 'live data', which determines the minimal heap size, is in both implementations the same. With a destructive update each live vector stays live. Without update-in-place, the old vector is copied and is eventually removed by the garbage collector, the copied vector will be the new live vector, so the amount of live data stays the same.

	Min	Test run		
Array size	Heap size	Heap size	Exec time	GC time
625^2	50 Kb	50 Kb	0.11	0.00
2500^2	170 Kb	170 Kb	0.91	0.00
10000^2	650 Kb	650 Kb	7.50	0.04
40000^2	2580 Kb	2580 Kb	76.24	0.24
160000^2	10350 Kb	10350 Kb	725.50	0.87

Table 5. Unboxed arrays with map fusion and update-in-place

4 Other languages

4.1 C

There have been quite a few claims that code produced by compilers for functional languages is comparable with code generated by compilers for imperative languages. Now that we have a fast code for this numerical algorithm, we can compare it against an equivalent C program.

The Clean program we compared with is the optimised version from the previous section. For the C code we used Sun's `cc` version 3.0.1, with `-O4` optimisation. The process size is the memory size reported by UNIX command `top`. The execution times in second were obtained using the system function `getrusage()`.

Table 6 shows that Clean is on average 30% slower than C. The difference in process size is caused by the stack of Clean: its default size is 512K. If we take that into account the process sizes are equal.

A big advantage of the Clean version over C is that it can be adapted easily: we did not change the program code for the algorithm when we used arrays

	Clean		C	
Array size	Process size	Exec time	Process size	Exec time
2500^2	1600 Kb	0.91	904 Kb	0.69
10000^2	2080 Kb	7.50	1432 Kb	5.20
40000^2	4008 Kb	76.24	3544 Kb	66.20
160000^2	12 Mb	725.50	12 Mb	608.75

Table 6. Comparing efficient Clean and C code for the CG algorithm

instead of lists. Using a different datatype for vectors in the C code has more effect on the C code of the algorithm.

4.2 GHC

We also compared the Clean code to Haskell code, using the Glasgow Haskell compiler (GHC). Two kinds of arrays were used: the Haskell array and the mutable _Bytearray, which is a non-Haskell extension of the Glasgow compiler.

The Haskell arrays, as implemented in GHC, are not primitive. It is a combination of a tuple for the boundaries and a list with associations between indexes and values. The type of a Haskell array is Array a b where a is the type of the index and b the type of the elements. The most important function for creating Haskell arrays is array:

```
array :: (Ix a) ⇒ (a,a) → [(a,b)] → Array a b
```

It takes the boundaries of the new array and a list of associations from indexes to values and creates a new Haskell array. An association between index i and value x is denoted as i := x. Other functions on Haskell arrays are the operator !, for selecting an element from an array and // for incrementally updating an array. In the current implementation of GHC this updating is not in-place, a new array is created instead. In the future, the compiler could figure out where updates are possible, using update analysis and linear types.

Remarkable is the similarity of the Haskell code to the Clean code for arrays without updating (figure 3). The main difference is that Clean's array comprehensions are similar to list comprehensions, while Haskell uses the array function in combination with a list comprehension.

We tested this implementation with version 0.29 of GHC, with options `-O -fvia-C -O2-for-C`. The second optimisation level `-O2` did not give an important speedup. As GHC's Haskell arrays are not primitive, the measured times are not so good (see table 7). We were only able to test for problem size 625^2. This makes the GHC implementation of Haskell arrays not suitable for numerical computing.

The mutable _Bytearray of the Glasgow compiler is a primitive array, comparable with the unboxed array of Clean. The elements of both arrays are stored

```
saxpy :: Double → Vector → Vector → Vector
saxpy a x y = array b [ i := a * x!i + y!i | i ← range b ]
    where b = bounds x

(#*) :: Matrix → Vector → Vector
(#*) (offset, d0, d1, d2) v =
    mul1 (n - offset) n d2 (mul1 (n - 1) n d1 mul0)
where
    b = bounds v
    n = snd b

    mul0 :: Vector
    mul0 = array b [ i := d0!i * v!i | i ← range b ]

    mul1 :: Int → Int → Diagonal → Vector → Vector
    mul1 0  i2 d u =
        u // [ 0 := u!0 + d!0 * v!i2, i2 := u!i2 + d!0 * v!0 ]
    mul1 i1 i2 d u = mul1 (i1 - 1) (i2 - 1) d
        (u // [ i1 := u!i1 + d!i1 * v!i2, i2 := u!i2 + d!i1 * v!i1 ])
```

Fig. 3. GHC code for the 'simple ax plus y' and matrix-vector multiplication functions

	Min	Test run		
Array size	Heap size	Heap size	Exec time	GC time
625^2	3950 Kb	16 Mb	67.38	41.62

Table 7. The performance of Haskell arrays for the algorithm

unboxed in one memory block and both can be updated in place. Where Clean uses uniqueness typing to do save update-in-place, GHC uses a monad [LPJ94]. Monads use a state to thread all functions using the state, which are called actions. Those actions are glued together using the combinator thenST :: ST s a → (a → ST s b) → ST s b: it executes the first action and passes the result, together with the updated state, to the second action. The result of the combination of the two actions is the result of the second action. The combinator seqST is similar to thenST, but now the second action does not get the result of the first action. Finally, the function returnST simply returns a value, without changing the world. All three combinators above have a variation which is strict in the state, respectively thenStrictlyST, seqStrictlyST and returnStrictlyST.

Monads have a big influence on the code:

```
saxpy :: Double → Vector s → Vector s → _ST s (Vector s)
saxpy a x y =
    case boundsOfByteArray x of {
        (p,q) → saxpy' p
            where
                saxpy' :: Int → _ST s (Vector s)
                saxpy' p
                    | p > q      = returnStrictlyST y
                    | otherwise =
                        readDoubleArray x p                   `thenStrictlyST` \xe →
                        readDoubleArray y p                   `thenStrictlyST` \ye →
                        writeDoubleArray y p (a * xe + ye) `seqStrictlyST`
                        saxpy' (p + 1)
    }
```

While the implementations of Clean's unboxed and GHC Bytearrays are similar, but they do not give the same results (table 8). The remark from Glasgow was that the state monad was not fast enough and that they have not had time to make it faster.

	Clean			Glasgow Haskell		
Array size	Heap size	GC time	Exec time	Heap size	GC time	Exec time
625^2	50 Kb	0.00	0.11	256 Kb	0.19	1.83
2500^2	170 Kb	0.00	0.91	1 Mb	0.71	13.30
10000^2	650 Kb	0.04	7.50	4 Mb	5.21	103.54

Table 8. Comparing the time and space requirements for the most optimal Clean implementation for the conjugate gradient algorithm and one implemented with GHC's _Bytearray

The list implementation in GHC was as fast as we expected, with a performance comparable to the Clean version with lists. Remarkably, this version is also the fastest implementation we had with GHC (see table 9).

5 Conclusions

In this paper we investigated various implementations of the conjugate gradient algorithm. The first implementations in Clean with lists and arrays showed that the overhead imposed by the lazy data type is high, especially memory consumption can be extreme. List are only an alternative to arrays when speed is not important or when arrays do not give good performance.

A fast version of the implementation in Clean was obtained using an unboxed array and by the inclusion of map fusion and update-in-place. When compared

Array size	Clean			Glasgow Haskell		
	Min Heap	GC time	Exec time	Min Heap	GC time	Exec time
625^2	850 Kb	0.32	1.09	820 Kb	0.58	1.06
2500^2	6170 Kb	2.68	9.06	5900 Kb	1.80	7.97

Table 9. Comparing implementations of the conjugate gradient algorithm in Clean and GHC, both GHC with lazy lists. The heap size for the problem sizes are for both languages 2 Mb and 16 Mb respectively

to C, that code proved to be only 30% slower, while the amount of allocated memory was the same. The Clean program looks similar to the C version, but because of the absence of side-effects, program transformations are easier.

The GHC implementation of Haskell arrays proved to be less suited for scientific computing. The code which can be written with them is clean, but the speed and memory consumption are not good. The current implementation of the _Bytearray provided by the Glasgow Haskell Compiler gives better performance: the memory consumption was reasonable, but the execution time can be improved. The list implementation was the fastest we had with the GHC compiler.

References

[BS93] E. Barendsen and J.E.W Smetsers. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, Computer Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, December 1993.

[CSJ95] S. Choi, T.K. Sarkar, and Choi J. Adaptive antenna array for direction-of-arrival estimation utilizing the conjugate gradient method. *Signal Processing (SIGPRO)*, 45(3):313–327, Sep 1995.

[EO94] R.G. Ellis and D.W. Oldenburg. The pole-pole 3-d dc resistivity inverse problem: a conjugate gradient approach. *Geophysical Journal International*, 119:187–194, 1994.

[GLPJ93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232, April 1993.

[H+96] P.H. Hartel et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal functional programming*, 6(4):621–655, July 1996.

[HPJW92] P. Hudak, S.L. Peyton Jones, and P. Wadler. Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[HS93] A.G. Hoekstra and P.M.A. Sloot. Implementation of a parallel conjugate gradient method for simulation of elastic light. *4th Physics Computing '92*, pages 345–346, 1993.

[JDG92] E.M. Johansson, F.U. Dowla, and D.M. Goodman. Backpropagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems*, 2(4), 1992.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., California, 1994.

[LPJ94] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[PJ96] S.L. Peyton Jones. Bulk types with class. In *Proceedings of the 1996 Glasgow Workshop on Fucntional Programming*, Ullapool, Scotland, July 1996.

[PJL91] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In *FPCA'91 — Functional Programming Languages and Computer Architecture*, pages 636–666, 1991.

[PvE93] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Publishers Ltd., 1993.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 233–242, Orlando, April 1993. ACM Press.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

This article was processed using the LaTeX macro package with LLNCS style