Some Lattice-based Scientific Problems, Expressed in Haskell

D.B. Carpenter

Department of Electronics and Computer Science,
Southampton University,
Southampton, S09 5NH

DRAFT

Abstract

The paper explores the application of a lazy functional language, Haskell, to a series of grid-based scientific problems—solution of the Poisson equation and Monte Carlo simulation of two theoretical models from statistical and particle physics. The implementations introduce certain abstractions of grid topology, making extensive use of the polymorphic features of Haskell. Updating is expressed naturally through use of infinite lists, exploiting the laziness of the language. Evolution of systems is represented by arrays of interacting streams.

1 Introduction

Lazy functional languages have not, to date, made an enormous impact in scientific computing. Partly, this has to do with performance—codes written in these languages often run an order of magnitude slower than imperative codes, which limits their appeal for number-crunching production codes[1]. Apart from that, there is a suspicion that programming without assignments or side-effects is difficult, or restrictive. Evidently, this is not the opinion of functional programmers—but perhaps there are things in scientific computation really cannot be *done* easily without assignments.

This article gives detailed Haskell implementations of three "scientific" algorithms. In order of increasing complexity they are: solution of a partial differential equation by the relaxation method; Monte Carlo simulation of a statistical mechanics system; and a similar simulation of a lattice gauge theory. These problems are taken from theoretical physics—a strongly mathematical discipline. Functional programming also has a very mathematical feel to it. So it should not be too surprising that these problems actually come out rather neatly in a functional language. (Other examples of scientific functional programming can be found in [1, 2, 3]).

Haskell is a relatively new language, put forward as an international standard for functional programming [4, 5, 6]. It is a lazy, pure functional language. Beyond that, the choice of Haskell as a mode of expression in this article is fairly incidental—any lazy functional language would probably have done as well. Having chosen Haskell, however, free use has been made of its more innovative features, including the array and class systems (the latter for achieving overloading, or *ad hoc* polymorphism, in the array manipulation functions we introduce). A fair understanding of Haskell is probably a

[1] It seems that this caveat on functional programming no-longer holds for some non-lazy functional language—see [3].

prerequisite for reading this article. By and large, the "prelude" functions of Haskell will be used without much comment, so a Haskell reference may be necessary.

Full understanding of the problems and algorithms described will require a certain amount of mathematical knowledge—some familiarity with simple partial differential equations, complex numbers, and a rough grasp of probability distributions. Section 2 describes the systems involved and the associated algorithms. The problems are all based on lattices (i.e., grids). They make important use of arrays.

Rather than work directly with standard Haskell arrays, it proved convenient to introduce a kind of generalised array, with more geometric significance. Section 3 describes the approach taken to lattice geometry, and the arrays defined on these lattices. The appendices give details of a possible Haskell implementation of the generalised arrays and operations on them.

The implementations of the main algorithms are given in section 4. The paradigm used in each case is one of interacting parallel processes, modelled by lazy lists (streams) representing the output of the processes. Partly this choice can be attributed to the source of funding for this work—the FAST project aims to provide a parallel implementation of annotated functional programs written in essentially this style [7]. Apart from that, the style comes naturally and is quite efficient in a lazy language, especially for algorithms with some "locality of reference".

Finally, some conclusions are collected in section 5.

We chose to handle the lattices in a rather abstract way. The result is that the geometry factors out from the problems in an attractive way—the implementations in section 4 make almost no reference to the details of this geometry. Abstraction certainly has dividends, but understanding it also requires some investment of effort. Needless to say, Haskell does not *enforce* this level of abstraction. It does enable it.

2 Three problems

The choice of problems here reflects an early motivation for this work, which was to implement some QCD codes in a functional language. *Quantum Chromodynamics* is the theory of the strong nuclear force. It has a reputation as a "grand challenge" numerical problem. In the present work (as it turned out) the emphasis is more on elegant specification of the underlying algorithms than high performance.

Through a series of mathematical tricks [8, 9, 10, 11] QCD can be expressed as a *lattice gauge theory*, in which form it is amenable to Monte Carlo simulation. In this article, lattice gauge theory will be introduced from a more algorithmic point of view, assuming no particular knowledge of the physical systems involved. First two simpler problems which share some of its features are introduced.

2.1 The 2-dimensional Poisson equation

A fundamental problem in electrostatics is calculation the electric field produced by a given distribution of charges. The electrostatic potential is obtained by solving the associated Poisson equation. For simplicity we describe the two-dimensional case—a third dimension introduces nothing essentially new. The given charge distribution is a function $c(x, y)$ of the coordinates. The electrostatic potential $u(x, y)$ is the solution of the equation

$$(\boldsymbol{\nabla}^2 u)(x, y) = -c(x, y). \tag{1}$$

Having solved for u, the electric field is the gradient $\boldsymbol{F}(x, y) = -(\boldsymbol{\nabla} u)(x, y)$.

The most simple-minded approach to solving (1), is to replace it by the finite difference approximation

$$u(x+1,y)+u(x-1,y)+u(x,y+1)+u(x,y-1)-4u(x,y)=-c(x,y). \qquad (2)$$

The coordinates x, y are now integers—assume that they and c have been rescaled to eliminate appearance of the grid spacing parameter. Rearranging (2) as

$$u(x,y)=\frac{1}{4}(\text{sum neighbouring } u \text{ values} + c(x,y)), \qquad (3)$$

immediately suggests the simplest iterative algorithm for solving this system. In the *relaxation* algorithm, starting from some initial guess for u, each iteration replaces the value of u at a point by the average value of its neighbours (plus an offset determined by the local charge). Variants differ in whether the u values inserted in the RHS of (3) are those from the previous sweep of the grid (Jacobi relaxation) or whether they are the most recent values, which may include some values already updated in the current sweep (Gauss-Siedel relaxation). The last version has further variations depending on the order in which the grid is traversed—a popular scheme is "red-black relaxation" in which all even sites are visited first, then all odd sites. Straight relaxation is a slow algorithm, but it does form the core update of more powerful methods [12].

2.2 The XY Model

Statistical physics is largely concerned with the study of phase transitions in complex systems. Many simplified mathematical models have been introduced which exhibit transitions similar to those of physical systems. The *XY model* is an example [13]. This kind of model can be motivated by referring back to the Poisson equation. Equations (1) or (2) can be obtained from variational principles. Solving the system (2) is equivalent to minimising the functional

$$E(u)=\sum_{x,y}\frac{1}{2}\left((u(x+1,y)-u(x,y))^2+(u(x,y+1)-u(x,y))^2\right)-c(x,y)u(x,y).$$

$E(u)$ is the electrostatic energy of the configuration u; the lowest energy state is the one "preferred by nature".

In statistical physics, states with higher energy are also taken into account. The aim of a simulation is to generate a sequence of configurations of the system, in which the frequency of appearance of a particular configuration u with energy $E(u)$ is weighted by the Boltzmann probability distribution:

$$P(u)\propto\exp(-E(u)/\theta).$$

Here, θ is the temperature of the system.

In the XY model $u(x,y)$ is a unit-length two-dimensional vector rather than a simple real number. This vector can be represented as a complex number with unit magnitude. The energy function can then be written as

$$E(u)=-\Re\sum_{x,y}u^*(x,y)u(x+1,y)+u^*(x,y)u(x,y+1),$$

where u^* means complex conjugate of u, and $\Re$ means real part.

Algorithms for generating sequences of configurations with Boltzmann distribution for this type of system resemble relaxation. They are iterative algithms, involving repeated

sweeps across the grid, and the update of a single site normally only needs knowledge of the current values at neighbouring sites.

In the *Metropolis* algorithm [14] the update of a site is as follows: choose a candidate new value for the local u by some pseudo-random procedure[2]. Calculate the change in energy, ΔE, which would be produced by replacing the old value of u by the candidate new one (this depends on the state of the neighbouring sites). If ΔE is negative, accept the candidate update. If ΔE is positive, accept the candidate with probability $exp(-\Delta E)$. If the candidate is rejected, retain the original value of u.[3]

2.3 Lattice gauge theory

This section describes the so-called $U(1)$ lattice gauge theory[4]. The main difference

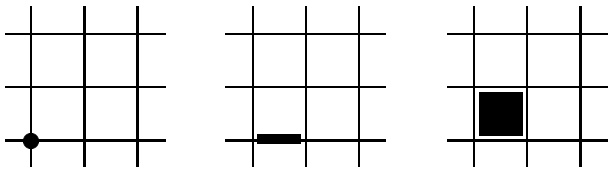


Figure 1: Some elementary cells of a lattice.

between $U(1)$ gauge theory and the XY model is geometric. In the XY model the variables u are associated with the *sites* of the lattice. The energy function (2.2) is a sum over the *links* of the lattice—it involves multiplying together the values of the variables at the ends of each link.

In gauge theory the variables are associated with the links of the lattice, and the energy function is computed on the next-higher dimension cells of the lattice—the elementary squares, or *plaquettes* (figure 1).

We can associate a group element (a unit-magnitude complex number) U_P with any path P in the lattice by multiplying together the link variables along the path. The energy function is

$$E(u) = - \sum_{P \in plaquettes} \Re(U_P)$$

where *plaquettes* is the set of elementary plaquette-paths in the lattice.[5]

The Metropolis algorithm can be carried over from the XY model. The local update is identical. The difference lies in how the neighbouring values which effect the local contribution to the energy are collected. In the XY model, the local contribution to the

[2]The procedure should give equal probabilities for rotating the vector by $+\delta$ and $-\delta$, for all angles δ. Also it should be "ergodic"—by combining some sequence of allowed updates it must be possible to get from any state to any other

[3]The proof of this algorithm is omitted. The crux is to show that, when the probability of a system state is given by the Boltzmann formula, the expected frequency of transitions (produced by the algorithm) from any state A to any other state B is the same as that from B to A ("detailed balance") so that the Boltzmann distribution represents equilibrium.

[4]$U(1)$ is the group of unit magnitude complex numbers, with complex multiplication as the product. This is the "gauge group" of *Quantum Electrodynamics* rather than QCD, but the two theories are closely related.

[5]In the XY model energy, one of the factors in the link product has to by conjugated. In gauge theory, variables associated with links which have negative incidence on the plaquette are conjugated. This will be illustrated more fully in the next two sections.

energy was determined by the values of site variables at neighbouring sites. In gauge theory it is determined by link variables on the "staples" illustrated (in the 3-dimensional case) in figure 2. For real physics, the lattice should be four-dimensional. It is usually taken as a 4d cubic grid, but the theory has sometimes been formulated on other lattice geometries; the implementations given here will be largely independent of the details of this geometry.

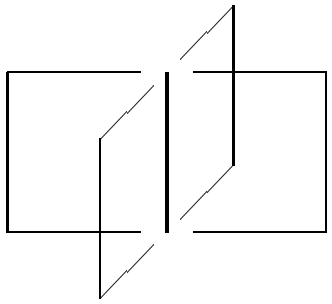


Figure 2: Surrounding links, contributing to the energy of the central one.

3 Generalised Arrays

A straightforward way to store the values of, for example, the potential function u in section 2.1 is in a Haskell array. This works well, provided the lattice is a regular grid and its extent is rectangular. It works less well in the case of lattice gauge theory (section 2.3). There the variables naturally live on the links of the lattice. One can still use an array indexed by lattice sites, with an extra dimension taking values in $1, \ldots, d$ to access the d links rooted at each site (d is the dimension of the lattice). This is quite clumsy.

We can think of the sites of a lattice as the 0-dimensional cells of the lattice, the links as the 1-dimensional cells, the plaquettes as 2-dimensional cells, and so on. Occurrence of variables which naturally live on cells with dimension higher than 0 is more common than one might initially suppose. Even in the Poisson equation the electric field, which is the gradient of the potential, sits most naturally on the links of the lattice. The field in the x-direction is associated with links in that direction; its value is just the difference of the potential at the two ends of its link. In the XY model there occur certain "vortex configurations" centred on the plaquettes of the lattice—one can define a function which counts the density of vortices in a general configuration, and this function sits naturally on the plaquettes of the lattice. As explained in section 2.3, the energy function of a gauge theory also "lives" on the plaquettes of a lattice.

This suggests that it may be useful to generalise the usual model of an array so that it is defined on an arbitrary (not necessarily rectangular) lattice, and that the index space of a particular array can be cells of that lattice of arbitrary (fixed) dimension. This generalised kind of array will be called a "form"[6].

The lattice itself is no longer thought of just as a set of connected sites, but as a "cell complex" containing cells of all dimensionality up to that of the lattice itself.

3.1 Cells, lattices and forms

This section assumes some knowledge of the Haskell class system. The standard prelude defines, for example, the class `Ix`. All types which can be used as index types for an array

[6] This terminology is motivated by an analogy between this kind of array and the p-forms of differential geometry [15].

belong to this class. A type specification involving type variables may be preceded by an assertion that some of those variables belong to class `Ix`. Array operations like the array subscripting operator `(!)` (which, by the way, is an ordinary infix function—not special syntax) are *overloaded* to apply to all instances of `Ix`.

We need a Haskell class `Cell` which parallels the index class `Ix` associated with ordinary arrays. Two type constructors `Lattice` and `Form` go with this class. They construct the following types[7]:

```
(Cell a) => Lattice a
(Cell a) => Form a b
```

The latter is directly analogous to the ordinary Haskell `Array` constructor—`Form a b` is the type of a form with cell-type `a` and elements of type `b`. `Lattice a` is the type of a data structure describing a lattice with cells of type `a`. It takes over the role of the bounds pair for an ordinary array.

To make this more concrete, a possible instance of the `Cell` class, suitable for handling 2-dimensional rectangular lattices, is

```
data Cell2d = Cell2d0 (Int, Int)     |
              Cell2d1 (Int, Int) Int |
              Cell2d2 (Int, Int)
```

which can represent a site, link or plaquette of the lattice[8]. The numeric fields are the cell coordinates, and in the 1-cell case, the link direction. The cell type alone does not define the lattice shape. We need at least to know the size of a rectangular lattice, and besides that, the "edge conditions" have to be specified—the lattice may have a boundary at the edges, or be connected cyclically, etc. We will assume the existence of a predefined function such as

```
torus2d :: Int -> Int -> Lattice Cell2d
```

which takes the width and height of the lattice as parameters, and returns an instance of an abstract data type representing (in this example, let's say) a periodic lattice of the specified size. A possible implementation of `torus2d` is given in appendix C. Now, we can start building forms. The function analogous to the ordinary Haskell `array` is

```
form :: (Cell a) => Lattice a -> Int -> [Assoc a b] -> Form a b
```

This constructs a form from an "association list". The first parameter is the lattice on which the form is to be defined; the second is the "rank", r, of the form—that is, the dimension of the cells on which it is defined; the third is an association list completely analogous to the corresponding argument of `array`—a list of pairs associating each cell of dimension r in the lattice with a value. As a mundane example, the expression

```
form l 1 [c := 0.0 | c <- cells l 1]
```

is 1-form with value 0.0 on each link of lattice `l`.

```
cells :: (Cell a) => Lattice a -> Int -> [a]
```

is analogous to `range`, returning a list of all cells of a specified dimension in the given lattice. Ultimately, the following operations on forms will also be needed

[7] The `(Cell a) =>` prefix asserts that the type `a` must be an instance of class `Cell`.

[8] `Form` and `Lattice` types will be treated, in this section at least, as abstract types. There seems to be little advantage in treating the cell types themselves as abstract—it is convenient to have access to the constructors, for referring to an element of the lattice at known coordinates

```
accumForm :: (Cell a) => Lattice a -> Int -> [Assoc a b] -> Form a [b]
lattice   :: (Cell a) => Form a b -> Lattice a
rank      :: (Cell a) => Form a b -> Int
(!#)      :: (Cell a) => Form a b -> a -> b

(//#)     :: (Cell a) => Form a b -> [Assoc a b] -> Form a b

listForm  :: (Cell a) => (Lattice a) -> Int -> [b] -> Form a b

fassocs   :: (Cell a) => Form a b -> [Assoc a b]
findices  :: (Cell a) => Form a b -> [a]
felems    :: (Cell a) => Form a b -> [b]

fmap      :: (Cell a) => (b -> c) -> Form a b -> Form a c
fmapi     :: (Cell a) => (a -> b -> c) -> Form a b -> Form a c
fzipWith  :: (Cell a) => (b -> c -> d) -> Form a b -> Form a c
                                                   -> Form a d
```

Most of these are fairly self-explanatory generalisations of similarly-named Haskell prelude functions. A possible implementation of them is given in appendix A. Use of those definitions requires an instance of the cell class. An example instance is given in appendicix B.

`(!#)` is the infix subscripting operation on a form, so if `a` is a form and `c` is a cell of the right type and dimension, `a !# c` is the corresponding element of `a`. `lattice` and `rank` generalise the standard `bounds` function, returning the lattice and rank of a given form. `accumForm` is similar to `form` but the requirement of one association for every cell is relaxed—the result is a form containing for each cell a list of all values the association list binds to that cell. `fmap` is a straightforward mapping function. `fmapi` is similar, but the mapping function is passed an extra argument, the cell on which the function map is being applied. `fzipWith` is similar to the `zipWith` function on lists. It takes two conforming forms and produces a third by applying the zipping function at each cell.

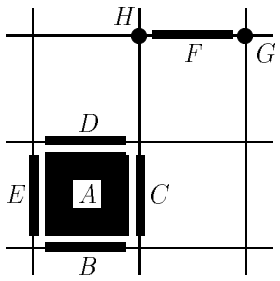
3.2 Lattice topology

The final and most interesting ingredient is the `boundary` operation.

A powerful way of representing the topology of a cell complex is through its *incidence function*. This is a function I from pairs of cells to the set $\{-1, 0, +1\}$. Its value is non-zero only when the cell pair is of the form $(c^{(r)}, c^{(r-1)})$, where $c^{(r)}$ is an r-dimensional cell, and $c^{(r-1)}$ is an $(r-1)$-dimensional cell, *and* $c^{(r-1)}$ *is on the boundary of* $c^{(r)}$. When it is non-zero, its value is $+1$ or -1 according to whether $c^{(r-1)}$ is positively or negatively oriented on the boundary of $c^{(r)}$. Orientation is easier to grasp through pictures than by formal definition—see figure 3. In the figure the conventions are adopted that the links are directed from left to right or from bottom to top, and the boundary of a plaquette is oriented anti-clockwise around the square. So, for example, link B has positive incidence on plaquette A, because it follows the direction of the boundary path, whereas D has negative incidence because it is directed left to right while the top boundary of A goes right to left. For the incidence of a site on a link, the convention is that it is positive if the link is directed in to of the site, and negative if it is directed out of the site[9].

Now the `boundary` function,

[9] Reversing any of these conventions on orientation may change local signs in the incidence function, but *not* the final, scalar results of computations.



$$
\begin{aligned}
I(A,B) &= +1 \\
I(A,C) &= +1 \\
I(A,D) &= -1 \\
I(A,E) &= -1 \\
I(F,H) &= -1 \\
I(F,G) &= +1 \\
I(A,F) &= 0 \\
I(B,H) &= 0
\end{aligned}
$$

Figure 3: Example values of the incidence function on a 2d lattice.

```
boundary  :: (Cell a, Num b) => Lattice a -> a -> [Assoc a b],
```

applied to an r-cell `c` returns the list of $(r-1)$-cells on the boundary of that cell, paired with the incidence function of the boundary cell on `c`. Taking examples in figure 3:

```
boundary l A = [B := 1, C := 1, D := -1, E := -1]
boundary l F = [H := -1, G := 1]
boundary l H = []
```

where `l` is the associated lattice data structure. The result for H is empty, because by convention 0-cells have no boundary. (There is no huge significance in the choice of `Assoc` pairs as the result of `boundary`. In principle, any type capable of representing a cell and a numeric object would do.) Appendix C shows how a particular lattice topology can be set up—defining an instance of the `boundary` function.

The `boundary` operation probably looks a bit contrived at first sight. The idea comes from homology [16]. As the examples in the next section show, it is a convenient practical method of encoding the shape of a lattice.

To close this section we record a couple more functions which will be used later, and which occur frequently in other applications of the "form" technology introduced above. First, a function which "contracts" a form on an association list containing cells and values of the same type:

```
cntrct :: (Cell a, Num b) => Form a b -> [Assoc a b] -> b
cntrct a c = sum [v * (a !# j) | j := v <- c]
```

(This is a sort of scalar product between a form and an association list).

Second, `mltply` multiplies all values in an association list by its first argument.

```
mltply :: (Cell a, Num b) => b -> [Assoc a b] -> [Assoc a b]
mltply m ivs = [i := m * v | i := v <- ivs]
```

4 Form-based implementation of the three algorithms

4.1 Accessing neighbours

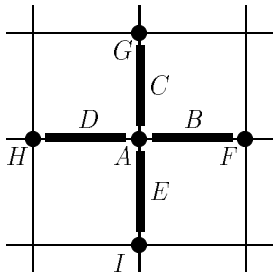
Consider the function

```
neighbours :: (Cell a, Num b) =>
              Lattice a -> Int -> Form a [[Assoc a b]]
```

```
neighbours l r = accumForm l r
  [j := mltply v (if v == (-1) then reverse b else b)
                | b      <- [boundary l i | i <- cells l (r + 1)],
                  j := v <- b]
```

We claim that `neighbours l r` is an r-form, each element of which contains a certain representation of the set of r-cells neighbouring the subscript r-cell in lattice l[10]. This claim will now be illustrated in the case where `r` is zero. The list-comprehension variable `b` is instantiated in turn to the boundary of every link in the lattice. Link B in figure 4 would yield (up to a possible re-ordering of lists)

```
b = [A := -1, F :=  1]
```



$$
\begin{aligned}
I(B,A) &= -1 \\
I(C,A) &= -1 \\
I(D,A) &= +1 \\
I(E,A) &= +1 \\
I(B,F) &= +1 \\
I(C,G) &= +1 \\
I(D,H) &= -1 \\
I(E,I) &= -1
\end{aligned}
$$

Figure 4: Collection of 0-cells neighbouring A.

This gives rise to the terms

```
[ ..., A := [A :=  1, F := -1],
       F := [F :=  1, A := -1], ...]
```

in the outer list comprehension (the list in the second association is obtained by reversal and multiplication by -1). After the application of `accumForm`, the value at site A will be (again, up to possible re-ordering of lists)

```
[[A :=  1, F := -1],
 [A :=  1, G := -1],
 [A :=  1, H := -1],
 [A :=  1, I := -1]]
```

The form produced by `neighbours` contains some redundant information (probably all that was really needed was the identities of F, G, H and I). However, the definition here is generalised to produce forms of neighbours for arbitrary-dimension lattice cells, which will be useful later.

Now we define a mapping function for the 0-cell case, which operates on a pair-wise, nearest-neighbour basis

```
nns :: (Cell a) => (a -> a -> b -> c) -> Form a b -> Form a [c]
nns f a = fmapi (\i ivss ->
```

[10] The application of `reverse` in the definition of `neighbours` is no more than cosmetic for the examples given here, because none of them is sensitive to the list ordering in the result of `boundary`, but see footnote 16.

```
        [f i j (a !# j) | ivs <- ivss, j := v <- ivs, v < 0])
            (neighbours l 0)
    where l = lattice a
```

This exploits the knowledge that neighbours proper appear with a negative value in the association list, filtering out occurrences of the "destination" cell itself. The mapping function is passed the destination cell, the "source" cell and the value of the original form at the source cell. In the example above we would have

```
(nns f a) !# A = [f A F (a !# F), f A G (a !# G),
                  f A H (a !# H), f A I (a !# I)]
```

4.2 Poisson equation relaxation.

With the geometry well sorted out, we can do an application. Define a data type which captures the local state in a relaxation calculation:

```
type Obs    = Float

data State  = Interior Float Obs | Boundary Obs

siteObs :: State -> Obs
siteObs (Interior _  u) = u
siteObs (Boundary    u) = u

siteTrans :: State -> [Obs] -> State
siteTrans (Interior c _) nbs =
           Interior c ((sum nbs + c) / fromIntegral (length nbs))
siteTrans  b             _   = b
```

There are two types of site: normal sites on the "interior" of the lattice, and those on some designated boundary (which could be empty)[11]. Sites on the interior have a local charge density and a potential value. Those on the boundary just have a potential value, which will remain a fixed throughout the iteration. The function `siteObs` extracts the "observable", i.e. the potential, from a `State` variable. `siteTrans` performs the update of equation (3) on the interior sites and leaves boundary sites unchanged. It is passed the old value of the site and a list of potential values at nearest neighbours.

The history of a single site can be regarded as a stream of `State` values[12]. The site evolves through application of a transition function which depends on the (changing) environment of the site. This pattern of evolution can be captured through the function[13]

```
process :: (s -> e -> s) -> s -> [e] -> [s]
process trans st ~(env : envs) = st : process trans (trans st env) envs
```

In the present case the current environment is a list of neighbour potential values, and the last argument to `process` will be a stream of such lists—the history of the environment.

The result of the computation will be a 0-form of lists of `State` values, defining the history of the whole lattice. Suppose we are given this form and need to calculate the 0-form of environments from it. This can be done by

[11] This grid "boundary" should not be confused with the cell `boundary` operation

[12] By definition, a "stream" is an infinite list

[13] `process` can also be implemented by `scanl`. Note the lazy pattern-match on the last parameter of `process`. This laziness is required for the implementations below.

```
env :: (Cell a) => Form a [State] -> Form a [[Obs]]
env = fmap transpose . nns (\_ _ -> map siteObs)
```

The application of `nns`, in which the mapping function extracts a stream of potential values from the stream of `State` values, gives a form containing a list of streams of neighbour histories. Mapping `transpose` over it turns these into streams of neighbour lists. These are just what `siteTrans` needs as its argument. So the implementation of relaxation is

```
evolve :: Form Cell2d State -> Form Cell2d [State]
evolve init = x where x = fzipWith (process siteTrans) init e
                      e = env x
```

The argument is a 0-form of initial states (`fzipWith` is defined to pick up the lattice from its first array parameter).

This code implements Jacobi relaxation. More general schemes, like Gauss-Siedel, can easily be implemented in the same framework. Suppose a site A has site B as a neighbour. The scheme above calculates the new value at A in terms of the previous generation value at B. To calculate the value of A in terms of the *latest* value of B, we should look one place forward in the history of B—in other words the B stream should be tailed before feeding it into the environment of A. To organise this, it is convenient to attribute a colour to each site. If the colour of B, say, is less than the colour A, A is updated in terms of the latest value of B. Otherwise it is updated in terms of the previous generation value of B. The function `env` is changed to

```
env = fmap transpose .
        nns (\i j -> map siteObs .
                     if colour j < colour i then tail else id)
```

Red-black Relaxation in 2 dimensions can be implemented by

```
colour (Cell2d0 (x, y)) = (x + y) `mod` 2
```

4.3 XY model simulation

To describe the local state in an XY model simulation, the `State` data type is changed in the following way:

```
type Obs    = Complex Float

data State = MkStat Float [Float] Obs

siteObs (MkStat _ _ s) = s

siteTrans (MkStat theta rans spin) neighbs = MkStat theta rans'' spin'
  where
    (cand, rans')   = metroCand rans spin
    eDiff           = deltaE neighbs spin cand
    (spin', rans'') = metroChoice spin cand (eDiff / theta) rans'
```

The fields in the `State` record are respectively the temperature, an infinite list of randoms, and the local value of u (sometimes called the "spin"). Each site must hold an

independent stream of random numbers. Creating the random number streams is part of initialisation—we omit details here[14].

As explained in section 2.2, the Metropolis update has three stages—choose a candidate update; calculate the change it would make to the system's energy; decide on the basis of this whether to accept the candidate or retain the old value at the site. The details are:

```
-- Energy difference between two spin states.
deltaE neighbs s s' = - real (conjugate (sum neighbs) * (s' - s))
                      where real (x :+ _) = x

-- Canditate new spin for Metropolis update.
metroCand (r : rans) old = (exp (0 :+ delta) * old, rans)
                           where delta = max_delta * (2 * r - 1)

-- Metropolis choice.  Inputs are old and candidate new variable,
-- energy difference of configurations, and a list of randoms.
metroChoice _   new ediff  rans       | ediff < 0 = (new, rans)
metroChoice old new ediff (r : rans') =
  (if accept then new else old, rans') where accept = r < exp (- ediff)
```

`metroCand` returns a unit-magnitude 2-vector which differs from the one input by rotation through a random angle in the range [−`max_delta`, +`max_delta`].

The remainder of the code is identical to that in the previous section[15].

4.4 Lattice gauge theory simulation

The 1-form analogue of `nns` is

```
staples :: (Cell a, Num b) => (a -> [(a, b, c)] -> d) ->
                              Form a c -> Form a [d]
staples f a =
  fmapi (\i ivss ->
         [f i [(j, v, a !# j) | j := v <- snip i ivs] | ivs <- ivss])
                                                    (neighbours l r)
  where l           = lattice a
        r           = rank a
        snip i ivs  = tail (rotate (\(j := _) -> j == i) ivs)
        rotate p xs = b ++ f where (f, b) = break p xs
```

The mapping function has a more complicated specification here. There are two reasons for this. First, each adjacent plaquette defines several (in a cubic lattice, three) neighbours to a given link, whereas an adjacent link only defined a single neighbour to a given site. Second, orientation information is irrelevant to sites, but important for links (links are directed; sites are not). The mapping function again takes the "destination" cell as its first argument. Its second argument is a list. The items in this list correspond to the links in a single staple (see figure 2). Each item is a tuple containing the coordinate (cell) of

[14] There are various approaches to getting independent streams of pseudo-random numbers. If the basic generator has a long enough cycle, simply taking independent seed values (perhaps chosen by a different generator) may be acceptable. Alternatively, there are various "leapfrog" schemes available [17], which are quite elegant. I actually used a leapfrogged 64-bit linear congruential generator.

[15] The Red-black variant is preferred—in this case the "Jacobi" version would actually be invalid because neighbouring sites connected by the energy function would be updated "concurrently", and localised calculation of the energy change would be erroneous.

the link, its orientation relative to the destination cell (+1 or −1 depending on whether it points in the same or the opposite direction around the boundary of the plaquette), and the value held at the link[16].

Now, the definition of `env` is changed to

```
env = fmap transpose .
      staples (\i lss ->
                 (map product . transpose)
                 [((if v == -1 then map conjugate else id) .
                   map siteObs .
                   (if colour j < colour i then tail else id)) ls |
                                                 (j, v, ls) <- lss])
```

The list comprehension pre-processes the streams of states associated with the links on a given staple. In contrast to the previous version, variables are conjugated here if they have negative orientation relative to the destination link (in the XY model, this conjugation was also necessary, but was deferred to `deltaE`). The resulting streams are zipped together by the `transpose`, `map product` pipe, to produce a stream of products around the staple.

Colouring is slightly more complicated than before. We have to ensure that no two links lying on the same plaquette have the same colour (see footnote 15). For a 4 dimensional cubic array a suitable definition might be[17]

```
colour (Cell4d1 (t, x, y, z) d) = 4 * ((t + x + y + z) `mod` 2) + d.
```

Finally, the local state and updating can be lifted verbatim from the XY model, except that `deltaE` is simplified to

```
deltaE neighbs s s' = - real (sum neighbs * (s' - s))
                      where real (x :+ _) = x
```

because the conjugation was already done in `env`. The top-level evolution is unchanged from section 4.2 (the type specification of `evolve` has to be altered slightly). One might want to change some names: the "site" prefixes ought to become "link" or, more generically, "cell".

5 Discussion

The algorithms described involve state transition in an essential way, but it is doubtful whether their description would have been any clearer in terms of variables and assignments. The explicit representation of sequences as infinite lists seems just as natural.

The implementations in section 4 create an "array of streams". An alternative approach is to create a "stream of arrays", in which each update is performed on the whole lattice to create a new global state. It only involves minor changes to the implementation to create such a stream of arrays directly. Alternatively one can feed the results of the given implementation to a pipeline of `felems`, `transpose` and `map (listForm l r)`

[16] The function `snip` cuts out the destination link from the list of links around the plaquette. In `nns`, this was done by a filter in the list comprehension. The same method could be used here. But that would loose information on the order of links around the staple (assuming that the implementation of `boundary` returned the links around a plaquette in path-order to start with). This information is preserved by the device of rotating the list so the destination link is at the front, and tailing it. If the gauge group was non-Abelian (as it is in full QCD, but not in the simplified case described here) this path-ordering information would be important.

[17] To save space, this article does not define an implementation of 4-dimensional geometry. The modifications necessary to the 2-dimensional case given in the appendices are reasonably straightforward.

to produce a list of arrays. There are some advantages to the "array of streams" approach (apart from the fact that it models process-parallelism). Some existing Haskell implementations do not handle array accesses very efficiently[18], and in this approach the array operations are confined to an initial stage where the streams are "tied together". In the "stream of arrays" approach array operations are needed in all updates. On the other hand it does seem more natural to have a list of arrays when it comes to processing the data (the set of configurations) generated by a simulation.

In any case, once the ideas about lattices and arrays in section 3 are absorbed, implementation of the three algorithms is very straightforward. It was worth the effort of doing the geometry carefully, for the level of generality it gave in the implementations. For example, apart from the definition of `colour`, the modifications made to the XY model simulation in section 4.4 to encompass lattice gauge theory did not actually invalidate the code as an implementation of the XY simulation—the final version works for systems with variables defined on arbitrary dimension cells of an arbitrary dimension lattice (given an appropriate instance of the `Cell` class). Achieving this level of abstraction depended on both ordinary (parametric) polymorphism and overloading, together with use of higher order functions.

The price one pays for this generality is that while all the programs given in this article are executable, they run too slowly to be regarded as production simulation programs. Unless functional compilers make rapid progress, it is difficult to imagine that *exactly* the kind of programs given here could be competitive in this rôle in the immediate future. Presumably the programs could be optimised to gain significant factors in speed (and efficiency of memory utilisation). Higher order functions could be used with sparingly, and probably "granularity" could be increased in various ways. A few efforts were made in this direction, but these were hindered by difficulty in understanding the behaviour (in terms of execution speed and memory utilisation) of the programs. Programmers who need speed are usually happy to invest some effort in optimising their own codes, which they often do better than compilers, provided they have a reasonably accurate idea of what goes on during execution. It is relatively difficult to form such a mental model for a lazy functional language, but good profiling tools may make the job easier[18].

As they stand, though, the programs capture the essence of the algorithms in a way that a production FORTRAN code probably never could. At the very least, programs like this must have a rôle in specification or prototyping.

References

[1] J. Liu, P.H.J. Kelly, S.M. Cox and F.S. Taylor, "Functional Programming for Scientific Computation" in *Functional Programming for Arrays of Transputers: The Collected Papers*, in press.

[2] W.G. Vree, "The Grain Size of Parallel Computations in a Functional Program" in *Parallel Processing and Applications*, edited by E. Chiricozzi and A. D'Amico, Elsevier Science Publications (1987).

[3] D. Cann, "Retire Fortran? A Debate Rekindled", *Communications of the ACM*, volume 35, No. 8 (1993) p81.

[4] P. Hudak and J.H. Fasel, "A Gentle Introduction to Haskell", *ACM SIGPLAN Notices*, volume 27, No. 5 (1992).

[18] It seems that functional programming and parallel programming share this characteristic—if "array" is interpretted as distributed data in the parallel case. Locality of reference is a helpful feature of an algorithm for both paradigms.

[5] P. Hudak, S. Peyton Jones and P. Wadler (editors), "Report on the Programming Language Haskell, A Non-strict Purely Functional Language", *ACM SIGPLAN Notices*, volume 27, No. 5 (1992).

[6] A.J.T. Davie, "An Introduction to Functional Programming Systems Using Haskell", Cambridge University Press (1992).

[7] S. Cox, S.-Y. Huang, P. Kelly, J. Liu, F. Taylor, "An Implementation of Static Functional Process Networks", in *PARLE '92: Parallel Architectures and Languages, Europe*, edited by D. Etiemble and J.-C. Syre, Springer Verlag (1992) p497.

[8] R.P. Feynman and A.R. Hibbs, "Quantum Mechanics and Path Integrals", McGraw-Hill, New York (1965).

[9] K. Wilson, "Confinement of Quarks", *Physical Review, D: Particles and Fields*, volume 10 (1974) p2445.

[10] M. Creutz, "Quarks, Gluons and Lattices", Cambridge University Press (1983).

[11] J.B. Kogut, "An Introduction to Lattice Gauge Theory and Spin Systems", *Reviews of Modern Physics*, volume 51 (1979) p659.

[12] See, for example, A. Brandt, "Multi-level Adaptive Solutions to Boundary-Value Problems", *Mathematics of Computation"*, volume 31 (1977) p333.

[13] J.M. Kosterlitz and D.J. Thouless, "Ordering, Metastability and Phase Transitions in Two-dimensional Systems", *Journal of Physics C: Solid State Physics*, volume 6 (1973) p1181.

[14] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, *Journal of Chemical Physics*, volume 21 (1953) p1087.

[15] H. Flanders, "Differential Forms, with Applications to the Physical Sciences", Academic Press (1963).

[16] See, for example, M.A. Armstrong, "Basic Topology", Springer Verlag (1983).

[17] See, for example, S. Aluru, G.M. Prabhu and J. Gustafson, "A Random Number Generator for Parallel Computers", *Parallel Computing*, volume 18 (1992) p839.

[18] C. Runciman and D. Wakeling, "Heap Profiling of Lazy Functional Programs", Dept. of Comp. Sci., Univ. of York, England, Technical Report 172 (1992).

A The "form" library

```
module Lattice (Cell (cellEnum, cellArray, cellAccum, dimension),
                Lattice (MkLattice),
                Form,
                cells, boundary,
                form, accumForm, lattice, rank, (!#), (//#),
                listForm, fassocs, findices, felems,
                fmap, fmapi, fzipWith) where

-- The underlying technology of lattices and their forms...
-- This parallels the 'PreludeArray' section for arrays.
```

```
infixl 9 !#
infixl 9 //#

-- The `Cell' class is the analogue of the `Ix' class in `PreludeCore'.

class (Eq a) => Cell a where
  cellEnum  :: Int -> [a] -> [a]
  cellArray :: Int -> [a] -> [Assoc a b] -> a -> b
  cellAccum :: Int -> [a] -> [Assoc a b] -> a -> [b]
  dimension :: a -> Int


-- First field in `Lattice' defines extent of the complex, where required.
-- Second field is the `boundary' function for the complex.

data (Cell a) => Lattice a = MkLattice (Int -> [a]) (a -> [Assoc a Int])
data (Cell a) => Form a b  = MkForm (Lattice a) Int (a -> b)

cells     :: (Cell a) => Lattice a -> Int -> [a]
boundary  :: (Cell a, Num b) => Lattice a -> a -> [Assoc a b]


form      :: (Cell a) => Lattice a -> Int -> [Assoc a b] -> Form a b
accumForm :: (Cell a) => Lattice a -> Int -> [Assoc a b] -> Form a [b]
lattice   :: (Cell a) => Form a b -> Lattice a
rank      :: (Cell a) => Form a b -> Int
(!#)      :: (Cell a) => Form a b -> a -> b

(//#)     :: (Cell a) => Form a b -> [Assoc a b] -> Form a b

listForm  :: (Cell a) => (Lattice a) -> Int -> [b] -> Form a b

fassocs   :: (Cell a) => Form a b -> [Assoc a b]
findices  :: (Cell a) => Form a b -> [a]
felems    :: (Cell a) => Form a b -> [b]

fmap      :: (Cell a) => (b -> c) -> Form a b -> Form a c
fmapi     :: (Cell a) => (a -> b -> c) -> Form a b -> Form a c
fzipWith  :: (Cell a) => (b -> c -> d) -> Form a b -> Form a c
                                                   -> Form a d

-- `Lattice' primitives...

cells    (MkLattice cls _) r = cellEnum r (cls r)
boundary (MkLattice _ bdy) i = [j := fromIntegral s | j := s <- bdy i]

-- `Form' primitives...

form      l@(MkLattice cls _) r ivs = MkForm l r (cellArray r (cls r) ivs)
accumForm l@(MkLattice cls _) r ivs = MkForm l r (cellAccum r (cls r) ivs)
```

```
lattice (MkForm l _ _) = l
rank    (MkForm _ r _) = r
(!#)    (MkForm _ _ d) = d

-- Higher level operations on Forms...

a //# us   = form (lattice a) (rank a)
             ([i := a !# i | i <- findices a \\ [i | i := _ <- us]] ++ us)

listForm l r vs = form l r (zipWith (:=) (cells l r) vs)

fassocs  a = [i := a !# i | i <- cells (lattice a) (rank a)]
findices a = cells (lattice a) (rank a)
felems   a = [a !# i | i <- findices a]

fmap   f a = form l r [i := f (a !# i) | i <- cells l r]
             where l = lattice a
                   r = rank    a

fmapi  f a = form l r [i := f i (a !# i) | i <- cells l r]
             where l = lattice a
                   r = rank    a

fzipWith f a b = form l r [i := f (a !# i) (b !# i) | i <- cells l r]
                 where l = lattice a
                       r = rank    a
```

B An example instance of the "Cell" class

```
module Cell2d (Cell2d (Cell2d0, Cell2d1, Cell2d2)) where

-- A 'Cell' instance suitable for representing cells in a 2d grid.

import Lattice (Cell (cellEnum, cellArray, cellAccum))

data Cell2d = Cell2d0 (Int, Int)     |
              Cell2d1 (Int, Int) Int |
              Cell2d2 (Int, Int)         deriving (Text)

ix0 (Cell2d0 i)        = i
ix1 (Cell2d1 (x, y) d) = (x, y, d)
ix2 (Cell2d2 i)        = i

bd0 [c1, c2]    = (ix0 c1, ix0 c2)
bd1 [c1, c2]    = (ix1 c1, ix1 c2)
bd2 [c1, c2]    = (ix2 c1, ix2 c2)

cell0 i         = Cell2d0 i
cell1 (x, y, d) = Cell2d1 (x,y) d
cell2 i         = Cell2d2 i
```

```
instance Cell Cell2d where

  cellEnum r cls =
    case r of
      0 -> [cell0 i | i <- range (bd0 cls)]
      1 -> [cell1 i | i <- range (bd1 cls)]
      2 -> [cell2 i | i <- range (bd2 cls)]

  cellArray r cls ivs =
    case r of
      0 -> (array (bd0 cls) [ix0 i := v | i := v <- ivs] !) . ix0
      1 -> (array (bd1 cls) [ix1 i := v | i := v <- ivs] !) . ix1
      2 -> (array (bd2 cls) [ix2 i := v | i := v <- ivs] !) . ix2

  cellAccum r cls ivs =
    case r of
      0 -> (accumArray (flip (:)) [] (bd0 cls) [ix0 i := v | i := v <- ivs] !)
             . ix0
      1 -> (accumArray (flip (:)) [] (bd1 cls) [ix1 i := v | i := v <- ivs] !)
             . ix1
      2 -> (accumArray (flip (:)) [] (bd2 cls) [ix2 i := v | i := v <- ivs] !)
             . ix2
```

C Example definition of lattice geometry

```
module Torus2d (torus2d) where

-- `torus2d' returns an object of type `Lattice Cell2d' implementing
-- a 2 dimensional grid with periodic boundary conditions.

import Lattice (Cell (cellEnum, cellArray, cellAccum), Lattice (MkLattice))

import Cell2d (Cell2d (Cell2d0, Cell2d1, Cell2d2))

torus2d :: Int -> Int -> Lattice Cell2d
torus2d lx ly = MkLattice (cls lx ly) (bdy lx ly)

-- Lattice bounds, specified by smallest and largest cell of each dimension:

cls lx ly 0 = [Cell2d0 (0, 0),   Cell2d0 (lx - 1, ly - 1)]
cls lx ly 1 = [Cell2d1 (0, 0) 0, Cell2d1 (lx - 1, ly - 1) 1]
cls lx ly 2 = [Cell2d2 (0, 0),   Cell2d2 (lx - 1, ly - 1)]

-- Lattice topology (cell boundary):

bdy _ _ (Cell2d0 _) = []

bdy lx ly (Cell2d1 (x, y) d) =
  [Cell2d0 (x, y) :=  -1,
   Cell2d0 (case d of
              0 -> ((x + 1) `mod` lx, y)
```

```
              1 -> (x, (y + 1) `mod` ly)) := 1]

bdy lx ly (Cell2d2 (x, y)) =
  [Cell2d1 (x, y) 0 :=  1,
   Cell2d1 ((x + 1) `mod` lx, y) 1  :=  1,
   Cell2d1 (x, (y + 1) `mod` ly) 0  := -1,
   Cell2d1 (x, y) 1 := -1]
```