Expressing some grid-based algorithms in Haskell

D.B. Carpenter

November 29, 1992

Contents

1 Ojectives

- Explore expressive power of functional language, applied to some favourite scientific problems.
- Exercise the FAST compiler, on some number-crunching applications.
- Produce some demos for Hugh.

The applications will be grid oriented, including PDE (Poisson equation), statistical mechanics spin models, and lattice gauge theory (QCD).

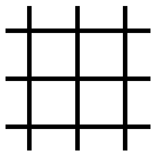
First aim is to exploit the high-level features of functional language to express the algorithms in the most natural mathematical way.

Second aim is to go fast.

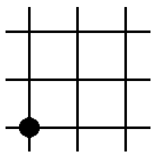
2 Nomenclature

Will use nomenclature common in QCD:

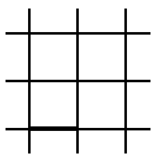
Lattice = "grid"



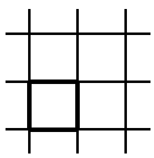
Site = "point" or "node".



Link = "edge" or "arc".



Plaquette = "square".



In general, lattice has arbitrary dimensionality, arbitrary boundary-conditions (eg, open or toroidal).

Can regard lattice simply as an array of sites, or as some sort of complex of sites, links, plaquettes, etc, with some incidence function between these elements.

3 Poisson relaxation

Continuum Poisson equation:

$$(\Box u)(x, y) = c(x, y),$$

where c is given charge distribution; u is electrostatic potential, to be calculated. Discretise:

$$\begin{aligned} u(x+1, y) + u(x-1, y) + & \\ u(x, y+1) + u(x, y-1) - 4u(x, y) &= c(x, y). \end{aligned}$$

Naive relaxation—rearrange equation:

$$u(x, y) = \frac{1}{4}(\text{neighbouring } u \text{ values} - c(x, y)),$$

then iterate from some initial guess—new u obtained by substituting old u in RHS.

Two natural ways of arranging this iteration in a lazy language:

- Build a stream of arrays. Each successive item in the stream is an improved approximation to the array u.

- Build an array of streams. Each element of the array is a list of gradually improving approximations to the value of u at a particular site.

Introduce some useful bits and pieces. First some very general operations:

A handy generalisation of *amap*

```
amapi f a = array b [i := f i (a ! i) |
                     i <- range b]
            where b = bounds a
```

A useful primitive for evolving state transition systems

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

(NB: another implementation is $process = scanl$)

Some things more specific to Poisson's equation:

An operation for accessing neighbours

```
neighbours (x, y) = [(x - 1, y), (x + 1, y),
                     (x, y - 1), (x, y + 1)]

a !# i = [a ! j | j <- neighbours i]
```

A type to capture the state of a site

```
data State = Interior Float Float |
             Boundary       Float

siteObs (Interior _ u) = u
siteObs (Boundary   u) = u

siteTrans (Interior c _) nbs =
           Interior c (0.25 * (sum nbs - c))
siteTrans b              _   = b
```

siteObs extracts the potential. *siteTrans* is the transition function for relaxation (sites on a boundary aren't updated).

A function which returns the history of a site, given an initial state and the history of its neighbours:

```
siteStrm _ ini nbStrms =
              process siteTrans ini env
              where env = transpose nbStrms
```

Now tie up the site streams:

```
arrStrms inia = ssa
    where ssa      = amapi f inia
          f i ini = siteStrm i ini (osa !# i)
          osa      = amap (map siteObs) ssa
```

(Use convention that identifiers ending *s* are lists, in *a* are arrays; *s* stands for *State*, *o* for *observable*). *arrStrms* returns an array of *State* streams, given an array of initial states.

Note—neighbour operations don't go out of array bounds, due to lazy evaluation.

For comparision, a "stream of arrays" implementation built from the same components:

```
arrTrans sa oa = amapi f sa
         where f i s = siteTrans s (oa !# i)

strmArrs inia = sas
      where sas   = process arrTrans inia oas
            oas   = map (amap siteObs) sas
```

4 XY model

Solving Poisson equation is equivalent to the variational problem: minimise

$$E(u) = \sum_{x,y} \frac{1}{2}\Big(\begin{array}{l} (u(x+1,y) - u(x,y))^2 + \\ (u(x,y+1) - u(x,y))^2 \;\Big) + \\ c(x,y)u(x,y). \end{array}$$

Typical statistical physics simulations involves a similar type of energy functional, but rather than minimising E, the object is to generate an ensemble of configurations with a Boltzmann probability distribution:

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

where θ is the temperature of the system. (NB. Equivalent to minimisation when $\theta \to 0$; Simulated Annealing).

XY model is a "spin system". The degree of freedom at a site is a unit 2-vector, or unit magnitude complex number (rather than a single real number in Poisson). Energy function is

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

Model is famous for having an unusual phase transition, driven by condensation of topological dislocations (vortices).

Main changes to Poisson code are in definition of *State* type. (I also changed `!#` to implement periodic boundary conditions, so no worry about boundaries and interiors here.)

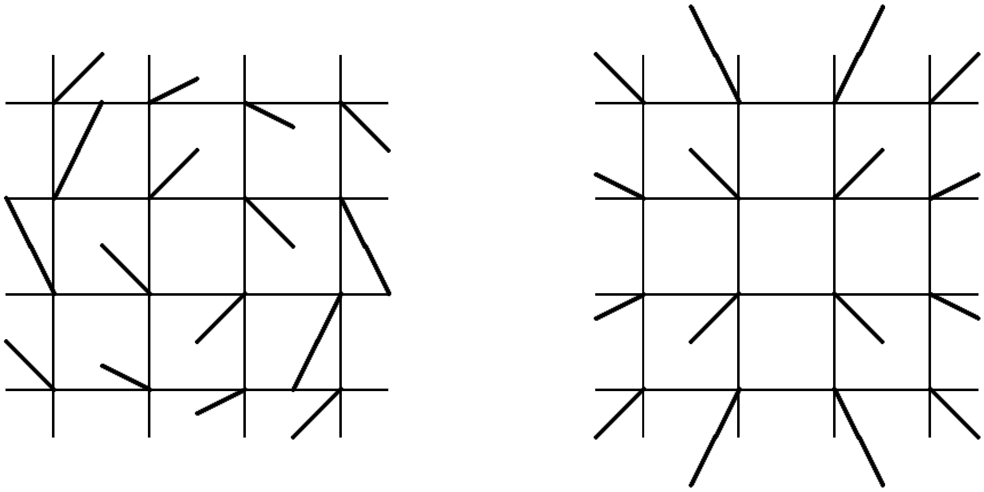
Initial array has to include a stream of independent random numbers for each *State* variable.

```
data State = MkStat Float [Float] Spin
-- temperature, a list of randoms and ``spin''.

type Spin  = Complex Float

siteObs :: State -> Spin
siteObs (MkStat _ _ s) = s

siteTrans :: State -> [Spin] -> State
-- Local Metropolis update
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'
```

Details of Metropolis code:

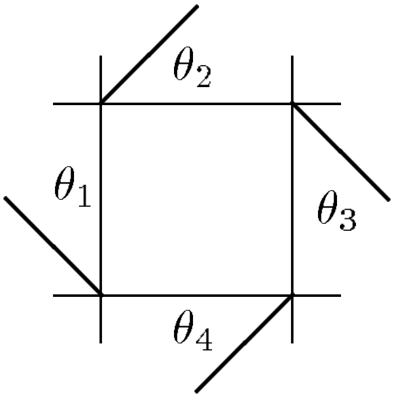
```
-- Energy difference between two spin states.
deltaE neighbs s s' = x
  where x :+ _ = - (conjugate (sum neighbs)) *
                   (s' - s)

-- 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)
```

Would like to automatically spot vortex configurations, eg



Number of vorticies inside a given cyclic contour given by integrating the local rate of change of vector direction around the contour (and dividing by 2π).

The smallest cyclic "contour" in lattice is plaquette. Number of vorticies in plaquette:



is

$$(\Delta(\theta_2, \theta_1) + \Delta(\theta_3, \theta_2) + \Delta(\theta_4, \theta_3) + \Delta(\theta_1, \theta_4))/2\pi$$

where

$$\Delta(x, y) = x - y \bmod 2\pi$$

Can't be done conveniently with the simple *amapi*, `!#` primitives introduced so far.

5 Lattice gauge theory

Quantum field theories like QCD can be reformulated to look like statistical mechanics problems.

In a gauge theory, the variables are *group elements*, U, which lie on *links* of lattice. Energy function associated with plaquettes of lattice (cf XY model, where variables associated with sites and energy associated with links).

With a closed path P starting at any point in lattice can associate a group element U_P by multiplying together the link variables in path order.

Energy is

$$E(U) = \sum_{P \in plaquettes} \chi(U_P)$$

where *plaquettes* is the set of plaquette-paths in the lattice and χ is a characteristic function for the group. Normally U represented by complex matrices (eg, 3×3 unitary matrices) and χ just real part of trace of a matrix.

Gauge theories "trivial" in 2 dimensions, theoretically interesting in 3 dimensions, and real QCD lives in 4 dimensions.

Again, *amapi*, `!#` primitives inadequate to capture this geometry.

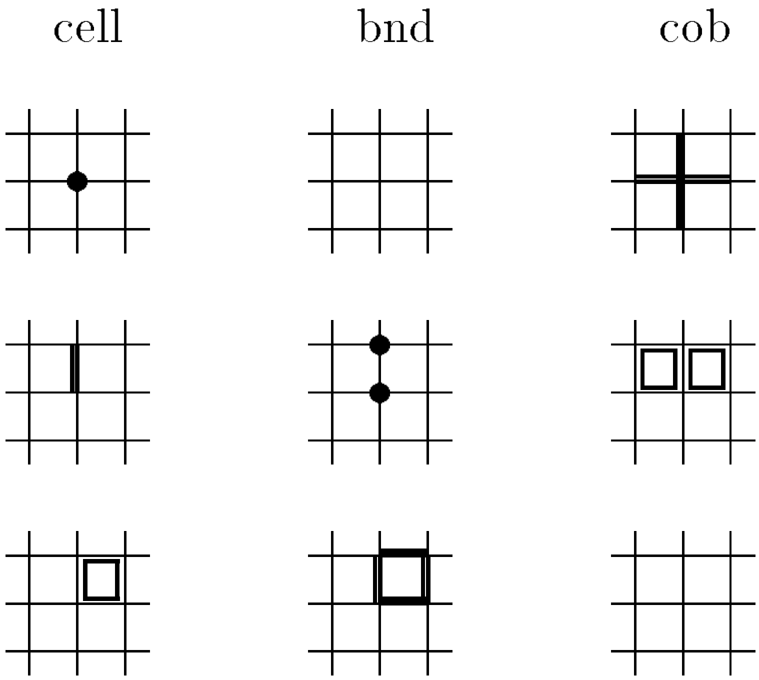
6 A more general view of lattice geometry?

Currently, effectively just have arrays defined on sites of lattice.

- Program:

  Generalise so that array index space can also be links, plaquettes, etc.

  Replace *neighbours* operation by operations which take the *boundary* or *coboundary* of a particular cell



  Now replace *amapi* by a function *bndmapi* which takes an array over (eg) the plaquettes to an array over the links, applying the map function on a "per-incidence" (overlap of plaquette and link) basis. Similarly *cobmapi*.