A Clean Conjugate Gradient Algorithm

P.R. Serrarens

Computer Science Institute, University of Nijmegen
Nijmegen, The Netherlands

Abstract. This paper shows the usage of functional programming in the area of scientific computing. It is an important area because time and space efficiency are crucial: many scientific programs work with large data sets and run for a long time. The paper shows that functional programming can be a real alternative to imperative methods. The example used in this paper is the conjugate gradient algorithm. It will show that the functional way is cleaner, easier to adapt and almost as quick and lean as imperative implementations.

We are in the middle of a functional programming revolution. After years of relative silence, the late 1980's brought new interest in functional programming. This lead to the development of a large number of compilers with often remarkable achievements. Nowadays many problems of functional languages have been tackled, and it seems to be that functional programming is a real universal programming paradigm that can replace the traditional imperative languages.

As said, many problems have been tackled, but there are still some doubts about the usability of functional programming languages in a number of area's. One of these is the Formula One of computer science: scientific computing, setting extreme requirements on speed and memory usage, two topics which are traditionally weak points of functional programming. Though, even this is being tackled at the moment. Sisal has already gained quite some success on sequential and parallel architectures, being even faster than Fortran [Can92]. In this paper we want to show that a general purpose lazy functional language can also perform well in scientific computing, being as quick as C.

An algorithm used many times in scientific computing is the conjugate gradient algorithm. It's an iterative method to solve the equation $Ax = b$, for a given matrix A and vector b. The paper does not explain the algorithm itself; it's about implementing it in Clean, a lazy functional programming language [PvE93], in a way that results in a well build and efficient program.

One of the nice things about functional programming is that one can write good specifications with it. We will use this property to build up our implementation of the conjugate gradient algorithm. Beginning with a specification of the algorithm, we will look what parts are lacking and design those parts. Hereafter we will look at the performance of the program. Next, in section 3 and 4, two ways will be discussed to achieve better performance, namely map fusion and array update. The new, more efficient, Clean program will be compared with C again in section 5. Lastly, the results will be discussed in 6 and current and future research will be described.

1 A Specification of the Algorithm

The conjugate gradient can be specified in Clean fairly easy. The only thing we can't do is using the *-symbol to denote matrix-vector and scalar multiplication. This is because the overloading mechanism restricts the use of *-multiplication symbol to that with two arguments of the same type. To overcome this problem, we use #* for matrix-vector multiplication and .* for scalar multiplication. The next piece of code shows the algorithm as it can be specified in Clean.

```
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‘     = x + tau .* d
        g‘     = g + tau .* h
        delta1 = InnerProd g‘ g‘
        beta   = delta1 / delta0
        d‘     = beta .* d - g‘
```

A more detailed description of the algorithm can be found in [KGGK94]. For this paper it's sufficient to say that the algorithm approximates the equation $Ax = b$, with x being an unknown vector, A a given matrix and b a given vector.

1.1 The Vectors

Vectors can be implemented easily in Clean, since we have arrays. The most efficient array offered is the unboxed array, denoted as {#x}, where the array is stored in a continuous block of memory. So a good representation of a vector is:

```
::  Vector :== { #Real }
```

A number of operations are defined on vectors, like addition, negation, scalar multiplication and inner product. Having ZF-expressions for arrays (called array comprehensions), we can express those operations easily:
(we assume here that the list functions `fold`, `map` and `zipwith` are overloaded for use with both lists and array's. The construction `a.[i]` denotes the selection of the `i`-th element of array `a`)

```
instance ~ Vector
where
    ~ v = map ~ v

instance + Vector
where
    (+) v1 v2 = zipwith (+) v1 v2
```

```
(.*) infixl 7 :: Real Vector -> Vector
(.*) x v = map ((*) x) v

InnerProd :: Vector Vector -> Real
InnerProd v1 v2 = InProd (size v1 - 1)
where
    InProd :: Int -> Real
    InProd 0 = v1.[0] * v2.[0]
    InProd i = v1.[i] * v2.[i] + InProd (i - 1)
```

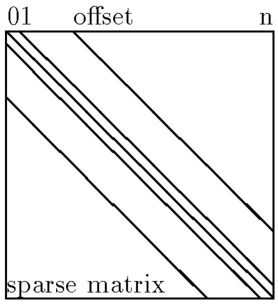
New instances are made for the negation ~ and addition + operators, because they are overloaded. A new operator is defined to denote the scalar multiplication. The implementation of the inner product may look a little ugly, but it is the most efficient way to do it. The alternative is to use fold- and map-like functions to calculate the result:

```
InnerProduct :: Vector Vector -> Real
InnerProduct v1 v2 =
    foldl1 (+) (zipwith (*) v1 v2)
```

It looks better, but it is less efficient, because it needs two scans over the length of the vectors: one for the multiplication of the two input vectors and one for the folding with the addition operator. (see also chapter 3)

1.2 The Matrix

The representation of the matrix needs some more thinking. A straightforward way is to represent it as an array of arrays, but that almost never a good choice, because the conjugate gradient method is a method to solve large sparse systems of linear equations. This means two things. The first is that the matrix is usually large very large (often larger than 100.000×100.000). With a direct representation it will take more than 74 gigabytes to store a matrix of size 100.000×100.000.



The second fact is that the matrix is sparse, meaning that many matrix elements are zero. In our example case the all non-zero elements are aligned along one of the five diagonals. Combining this with the fact that the matrix is symmetric, we only have to store 3 values per row, with the result that we only need a little more than 2 megabytes to store the matrix above.

The diagonals in the matrix are parallel to the main diagonal, in fact three diagonals form a band in the middle and the two other are a little farther away (see above). In our implementation we only store the upper three diagonals with the offset of the third, outermost diagonal:

```
:: Matrix :== ( Int, { #Real }, { #Real }, { #Real } )
```

The matrix-vector multiplication is the hardest part of the specification, because it depends heavily on the representation of the matrix. Every value in the resulting vector is the result of an inner product of the vector and the corresponding row of the matrix. Since we don't store the rows, we have to do it differently.

Going from the middle to the outside diagonals, we first multiply the middle diagonal with the vector. Then, we go to the next two diagonals. For the upper diagonal we multiply the first $n-1$ elements of the vector with the elements of the diagonal and add that to the result. The lower diagonal goes similar, but then with 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:
(The construction { `a & [i] = x` } means array `a` updated at position `i` with `x`. Pattern matching on array's can be done with `a=:`{`[i]=x`}, where `x` is extracted from the `i`-th position of array `a`).

```
(#*) infixl 7 :: Matrix Vector -> Vector
(#*) (offset, d0, d1, d2) v =
    ( Mul1 (n - offset) n d2  o  Mul1 (n - 1) n d1  o  Mul0 ) v
where
    n = size v - 1

    Mul0 :: Vector -> *Vector
    Mul0 v = { de * ve \\ de <-: d0 & ve <-: v }

    Mul1 :: Int Int { #Real } *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] }
```

1.3 Summary

We now have an implementation for the conjugate gradient algorithm using the specification introduced in the first part of this section. It is supported by some straightforward vector operations and a matrix-vector multiplication based on the symmetrical sparse matrix we introduced here. In the next section we will look at the efficiency of it. Will it compete with implementations of the algorithm in, for example, C?

2 Time and Space behaviour

To test the efficiency of the code in the previous section, we tested it against optimal C code used by mathematicians at our university. 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 C code was compiled with cc version 3.0.1, with -xO4 optimising. The compiler for Clean was Clean 1.1 for Solaris. In table below a comparison is made between the conjugate gradient algorithm in Clean and C.

	Clean		C	
Array size	Heap size	Exec time	Exec time	Speedup
625	128 Kb	0.18	0.10	1.8
2500	512 Kb	1.45	0.72	2.0
10000	2 Mb	14.57	5.90	2.5
40000	8 Mb	159.13	75.70	2.1

As can be seen from the table, the C versions of the programs were approximately 2 times as fast as the Clean programs. This is remarkably good, as we hardly cared about efficiency yet (except for the inner product). So you might expect that we can do better than this. Therefore we will look at two optimisations in the next two sections, namely map fusion and update-in-place.

3 Intermediate values

Among of the inefficient things are the extra loops and intermediate vectors caused by expressions like $\bar{v} + x \cdot \bar{u}$. With a straightforward implementation like we have at the moment, this expression will be transformed into two subsequent expressions, because it consists of two functions:

```
v + x .* u
{ ve + ye \\ ve <-: v & ye <-: (x .* u) }
{ ve + ye \\ ve <-: v & ye <-: { x * ue \\ ue <-: u } }
```

It's now visible that the expression will be evaluated by execution of two loops, represented by the two array comprehensions. Also visible are the intermediate values (`ye`) to carry values from one loop to the other. This is most likely not what programmers have in mind when writing the first expression. They expect one loop applying the function to the vectors, thus something like:

```
{ ve + x * ue \\ ve <-: v & ue <-: u }
```

It consists of only one loop and doesn't need the intermediate values.

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

```
x‘ = { xe + tau * de \\ xe <-: x & de <-: d }
g‘ = { ge + tau * he \\ ge <-: g & he <-: h }
d‘ = { beta * de - ge \\ de <-: d & ge <-: g }
```

This adaption increased the efficiency of the program by 20-40% as can be seen in the table below. The garbage collection time went down too, because less intermediate vectors were needed.

		Standard Clean		Clean with transf.		
Array size	Heap Size	Exec time	GC time	Exec time	GC time	Speedup
625	128 Kb	0.18	0.03	0.15	0.00	1.2
2500	512 Kb	1.45	0.13	1.16	0.05	1.3
10000	2 Mb	14.57	1.76	10.68	1.14	1.4
40000	8 Mb	159.13	21.83	110.17	11.08	1.4

Unfortunately, map fusion must be done completely by hand at the moment. The second disadvantage is that it changes the specification and that the specification is not independent from the vector definition, because the optimised functions use the structure of the vector definition.

The technique for doing such transformations is usually called deforestation and many work has already been done on that topic, with the paper of Wadler [Wad90] begin one of the first. However, only one usable implementation of it exists [GLJ93], which is only applicable in a subset of cases. It only works for functions which take one list. It cannot deforest operations like addition for both arguments.

4 Array updates

Functional languages used to be rejected because they couldn't do update-in-place. Functional programmers tended to say that you didn't need update-in-place, only when you were in need of full speed but then you should use languages like C, Fortran or even assembly instead of a functional language.

Nowadays we can do update-in-place through of a number of techniques like uniqueness typing [BS93], monads [LJ94] and copy elimination [GH89]. In our implementation of the conjugate gradient algorithm, we didn't use the uniqueness typing system of Clean. Instead of update-in-place, new vectors were created everywhere. This is not very efficient: for the n = 10.000 problem, 90 Mb of memory is allocated during the run, just for the intermediate vectors. We removed some intermediate vectors in the previous section, but even then 51 Mb of memory is allocated. If update-in-place is used well, we can bring this back to 0 Mb.

Using uniqueness typing for update-in-place has one disadvantage: one has to return every unique type used. Take for example the following piece of code:

```
g x y z = y‘‘
where
```

```
(x‘, y‘) = f x y
(y‘‘, z‘) = f y‘ z
```

The function `f` updates its first argument in place. The result of `f` is returned as the first element of a tuple. The second element of that tuple is the second argument, unchanged. This second element is needed when it will be updated later on. In the program code above, `y‘` is needed to do the update-in-place in `f y‘ z`, because the first argument must be unique. When `y` was used here, it wouldn't be unique, because it also used in `f x y`.

So, when using uniqueness typing to get update-in-place, we have to alter the specification:

```
CGloop a x g delta0 d h
    | (delta1 <= Epsilon) = x1
    | otherwise           = CGloop a x1 g3 delta1 d4 h3
    where
        (h1, d1)        = MatMulVec h a d
        (ip_dh, d2, h2) = InnerProd d1 h1
        tau             = delta0 / ip_dh
        (x1, d3)        = VecPlusScalVec x tau d2
        (g1, h3)        = VecPlusScalVec g tau h2
        (delta1, g2)    = InnerProd2 g1
        beta            = delta1 / delta0
        (g3, d4)        = ScalVecMinVec beta d3 g2
```

This will alter the vector and matrix functions too. As an example, the implementation of `VecPlusScalVec` is given below:

```
VecPlusScalVec :: *Vector Real *Vector -> (*Vector, *Vector)
VecPlusScalVec v1 x v2 = VPSC (n - 1) v1 v2‘
where
    (n, v2‘) = usize v2

    VPSC :: Int *Vector *Vector -> (*Vector, *Vector)
    VPSC 0 v1=:{[0]=v1e} v2=:{[0]=v2e} =
        ({ v1 & [0] = v1e + x * v2e }, v2)
    VPSC i v1=:{[i]=v1e} v2=:{[i]=v2e} =
        VPSC (i - 1) { v1 & [i] = v1e + x * v2e } v2
```

This is all certainly much less elegant than without update-in-place. Again, we would like to get update-in-place without changing the specification. Monads give better code:

```
CGloop a x g delta0 d h =
    MatMulVec h a d          >>
    InnerProd d h            >>= \ip_dh ->
    return (delta0 / ip_dh)  >>= \tau ->
```

```
VecPlusScalVec x tau d   >>
VecPlusScalVec g tau h   >>
InnerProd g g            >>= \delta1 ->
if (delta1 <= Epsilon) then
(   return x1
)
(   return (delta1 / delta0) >>= \beta ->
    ScalVecMinVec beta d g   >>
    CGloop a x g delta1 d h
)
```

However, it isn't much clearer, because it's not obvious which nodes are updated. The function `VecPlusScalVec x tau d` might represent either $\bar{x} = \bar{x} + tau \cdot \bar{d}$ or $\bar{d} = \bar{x} + tau \cdot \bar{d}$, you can't tell from the code which vector is updated.

In table below the effect of using uniqueness can be seen. We were able to halve the heap size without loss of efficiency, the garbage collection times are even lower than without update-in-place. The average performance increase is 40%. So in contrary to results from Sisal [Can92], we don't suffer heavily when not using update-in-place. This is because strict functional languages create much more closures than lazy functional languages, because they lack sharing.

	Clean with transformations			Clean with array update			Speed-
Array size	Heap	Exec time	GC time	Heap	Exec time	GC time	up
625	128 Kb	0.15	0.00	64 Kb	0.11	0.00	1.4
2500	512 Kb	1.16	0.05	256 Kb	0.91	0.01	1.3
10000	2 Mb	10.68	1.14	1 Mb	7.15	0.02	1.5
40000	8 Mb	110.17	11.08	4 Mb	74.53	0.27	1.5

5 Clean vs. C - The final

Finally, let's return to the Clean against C comparison again. We started with the result that Clean was more than 2 times as slow as C. What will the situation be when we use the Clean version with map fusion and update-in-place? The table below shows us the results. Clean is on average 21% slower than C.

	Clean with array update		C	
Array size	Heap size	Exec time	Exec time	Speedup
625	64 Kb	0.11	0.10	1.1
2500	256 Kb	0.91	0.69	1.3
10000	1 Mb	7.15	5.20	1.4
40000	4 Mb	74.53	66.20	1.1
160000	16 Mb	695.70	608.75	1.1

When doing scientific computing memory usage is also very important, what is the memory consumption compared with C? We haven't been very careful with the heap size, it has been determined rather arbitrarily.

Some trail-and-error experiments showed that identical execution times can be achieved for array sizes 10.000 and 40.000 with an heap size of 650 Kb and 2800Kb respectively. With those heap sizes, tests with `ps` showed that the memory usage of the Clean program was close to that of C. The Clean program reserved for the problems of size 10.000 and 40.000 respectively 2.064 and 4.216 kilobytes of memory, the C program 1.424 and 3.546 kilobytes. This shows that the memory usage for rather large problems is within 150% of the C memory needs.

6 Conclusion and Future Work

We have showed that functional languages can be used to write a specification of an algorithm like the Conjugate Gradient in a clean way. Furthermore, we showed that a straightforward implementation of it is approximately 2 times as slow as a comparable implementation in C. With two adaptions, namely map fusion and update-in-place we made the functional program nearly as fast as the C version with comparable memory needs.

The drawback of it is that we can't use the specification of the first section anymore. Due to the uniqueness typing, we had to return all unique arguments of a function, making the specification much less readable. At the moment a new notation is tested which eliminates this explicit threading and renaming of variables.

The other transformation needed was the elimination of intermediate results and extra loops through map fusion. This also affected the specification, because we had to replace easy-to-understand infix operators by integrated functions performing multiple operations at once. Currently, we are working on a deforestation technique to do this transformation automatically, by compile-time transformations.

Another topic of research is parallelism. Can we use the same specification for the algorithm, achieving good parallel performance? This seems to be true, the main difference is that the vectors will be distributed over the processors, only complicating the matrix-vector multiplication. It is to be seen whether we can get good efficiency with clear code here.

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.

[Can92] D. Cann. Retire fortran? a debate rekindled. *Communications of the ACM*, 35(8):81–89, Aug 1992.

[GH89] K. Gopinath and J.L. Hennesy. Copy elimination in functional languages. In *Proceedings of POPL'89*, 1989.

[GLJ93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming and Computer Architecture (FPCA'93)*, April 1993.

[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.

[LJ94] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

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

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