Numerical Software Development with Functional Languages*

Chris Angus
Dept. of Computing Science
University of Newcastle upon Tyne
NE1 7RU
c.m.angus@ncl.ac.uk
http:://www.cs.ncl.ac.uk/people/c.m.angus/

August, 1996

Abstract

Functional programming languages such as Haskell allow numerical algorithms to be expressed in a concise, machine-independent manner that closely reflects the underlying mathematical notation in which the algorithm is expressed. Unfortunately the price paid for this level of abstraction is usually an increase in execution time and space usage.

Ongoing work towards the goal of developing efficient large-scale numerical software in a functional language will be described in two stages:

Use of the purely functional language Haskell for the development of a (problem / dimension)-independent finite element analysis system and the advantages of working in this style, focusing on such areas as the usefulness of Haskell-style type classes and the appropriateness of non-strict semantics in this area.

Lessons learned from experiences described above which contribute to design decisions, implementation techniques and necessary multi-parameter extensions to Haskell-style type classes for a functional language specialised to the area of numerical software development.

1 Introduction

It has often been claimed that the use of functional programming languages for scientifically significant computations offers the advantages over procedural languages of modularity, conciseness, clarity, ease of proof, ease of introducing problem-oriented notation and independence from hardware peculiarities [6]. Since computer performance is increasing all the time we are now in the position of being able to solve problems of greater complexity than was previously possible, Unfortunately it is a sad fact that as

*This work was supported by EPSRC grant 94313569

problem complexity increases often program complexity increases also[1]. It is for exactly this reason that clarity, conciseness and correctness of code are important. A *"functional (or applicative) style is necessary to realise the promise of general-purpose parallel programming"* [26]. This promise draws from the fact that there is no underlying concept of a physical machine, store or sequencing, and although implicit parallel functional programming has often been promised, and to an extent never delivered, we believe that in the area of scientific programming this promise can be achieved[2].

1.1 Functional programming and scientific computing

The main advantages of using a functional language in this application domain are clear; programs can be written more quickly, more concisely, at a higher level (resembling more closely traditional mathematical notation) and are more amenable to formal reasoning, analysis and transformation-based compilation. Some of the disadvantages in this area are also clear; I/O is not as straightforward as in imperative languages[3]; low level machine actions are hard to program; often the price paid for the level of abstraction is an increased execution time and space usage [4]. We also need to consider issues relating to numerical analysis; execution efficiency is paramount; legacy code exists and will for many years to come[5]; twiddling will always be required[6]. In the design of any realistic tool for work in this area each of these points should be taken into account, maximising advantages and minimising the disadvantages while still respecting the numerical analysis aspects, making necessary compromises along the way between mathematical/theoretical elegance and pragmatic considerations.

1.2 Plan of study

The remainder of this paper will consider

1. Use of the functional language Haskell[15] for the development of numerical software including a (problem/dimension)-independent finite element analysis system.

2. Lessons learned from the above contributing to

   - design decisions
   - extensions
   - implementation techniques

   for a functional language specialised in the area of numerical analysis.

An appendix of further reading in the area of numerical analysis and applicative programming is also included.

[1] Picking through a PVM or MPI code should convince the reader of this.

[2] Given that the algorithm is not strictly seqential.

[3] This is as much a matter of unfamiliarity as conceptual difficulty especially with the increasing use of monads.

[4] This is especially true for lazy functional languages.

[5] Asking practitioners to re-do 40+ years of work is not feasible.

[6] Although not desirable, access to all levels will always be demanded.

2 Use in practice

2.1 Overview of the underlying mathematics

The area of mathematics we concentrate on for the purpose of this paper is the solution of partial differential equations (PDEs) via the finite element method (FEM) [3]. The FEM approximates the solution to a PDE over a region of interest. The region is descretised using an element mesh and the solution of the PDE is computed exactly at the node points of these elements. This is usually done by representing the solution of the PDE at these node points as the solution of a linear system $Ax = b$. The construction of A and b requires integrations to be performed over each element. The solution of the PDE over the region is then approximated using interpolation functions that depend on the values at neighbouring node points. This very simple explanation of the FEM does injustice to the method and is only meant to form a context for the following material, rather than an introduction to the method. For a more in depth introduction see [3, 1]. The sections which follow detail the Haskell implementation of methods which are used in the FEM such as linear system solvers, integration routines and interpolation functions; consider possible matrix representations; and look at problem specification using Haskell.

2.2 LU factorisation

A significant component of numerical problem solving is concerned with the solution of linear systems. In this area we find that algorithms such as LU factorisation can be expressed in a way that matches very closely their mathematical definition. The array comprehensions of Haskell naturally lend themselves to this area.

The LU factorisation of an $n \times n$ matrix A, consists of two matrices L and U such that A is expressed as the product $A = LU$, U being an $n \times n$ upper-triangular matrix and L being an $n \times n$ lower-triangular matrix. Since both of these systems are triangular they can be solved via forward and back substitution. If we are to perform this factorisation we must decide which form of $L+U$ to take as there are an infinite number of possibilities. If we set $l_{ii} = 1$ then by omitting explicit representation of this diagonal we can store the factorisation in the same amount of space as A. The mathematical formulation for this can be expressed as

$$
\begin{aligned}
l_{ij} &= \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \Big/ u_{jj} && (i > j) \\
u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} && (i \leq j)
\end{aligned}
$$

and the Haskell implementation as

```
> lu a = luf where
>  luf = array bnds [(i,j) := f i j | (i,j) <- range bnds]
>  bnds = bounds a
>  f i j | (i>j)  = (a!(i,j)-sum[luf!(i,k)*luf!(k,j)
>                                |k<-[1..(j-1)]])/luf!(j,j)
>        | (i<=j) =  a!(i,j)-sum[luf!(i,k)*luf!(k,j)
>                                |k<-[1..(i-1)]]
```

The 1-1 relationship between the mathematical specification and the Haskell implementation is valuable as it *allows direct correspondence of functional code with the mathematical equations, which makes the code easy to develop, write, read and modify* [9].

2.3 n-point Gaussian quadrature

Another significant component of numerical problem solving is the area of numerical quadrature, where an integral is approximated via one of a variety of methods. The applicative code to carry out an algorithm such as Gaussian quadrature is also as terse yet expressive as the mathematical specification. The use of lists greatly increases our ability to implement very general *n-point* quadratures simply by passing the appropriate weights and abscissae to the integration algorithm.

The method we consider estimates a definite integral

$$I = \int_a^b f(x)dx$$

by approximating f using a polynomial and integrating analytically the approximating function. The estimate is of the form

$$R = \sum_{i=1}^{n} w_i f_i$$

where $f_i = f(x_i)$ and the x_is are the *abscissae* or *integration points* with corresponding *weights* w_i. The formulation given below is a *symmetric* n-point Gaussian quadrature

$$\int_a^b f(y)dy \simeq p_r \sum_{i=1}^{n} w_i(f(p_m + p_r x_i) + f(p_m - p_r x_i))$$

$$\text{where} \quad \begin{aligned} p_m &= \tfrac{b+a}{2} \\ p_r &= \tfrac{b-a}{2} \end{aligned}$$

and the Haskell implementation is given by

```
> gq x w f (a,b) = pr * sum (zipWith f1 w x)
>  where f1 wi xi = wi * (f (pm+pr*xi) + f (pm-pr*xi))
>        pm = (b+a)/2.0
>        pr = (b-a)/2.0
```

where `zipWith` is a higher-order function which applies a function `f` elementwise to two lists such that, for example, `zipWith (+) [1,2] [3,4] = [4,6]`

2.4 Multidimensional integration

The use of quadrature can be extended to encompass multidimensional integration very easily by using the higher order nature of functional languages to write a general n-dimensional integration routine, passing an arbitrary 1-dimensional quadrature function as an argument to a higher-order multiple integrating function.

```
> multipleIntegral int = mi int []
>  where
>    mi int g [ab]      f = int f1 (ab g)
>          where f1 x = f (g++[x])
>    mi int g (ab:lims) f = int f1 (ab g)
>          where f1 x = mi int (g++[x]) lims f
```

As an example of this in use we can define the function `integrate` by combining our multi-dimensional integration function `multipleIntegral` and our Gaussian quadrature function `gq` with appropriate abscissae and weights.

```
> integrate = multipleIntegral (gq abscissae weights)
>  where abscissae = ...
>        weights   = ...
```

Using these functions we can very simply compute the volume of a sphere, or area of a circle, by specifying appropriate limits. The required formulae are

$$area_r = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} dydx$$

$$volume_r = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-y^2-x^2}}^{\sqrt{r^2-y^2-x^2}} dzdydx$$

and the integration limits can be expressed as

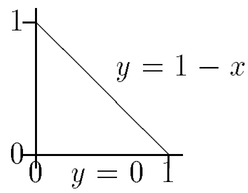
```
> circle_lims r = [lim1,lim2]
>  where lim1 _  = (-r,r)
>        lim2 (x:_) = (-sq,sq)
>         where sq = sqrt(r*r - x*x)

> sphere_lims r
>  = let
>     [lim1,lim2] = circle_lims r
>    in
>     [lim1,lim2,lim3]
>       where lim3 (x:y:_) = (-sq,sq)
>               where sq = sqrt(r*r - y*y - x*x)
```

We can now express the final computation very tersely as we would expect it to appear in a text i.e. $\int_{circle} dydx$.

```
> area   r = integrate (circle_lims r) (const 1)
> volume r = integrate (sphere_lims r) (const 1)
```

2.5 Basis functions for finite element analysis

More usefully, these ideas can be extended to the specification of interpolation functions for finite element analysis where we can very generally state facts about iso-parametric[7] basis functions. The essence of the FEM is to partition the domain of a problem into non-overlapping *elements* and provide an approximate solution which has a simple form within each element. The local representations are then patched together to form a global solution. The approximate solution is calculated using *basis* or *interpolation* functions dependent on the values at neighbouring node points. The simplest approach is to interpolate linearly within an element



In Haskell we can write this as

```
> simplex_2D    = (interpolations,limits,(dimensions,nodes))
>  where interpolations = [i1,i2,i3]
>             where i1 (r:s:_) = 1 - r - s
>                   i2 (r:s:_) =     r
>                   i3 (r:s:_) =         s

>       limits = [l1,l2]
>             where l1 _     = ( 0 ,  1 )
>                   l2 (x:_) = ( 0 , 1-x )

>       nodes  = 3

>       dimensions = 2
```

allowing us to model simplex elements very naturally, since the interpolation functions for a 1D simplex are

$$\begin{aligned} H_1(r) &= 1 - r \\ H_2(r) &= r \end{aligned}$$

[7]given a standard representation by being mapped onto a reference element in a unit region.

and the interpolation functions for a 2D simplex are

$$\begin{array}{rr|r} H_1(r,s) = & 1-r & -s \\ H_2(r,s) = & r & \\ H_3(r,s) = & & s \end{array}$$

This explains our use of `(x:y:_)` rather than `(x,y)` in that we express functions in a form that allows us to speak more generally later about the dimensionality. A similar idea is used in the multidimensional integration algorithm.

Now that an element has been specified, one of the things we need to do is integrate a function over its limits, which also can be specified very tersely.[8]

```
> getLimits (_,limits,_) = limits

> integrateOverElement = integrate.getLimits
```

Now we can integrate any function f over the element simply via

```
> integrateOverElement simplex_2D f
```

2.6 Quadtrees

Functional languages generally shy away from the use of arrays and tend to use structures such as lists or trees instead, writing algorithms in a very catamorphic[9] way. Two structures that lend themselves to this area are the quadtree and binary tree, which can be used to represent matrices and vectors respectively.

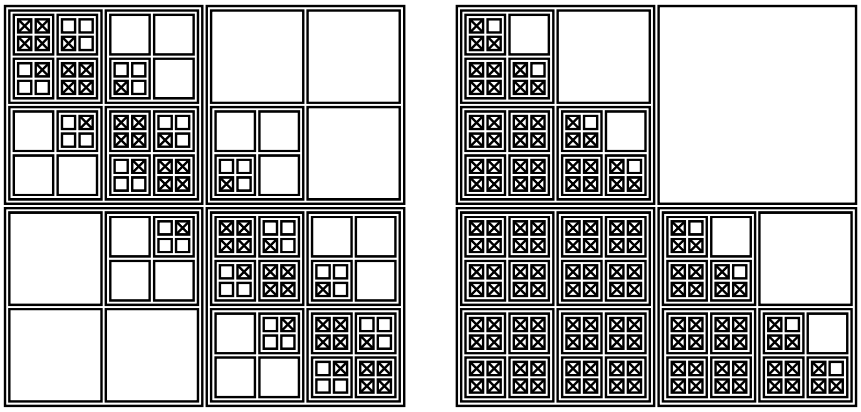


Figure 1: Tri-diagonal and Lower-triangular matrices as quadtrees

A matrix of order 2^p can be represented by a quadtree of depth p, where the tree is either;

- a zero matrix represented by 0;

[8]In Haskell `f.g` $\equiv f \circ g$ denotes function composition.
[9]Function following form.

- a scalar multiple aI $(a \neq 0)$ of the identity matrix represented by a;
- a 2×2 block matrix, each sub-matrix being of order 2^{p-1}.

Some of the advantages of using quadtrees are that they are useful for both sparse and dense storage patterns; Their asymptotic space complexity is linear in the number of non-zeros for common matrices[27][10]; and matrix algorithms can be expressed naturally as block algorithms using quadtrees.

For example an LU factorisation can be expressed using a block formulation as

$$\text{blockLU} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} \\ L_{21} & U_{22} \end{bmatrix}$$

$$\begin{aligned} \text{where} \quad U_{11} &= \text{blockLU}\ A_{11} \\ U_{12} &= \text{fwdSubst}\ U_{11} A_{21} \\ L_{21} &= \text{bkSubst}\ U_{11} A_{21} \\ U_{22} &= \text{blockLU}\ (A_{22} - L_{21} U_{12}) \end{aligned}$$

$$\text{blockLU}\ a = a$$

where we pattern match on the tree to determine whether decomposition is necessary.

2.7 Finite element analysis

The techniques described in the previous sections were brought together to form a simple finite element package implemented in Haskell. One of the requirements was that it should work in an arbitrary number of dimensions with plug-in basis functions and a plug-in linear system solver.

In the development of this system, the notation provided by Haskell was found to be very useful, often making the problem description clear. For example, in the FEM we establish the element equations for each element. Generally this is done by substituting the interpolation functions into the governing integral form. Historically these matrices have been called the *element stiffness matrix* and *load (or force) vector*, respectively[11]. Once the element equations have been established the contribution of each element is added to form the *system equations* $Sx = F$. This can be expressed in Haskell as

```
> linear_system :: Mesh -> Basis_Fn -> Linear_System

> linear_system mesh shape = MkLinear_System sm fv
>   where sm = sum [contrib_to_sm shape e | e <- elements]
>         fv = sum [contrib_to_fv shape e | e <- elements]
>         elements = getElements mesh
```

[10] It is possible to construct pathalogical matrix patterns that refute this such as permutation matrices. These however can be stored as Ahnentafel index trees as in [27].

[11] This naming scheme relates to a time where the FEM was predominately used for structural analysis.

where the element stiffness matrix and force vector depend on the interpolationg function `shape` and the particular element being considered.

By using Haskell's type class mechanism to define instances over an *Expression* datatype, high level descriptions of problems can be specified very naturally.

Consider the simple 1-D linear slider bearing shown in Figure 2, which is assumed to extend to infinity out of the plane of the figure[1](p. 207). It consists of a rigid bearing and a slider moving relative to to the bearing with velocity U.

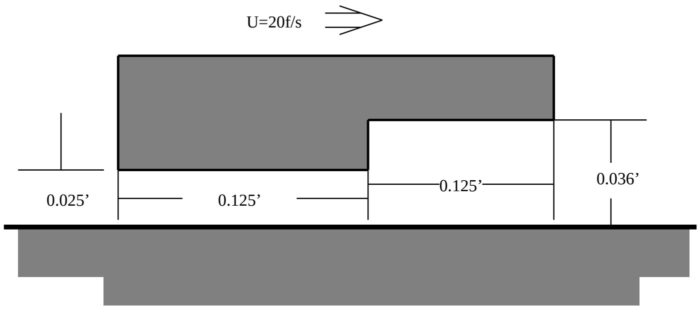


Figure 2: Linear Slider Bearing

The extremely thin gap between the bearing and the slider is filled with an incompressible lubricant having viscosity v. For the one-dimensionsal case the governing Reynolds equation reduces to

$$\frac{d}{dx}\left(\frac{h^3}{6v}\frac{dP}{dx}\right) = -\frac{d}{dx}(Uh)$$

where $P(x)$ denotes the pressure and $h(x)$ the distance between slider and bearing. The boundary conditions are that P must equal the known external pressures (usually zero) at the two ends of the bearing. Considering the governing differential equation as

$$L(P) = Q$$

we may use a weighted residual method by assuming an approximate solution

$$P^*(x) = a_1\phi_1(x) + \cdots + a_n\phi_n(x)$$

where the $\phi_i(x)$ are basis functions satisfying our boundary conditions, and define a residual error term.

$$R = L(P^*) - Q$$

Although we cannot force this term to vanish we can force a weighted integral, over the solution domain, of the residual to vanish. That is, the integral over the solution domain (Ω) of the product of the residual term and some weighting function is set equal to zero.

$$I = \int_\Omega w_i(x) R d\Omega = 0. \qquad i = 1, \ldots, n$$

In our above example we have

$$I = \int_0^L w_i(x) \left[\frac{d}{dx} \left(\frac{h^3}{6v} \frac{dP^*}{dx} \right) + \frac{d}{dx}(Uh) \right] dx = 0. \qquad i = 1, \ldots, n$$

which after integration by parts gives

$$I = \left[w_i(x) \{ \frac{h^3}{6v} \frac{dP^*}{dx} + Uh \} \right]_0^L - \int_0^L \frac{dw_i(x)}{dx} \left(\frac{h^3}{6v} \frac{dP^*}{dx} + Uh \right) dx = 0. \qquad i = 1, \ldots, n$$

Selecting $w_i(x) = \phi_i(x)$ allows us to simplify this to[12]

$$I = \int_0^L \frac{d\phi_i(x)}{dx} \left(\frac{h^3}{6v} \frac{dP^*}{dx} + Uh \right) dx. \qquad i = 1, \ldots, n$$

writing the basis functions as a vector $[\phi_1 \ldots \phi_n]$ and P^* as a dot product

$$P^* = [\phi_1 \ldots \phi_n][a_1 \ldots a_n]^T$$

defines our global stiffness matrix and force vector.

$$\frac{1}{6v} \int_0^L h^3 \begin{bmatrix} \phi_1' \\ \vdots \\ \phi_n' \end{bmatrix} [\phi_1' \ldots \phi_n'] dx \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} = -U \int_0^L h \begin{bmatrix} \phi_1' \\ \vdots \\ \phi_n' \end{bmatrix} dx$$

and defines the element stiffness matrices and force vectors as [1]

$$S^e = \frac{1}{6v} \int_{x_i}^{x_j} h^3 H_x^{eT} H_x^e dx$$

$$F^e = -U \int_{x_i}^{x_j} h H_x^{eT} dx$$

where the H^e are the element interpolation functions.

These can be written in Haskell using an *Expression* datatype as[13]

```
> element_sm_and_fv f g (x:_)
>  = ((h * h * h)/(6.0 * v)) * g' * f' :=: -u * h * g'
>     where
>            g' = D g (U 1)
>            f' = D f (U 1)
>            u = 20.0              -- velocity  = 20 ft/s
>            v = 0.002             -- viscosity = 0.002 lb s/ft^2
>            h = thickness x       -- thickness at point x
>            thickness x | x < 0.125 = 0.0250 -- ft
>                        | x > 0.125 = 0.0360 -- ft
```

[12] since $\forall i : \phi_i(0) = \phi_i(L) = 0$. Setting $w_i(x) = \phi_i(x)$ is known as the Galerkin[12] method.

[13] `D f (U n)` is taken to mean differentiate `f` with respect to the *nth* variable and $S :=: F$ is taken to mean that after integration S and F form the stiffness matrix and force vector for a general element.

Similarly for a 2-D Poisson problem taken from [3] (p.82)

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} - 2 = 0$$

whose Galerkin equations become:

$$\sum_{k,l=1}^{T} U_{kl} \int\int_R \left\{ \left(\frac{\partial\varphi_{kl}}{\partial x}\right)\left(\frac{\partial\varphi_{ij}}{\partial x}\right) + \left(\frac{\partial\varphi_{kl}}{\partial y}\right)\left(\frac{\partial\varphi_{ij}}{\partial y}\right) \right\} dxdy$$

$$+ \int\int_R 2\varphi_{i,j} dxdy = 0, \qquad (i, j = 1, \ldots, T)$$

where the $\varphi_{ij}(x, y)$ are the basis functions R is the region of interest divided into $(T+1)^2$ square elements, can be expressed as :

```
> element_sm_and_fv f g (x:y:_)
>  = (fx * gx + fy * gy :=: -2.0 * g)
>     where
>           gx = D g (U 1)
>           fx = D f (U 1)
>           gy = D g (U 2)
>           fy = D f (U 2)
```

3 Practical considerations

In this section the effect of non-strict semantics on our ability to express numerical algorithms will be considered along with the pragmatics of implementing lazy evaluation and the impact this has on the performance of arithmetic computations.

3.1 Advantages of non-strict semantics.

One of the advantages of non-strict languages is their ability to manipulate potentially infinite structures. Hence programs may be modularised further by de-coupling the process of producing a structure from the process consuming it [16]. An example of an infinite structure is an infinite sequence of approximations to a value defined by an iteration. If we take square roots as an explicit example we can define the method of producing the next value in a Newton iteration by the function

```
>           newton a x = (x + a/x) / 2
```

which, given a number a and an estimate x, produces a closer estimate of the square root of a. We may now use the higher order function *iterate* which produces an infinite list of values such that *iterate* $f\ a\ = [a, f(a), f^2(a), \ldots]$ when given the initial value a and function f.

```
>           iterate f a = a:iterate f (f a)
```

A function *within* returns the element of any sequence which has deemed to have converged on a value to within *eps* and can be defined as

```
>     within eps (a:b:rest) | abs (a-b) < eps = b
>                           | otherwise = within eps (b:rest)
```

and finally the square root

```
>          sqrt x eps a = within eps (iterate (newton a) x)
```

is constructed from reusable components.

Another advantage of non-strict languages is the ability to consume structures before they are complete and hence, define data structures in terms of themselves. For example, back substitution (the solving of a linear system $Ux = b$ in n unknowns where U is an upper-triangular matrix via solving the system for x_n, substituting this value into the system equations and then solving for x_{n-1}, etc.) can be defined as

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}, \qquad i = 1 \ldots n$$

and expressed in Haskell as

```
>    back_subst u b = x
>     where x = array bnds [(b!i - f i)/u!(i,i)|i<- range bnds]
>           f i = sum[u!(i,j)*x!j|j <- range (i+1,n)]
>           bnds@(1,n) = bounds b
```

where the vector **x** is defined in terms of itself.

However we may define back substitution very elegantly using a quadtree approach, i.e.

$$\text{bkSub} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{aligned} \text{where} \quad x_2 &= \text{bkSub}\, U_{22} b_2 \\ x_1 &= bkSub\, U_{11}(b_1 - U_{12} x_2) \end{aligned}$$

$$\text{bkSub}\, ub = b/u$$

which is pictorially expressive and terminates under eager evaluation.

The above approaches were tested with an LU decomposition method for matrices of various sizes. The program solved a densely stored linear system $Ax = b$ and output the scalar $x^T x$ so that timings reflected arithmetic work rather than I/O. The code was compiled and run under GHC 0.26[14] and HUGS 1.0 on a 70MHz 16Mb SPARC 5.

The timings show that, as expected, the cost of this type of recursive definition is expensive and greater efficiency is achieved when other methods are used.

[14] The command line options used were ghc -O2.

Timings for LU decomposition under GHC		
unknowns	Array	Quadtree
4	0.0	0.0
8	0.0	0.0
16	0.2	0.0
32	0.8	0.3
64	5.4	2.0
128	39.7	16.0
256	5:19.0	2:32.0

Reduction(cell) counts for LU decomposition under HUGS		
unknowns	Array	Quadtree
4	4913(9745)	564(1341)
8	23734(46468)	2766(6425)
16	140999(271807)	1652(38017)
32	951711(1818215)	113236(260582)
64	6942323(13192900)	832166(1927094)
128	52915119(100264791)	6351606(14789528)
256	*	*

3.2 Efficiency issues.

Lazy evaluation has advantages in areas such as heuristic searching, allowing elegant specifications and β-reduction is always valid. Unfortunately lazy languages also have the property of slowing down the execution of work which is necessary by delaying it if the compiler fails to determine that the work *is* necessary. This has a side-effect of creating a situation where the order of evaluation has little to do with the *loop-like* higher order functions and that closures persist longer than necessary.

3.3 Simulating lazy behaviour in a strict language

As mentioned in section 3.1, one of the advantages of lazy languages is the ability to manipulate potentially infinite structures. If written verbatim in a strict language this sort of program would result in non-termination. It is straightforward however to simulate this behaviour in a strict language (SML) and a lazy[15] list is defined as follows:

```
datatype 'a lazyList = Nil | Cons of 'a * (unit -> 'a lazyList);
```

together with our square root function in a style similar to before

```
fun newton n x = (x+n/x)/2.0;

fun iterate f a = Cons(a,fun()=>iterate f (f a));

fun within (eps:real) Cons(a1,as) =
     let val (Cons(a2,rest)) = as();
     in if
          (a1-a2) < eps
        then
          a2
        else
          within eps (Cons(a2,rest))
     end;

fun sqrt a eps n = within eps (iterate (newton n) a);
```

[15]Although we refer to this as a *lazy* list, this is a slight misnomer as no sharing occurs. A more accurate description would be a *call by name* list.

where we manually insert our own closures. In this way it can be seen that the modularity of the previous lazy program can be achieved within a strict framework. In the above example a list is represented by a value/function pair `(a,as)`, which can be viewed as being analogous to a list element together with a *pointer* or link to the next value/function pair in an imperative language.

3.4 Functional programming and arithmetic

When non-strict languages are compiled, the resulting code usually manipulates heap-allocated *box* numbers [21]. Thus in a naive implementation, numbers are always represented as a pointer to a heap-allocated object which may be an unevaluated closure or a *box* containing the number's value which has overwritten the closure. The consequence is that simple arithmetic operations which require a single machine instruction in a strict language require the following steps:

- fetch operands from their respective boxes
- perform the operation
- allocate a new box
- place the result in the box

Clearly it is more efficient to work with the bit patterns that reside in the boxes (*the unboxed values*) than with the boxes and contents (*the boxed values*). *With reference to large-scale scientific computing, retaining this efficiency is not only desirable but essential.*

Some compilers such as GHC allow the programmer access to these unboxed representations but at the expense of losing all higher order functions and polymorphism! This effect is reduced however by the use of specialisation pragmas in the code. Other features, such as strictness annotations, also exist in functional languages (Haskell 1.3 definition) and these features up to a point help to claw back efficiency. For instance the performance of level 1 BLAS operations are improved eight-fold when written in unboxed core Haskell [2].

4 Lessons learned

Since the performance of Haskell is so drastically improved by unboxing it is believed that success in using functional programming languages for scientific computing relies on the use of strict languages. When we unbox a value we impose a level of strictness. The benefit of carrying out unboxing decreases as the complexity of the problem increases, and the act of unboxing a datatype removes the ability to use higher order functions such as *map*, makes code monomorphic, and obscures the algorithm. As mentioned in previous sections, it is possible to simulate the manipulation of infinite structures in a strict language and it is also possible to express recursively-defined array problems in terms of matrix/vector operations. Because of these observations it is proposed that a strict language should be used for this type of application, where either lazy behaviour is explicitly coerced in the style of section 3.3 if necessary, or specialised language constructs are used for common patterns of computation. This approach has the benefits of not having to unbox arithmetic expressions, an improved execution, and an improved ability to reason about the runtime behaviour of programs.

This view is echoed by experience with the SISAL[7] language, where efficiency comparable with hand-written C/FORTRAN for serial and parallel machines (mostly shared memory) has been achieved. Although a functional language, SISAL is quite primitive by modern language standards and is perhaps more accurately described as a single assignment language with Pascal-like syntax. SISAL is monomorphic, explicitly typed and lacking many of the modern features of functional languages. The newer SISAL 90 [13] addresses some of these problems, such as the first order nature of SISAL 1.2, but concentrates mainly on improving array operations (a feature which SISAL 1.2 is strong in anyway).

Another important area of note when discussing the use of languages for numerical computation is the use of overloading. The typeclass system offered by Haskell is not powerful enough to support all the forms of overloading natural to the field of scientific computing. The main problem with Haskell type classes is that they only support single parameter parameterisation, and natural overloadings such as scalar multiplication and matrix-vector multiplication are disallowed. Also, the method in which overloaded functions are implemented (via dictionaries and implicit function parameters) reduces their appeal since there is an efficiency overhead incurred by using overloaded functions.

With these issues in mind a functional language *Functional Scientific Computing* (FSC) is being developed specifically for the purpose of numerical programming in the style of Haskell, together with features found in other functional languages.

4.1 FSC design issues

The main features of FSC are as follows:

- Multi-paramter typeclasses.
- An offside rule *à la* Haskell and a terse syntax
- Specialisation *à la* NESL [4]
- Mixed mode arithmetic
- 'Where' constructs and λ expressions
- Guarded expressions
- Support for ADTs and pattern matching
- Parametric polymorphism (*textual*)
- Allow uppercase characters to denote matrix identifiers

- Encourage literate programming *à la* Miranda / Haskell
- Real arrays *à la* SISAL with array comprehensions with dot/cross product loop fusing
- Strict semantics and eager evaluation
- Implicit typing
- Partial application and sections
- Iterations *à la* SISAL

The type system for this langauge will base its implementation around automatic specialisation of polymorphism and overloading to prevent the efficiency loss suffered by boxing data so that it can be treated uniformly. The obvious restriction this places on the language is that first order truly polymorphic values are disallowed. It is felt that the application domain reduces the need for this feature and this restriction is justified.

4.2 Multi-parameter type classes

The version of Haskell type classes proposed are similar to those of watML[19] in that multi-parameter templates are allowed in class definitions. For example, we can define a class `Plus (a,b,c)` which contains the function `(+)::a -> b -> c` and then we are able to define instances such as `(INT,REAL,REAL)` to allow true mixed-mode arithmetic.

As mentioned earlier, we do not wish to lose efficiency via this use of overloading and so the method that has been chosen to implement FSC polymorphism is what has been described as *textual polymorphism* [22], where a function has many types only at the source level. The advantages [22] of this technique are that

- it can produce optimum code for each application of the polymorphic procedure,
- it can implement non-uniform representations of data, and
- it can implement *ad hoc* polymorphism as well as universal polymorphism.

The disadvantages are that it cannot support true first class polymorphic values, and that there is a potential exponential explosion in the number of specialised polymorphic forms. In each case we believe these drawbacks will not prove to be detrimental due to the type of programs written within our chosen problem domain[16].

The remaining sections describe the motivation behind the use of this type system, together with the presentation of the algorithm to implement it.

[16] To an extent this view is backed up in [17] where the specialisation of type classes actually reduced the final executable's size.

4.3 Use and motivation

The simplest usage of multi-parameter overloading is as follows, where we provide overloadings for `+` so that it may work on integers and floats.

```
class Plus (a,b,c) where
 (+) a -> b -> c

instance Plus(INT,INT,INT) where
 (+) = primintintplus
instance Plus(INT,Float,Float) where
 (+) = primintfloatplus
...
```

Then in the main expression

```
main = 2 + 3 + 1.5
```

the type of `main` is `[+: INT -> INT -> a][+:a -> Float -> b] b`. A unique satisfying substitution exists for the above constraints and so the overloading is resolved. This is completely different to Haskell where implicit `fromInteger` and defaults are used.

A slightly more complicated example is

```
> f x y z = x + y + z

> main = f 1 2 3
```

where the function `f` exports 2 constraints and the types of `main` and `f` are:

```
f    :: [+:  a   -> b   -> c][+:  c -> d   -> e] a -> b -> d -> e
main :: [+:  INT -> INT -> a][+:  a -> INT -> b] b
```

the function `f` can be seen to form an abstraction requiring 5 type variables [a,b,c,d,e].

The motivation behind the use of this system can be understood by considering the following example:

4.4 Example

Consider the linear system $Ax = b$. It is possible to solve this system in various ways, one being the Conjugate Gradient Method (CGM), whose iteration can be described in a Haskell-like language as:

```
> dot x y = sum (zipWith (*) x y)
```

```
> sum [] = fromInteger 0
> sum [x] = x
> sum (x:xs) = x + sum x

> conjugate_gradient_iteration A (x,p,r) = (x',p',r')
>  where x'    = x + alpha * p
>        p'    = r' + beta * p
>        r'    = r - alpha * q
>        alpha = rr / pq
>        beta  = (alpha * qq) / pq -1
>        pq    = p `dot` q
>        rr    = r `dot` r
>        qq    = q `dot` q
>        q     = A * p
```

Often we do not wish the matrix A to be formed explicitly as this is sometimes very computationaly intensive. In such situations the CGM iteration above can be thought of as forming an abstraction. If we define an overloading of `(*)` such as function application `(*)::(a -> b) -> (a -> b)`, we would not explicitly form the matrix A although our iteration algorithm would retain its elegance.

4.5 Description of the type inference algorithm

Informally the typechecker runs through a Hindley-Milner inference[8] phase and collects type constraints. These constraints are then resolved. The approach taken here is very similar to [19], the difference being since we have made the decision to specialise our polymorphic forms and overloaded identifiers we may use a slightly more liberal resolution algorithm than [19].
As an example, consider

```
double x = x + x

main = double 2
```

After type inference `double` and `main` are typed as

```
double :: [+: a  ->  a  -> b]  a  -> b
main   :: [+:INT -> INT -> a] INT -> a.
```

An intermediate code is then generated resembling an extended second order λ-calculus lifted into recursive supercombinator definitions.

```
double ::([+: a -> a -> b] a -> b)[a,b](x:a RETURNS b)
         = (+)[a,a,b](x,x)

main   ::([+:INT -> INT -> a] a)[a](RETURNS a)
         = double[INT,a](2)
```

A unique solution for the constraints attributed to `main` ($[+ : INT-> INT-> \alpha]$) is then be searched for. If found, the code is specialised[17]. The formal syntax for expressions and types is given as follows

Formal syntax of expressions			
Identifier	x		
Expression	e	$::=$	x
		$\mid$	$e_1\ e_2$
		$\mid$	$\lambda x.e$
		$\mid$	let $x = e_0$ in e_1
		$\mid$	if e_0 then e_1 else e_2

Formal syntax of types			
Type Variables	α		
Type Constructors	χ		
Type Constraints	$\mathcal{C}$	$::=$	$\{x : \tau\}$
		$\mid$	$\mathcal{C}_0 \cup \mathcal{C}_1$
Types	τ	$::=$	α
		$\mid$	$\tau_0 \rightarrow \tau_1$
		$\mid$	$\chi(\tau_1, \ldots, \tau_n)$
		$\mid$	$\mathcal{C}_1, \ldots, \mathcal{C}_n\ \tau$
Type Schemes	σ	$::=$	$\forall \alpha.\sigma$
		$\mid$	τ

with an extension to Robinson's unification algorithm [23] summarised by:

Unification	
$unify_{\theta\mathcal{C}^*}(\mathcal{C}_1^*\ \tau_1, \mathcal{C}_2^*\ \tau_2)$	succeeds if $unify_{\theta,(\mathcal{C}^* \cup \mathcal{C}_1^* \cup \mathcal{C}_2^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,\mathcal{C}^*}(\tau_1, \mathcal{C}_1^*\ \tau_2)$	succeeds if $unify_{\theta,(\mathcal{C}^* \cup \mathcal{C}_1^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,\mathcal{C}^*}(\mathcal{C}_1^*\ \tau_1, \tau_2)$	succeeds if $unify_{\theta,(\mathcal{C}^* \cup \mathcal{C}_2^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,\mathcal{C}^*}(v_1, v_2)$	succeeds; v_1 and v_2 are bound to the same variable in θ.
$unify_{\theta,\mathcal{C}^*}(\chi_1[s_{11}, \ldots, s_{1m}], \chi_2[s_{21}, \ldots, s_{2n}])$	succeeds if $\chi_1 = \chi_2 \wedge m = n$ $\wedge unify_{\theta,\mathcal{C}*}(s_{11}, s_{21}) \wedge \ldots \wedge unify_{\theta,\mathcal{C}^*}(s_{1m}, s_{2n})$, and fails otherwise.
$unify_{\theta,\mathcal{C}^*}(v, \chi[s_1, \ldots, s_n])$	succeeds binding v to $\chi[s_1, \ldots, s_n]$ in θ if v does not occur in $s_1 \ldots s_n$, and fails otherwise
$unify_{\theta,\mathcal{C}^*}(\chi[s_1, \ldots, s_n], v)$	succeeds similarly

Here χ is a constructor, v is a variable, $\mathcal{C}$ is a constraint, and the s_i are arbitrary terms. $unify_{\theta,\mathcal{C}^*}(x, y)$ computes the most general unifier of x and y under θ and $\mathcal{C}^*$, together with the set of constraints that must be satisfied for x and y to be unified.

[17]Provided the type of main is now monomorphic.

The typechecking rules are given as follows, where overloaded instances are checked against their templates and all resulting constraints are collected.

Type rules		
TAUT	$A.x : \tau \vdash x : \tau$	
COND	$\dfrac{A \vdash e_0 : \mathcal{C}_0^* bool \quad A \vdash e_1 : \mathcal{C}_1^* \tau \quad A \vdash e_2 : \mathcal{C}_2^* \tau}{A \vdash (if\ e_0\ then\ e_1\ else\ e_2) : (\mathcal{C}_0^* \cup \mathcal{C}_1^* \cup \mathcal{C}_2^*) \tau}$	
ABS	$\dfrac{A.x : \sigma \vdash e : \mathcal{C}^* \tau}{A \vdash (\lambda x.e) : \mathcal{C}^* \sigma \rightarrow \tau}$	
LET	$\dfrac{A \vdash e_0 : \sigma \quad A.x : \sigma \vdash e_1 : \tau}{A \vdash (let\ x = e_0\ in\ e_1) : \tau}$	
GEN	$\dfrac{A \vdash e : \tau}{A \vdash e : \forall \alpha . \tau}$	(α not free in A)
SPEC	$\dfrac{A \vdash a : \forall \alpha . \tau}{A \vdash e : [\sigma / \alpha]\, \tau}$	
COMB	$\dfrac{A \vdash e_0 : \mathcal{C}_0^* \sigma \rightarrow \tau \quad A \vdash e_1 : \mathcal{C}_1^* \sigma}{A \vdash (e_0\ e_1) : (\mathcal{C}_0^* \cup \mathcal{C}_1^*) \tau}$	

4.6 Overloading resolution

The overloading resolution phase attempts to find an unique satisfying substitution of type instances over the type constraints resulting from type inference.

Overloading resolution

$$RESOLVE(I, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}) = SEARCH(I, MATCHES(I, \{\}, \mathcal{C}_1), \{\mathcal{C}_2, \ldots, \mathcal{C}_n\})$$

$$MATCHES(I, \theta, \mathcal{C})$$
$$= \{unify_{\theta,\{\}}(i_{type}, \mathcal{C}_{type}) \mid i \in I \wedge unify_{\theta,\{\}}(i_{type}, \mathcal{C}_{type}) \neq \perp \wedge \mathcal{C}_{name} = i_{name}\}$$

$$SEARCH(I, P, \mathcal{C}^*) = \begin{cases} \perp, & \text{if } P = \{\} \\ OUT(\{DERIVATION(p_i, I, \mathcal{C}^*) | p_i \in P\}), & \text{otherwise} \end{cases}$$

$$OUT(\{x\}) = x$$
$$OUT(\{\}) = \perp$$
$$OUT(\{x_1, \ldots, x_n\}) = \perp$$

$$DERIVATION((\theta, \{\}), I, \{\}) = (\theta, \{\})$$
$$DERIVATION((\theta, \{\}), I, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\})$$
$$= SEARCH(I, MATCHES(I, \theta, \theta(\mathcal{C}_1)), \{\mathcal{C}_2, \ldots, \mathcal{C}_n\})$$
$$DERIVATION((\theta, \mathcal{C}_1^*), I, \mathcal{C}_2^*) = DERIVATION((\theta, \{\}, I, \mathcal{C}_1^* \cup \mathcal{C}_2^*)$$

where I is the set of instances, θ are substitutions, $\mathcal{C}$ are constraints and the P are possible matches. In the above X^* denotes a set of X's. All constraints are taken to be name-type pairs; for example if $\mathcal{C} = \{+ : INT \rightarrow INT \rightarrow INT\}$ then $\mathcal{C}_{name} = +$ and $\mathcal{C}_{type} = INT \rightarrow INT \rightarrow INT$.

In the FSC implementation the typechecker is more detailed than suggested by the above, allowing such features as pattern matching and case statements.

5 Further work

In the previous sections the usefulness of a functional style for numerical computation was shown by displaying the terseness and elegance with which algorithms may be specified. Desirable features in a realistic language to achieve these ends were also mentioned together with an algorithm to implement the proposed overloading strategy. This study is now directed to the area of evaluating these ideas in a real situation via the completion of the FSC compiler, extending the type class structure to encompass areas such as array manipulation and basic IO. This system will then be used to build a finite element system and, via a period of iterative refinement, a final definition for the language will emerge. To retain efficiency this compiler will translate FSC code to SISAL and make use of SISALs already efficient implementation.

References

[1] J.E. Akin. *Application and Implementation of Finite Element Methods*. Academic Press, 1982.

[2] Chris Angus. An introduction to functional programming and its application to numerical software development. The University of Newcastle upon Tyne, Report prepared for thesis committee meeting dated June 1995.

[3] Mitchell A.R. and Wait R. *Finite Element Analysis and Applications*. Wiley, 1985.

[4] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[5] James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer. The construction of numerical mathematical software for the AMT DAP by program transformation. In *Proceedings of CONPAR VAPP V, LNCS 634*, pages 761–767. Springer-Verlag, September 1992.

[6] J.M. Boyle and T.J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1):81–126, January 1992.

[7] David C. Cann. *SISAL 1.2: A Brief Introduction and Tutorial*. Computing Research Group, Lawrence Livermore National Laboratory http::/sisal.llnl.gov/pub/sisal/, 1.2 edition.

[8] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2), 1987.

[9] J.C. Diaz and K. Shenoi. Domain decomposition and Schur complement approaches to decoupling the well equations in reservoir simulation. *SIAM Journal on Scientific Computing*, 16(1):29–39, 1995.

[10] Iain S. Duff. Data structures, algorithms and software for sparse matrices. Technical Report CSS158, Computer Science and Systems Division, AERE Harwell, Oxfordshire, OX11 ORA, 1984.

[11] J.F. Dwyer. Functional programming for finite elements. *Computers and Structures*, 33(6):1343–1348, 1989.

[12] Graeme Fairweather. *Finite Element Galerkin Methods for Differential Equations.* MARCEL DEKKER INC, 1978.

[13] John Feo, Patrick Miller, Stephen Skedzielewski, Scott Denton, and Cindy Solomon. Sisal 90. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing, ftp://sisal.llnl.gov/pub/hpfc/hpfc95.html*, pages 35–47, April 1995.

[14] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Some issues in a functional implementation of a finite element algorithm. In *FPCA '93. Conference on Functional Programming Language and Computer Architecture*. ACM Press, June 1993.

[15] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[16] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[17] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[18] Junxian Liu, Paul Kelly, and Stuart Cox. Functional programming for finite element analysis. Technical report, Department of Computing, Imperial College, 1993.

[19] John Ophel and Dominic Duggan. Multi-parameter parametric overloading. Submitted for publication , http://nuada.uwaterloo.ca/ dduggan/papers/, 1995.

[20] Rex L. Page and Brian D. Moe. Experience with a large scientific application in a functional language. In *FPCA '93. Conference on Functional Programming Language and Computer Architecture*. ACM Press, June 1993.

[21] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*. ACM Press, Sept 91.

[22] Morrison R., Dearle A., Connor R.C.H., and Brown A.L. An ad hoc approach to the implementation of polymorphism. *Transactions on Programming Languages and Systems*, 13(3):342–372, 1991.

[23] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 2(3):23–49, January 1965.

[24] Roger L. Wainwright and Marian Sexton. A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming*, 2(1):61–72, January 92.

[25] David S. Wise. Matrix algorithms using quadtrees. Technical Report 357, Computer Science Department, Indiana University, 1992.

[26] D.S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science No. 274. Springer-Verlag, 1987.

[27] D.S. Wise. Undulant block pivoting and integer-preserving matrix inversion. Technical Report 418, Computer Science Department, Indiana University, 1994.

A Further Reading

In this section we conclude by introducing some of the previous work in this area and relating this to the material presented.

There have been many attempts to apply functional programming to numerical analysis over the last few years.

- D. S. Wise [26, 25, 27] uses quadtrees to define matrix / vector algorithms in a 'divide and conquer' fashion. The quadtree is used in the same manner as it is in [10], where it is called a hypermatrix.

- This approach is also used in the work of Wainwright & Sexton [24] where conjugate gradient and SOR methods are coded in Miranda for various sparse matrix representations. In their report they conclude that the quadtree is more suitable for CG-type algorithms than SOR, although their row-based formulation of SOR rather than a block-based formulation would explain this.

- Page & Moe [20] define an executable specification of a reservoir model as a Miranda program embedded in a 200 page LaTeX document. This code, although executable, is not meant to be production quality and was later hand-coded into an imperative language. One of the interesting points arising from this study is the fact that compared to previous software the amount of documentation in the form of commentary approximately doubled, and the number of flagged lines of code approximately halved when a literate style of programming was employed. It is because of this that one of the aims of this proposal is to encourage a more literate style of programming.

- As part of the FLARE[18][14] project a finite element program simulating fluid dynamics was written in Haskell. This concurs with [24] in that quadtree-type data structures were found to be faster than list based structures for sparse linear system solvers. The study concludes that compared to procedural programming, functional programming is more expressive and more easily maintained with most errors being detected at the compilation stage. Lazy evaluation did not prove advantageous with evaluation having to be forced.

- Other very simple FE codes have been written in functional languages such as Miranda [18] (simple beam joint code), and SASL [11] (problem-specific elementary 1D PDE code), with emphasis mainly on the elegance of the implementation.

- Boyle, Fitzpatrick & Harmer [6, 5] use program transformations to translate code expressed as pure LISP and ML to FORTRAN. The approach proposed in this paper differs from this in that it aims to bring together many of the features found in various languages to support the construction of large scale numerical software, rather than writing transformational compilers for existing languages.

- Diaz & Shenoi [9] use Miranda to investigate decoupling systems of well equations in a Schur complement approach to domain decomposition using the number of reductions to measure work complexity. In the study they comment that *"functional syntax allows direct correspondence of functional code with the mathematical equations, which makes the code easy to develop, write, read and modify."*

[18] Functional Languages Applied to Real world Exemplars.