Expressing some scientific, lattice-based problems in Haskell

D.B. Carpenter

January 27, 1993

1 Introduction

Will describe work done in the FAST project. (Functional programming on Arrays of Transputers). Aim was to produce a set of "scientific" demonstrator applications.

One of the paradigms easily expressible in lazy functional languages is process-parallel programming—FAST project produced a system for parallel execution of annotated programs written in this style.

Plan of talk

1. Introduction to functional programming and Haskell
2. Lattices and generalised arrays
3. Implementation of some scientific algorithms

2 About functional languages

Functional programming languages are very mathematically "pure". They have no intrinsic idea of control-flow, store, assignment, etc. A functional program is a mathematical expression. This expression may be a function—say from inputs to outputs.

Higher order functions (functions which take other functions as parameters, or return other functions as results) can be used freely. In practise this allows imperative control-flow to be emulated, if necessary.

Lazy evaluation permits manipulation of "infinite" data structures, in particular infinite lists. These provide an alternate means of representing state transition—the history of a transition system is an infinite list (stream) of system states.

So we can recover sequential programming if desired (usually not), and also model more general imperative paradigms, including process parallelism, easily.

Beyond that, functional language are very *uniform*, and free of ad hoc restrictions. This makes them very expressive, and allows high levels of abstraction.

3 Some basic types

The Haskell system comes with a large number of pre-defined types (including `Int`, `Integer`, `Bool`, `Char`, `Float`, complex types, etc). A constant can be specified to have a particular type by a declaration

```
a :: Int
```

A function type is a type like any other

```
f :: Int -> Float
```

means `f` a function from `Int` to `Float`. Functions can take functions as arguments, or return functions as results

```
g :: (Int -> Float) -> Float
h :: Int -> (Int -> Float)
h :: Int -> Int -> Float
```

Multiple arguments represented by "currying". The two signatures of `h` are equivalent by definition and represent a function from an `Int` to a `Int -> Float`—aka a function from two `Int`s to a `Float`.

Another basic type is the tuple

```
a :: (Int, Float)
b :: (Int, Int, Int)
```

A function from a pair to a float:

```
h' :: (Int, Int) -> Float
```

4 Function application—expressions

Function application represented by juxtaposition

```
f 3
g f
h 3 4
h' (3, 4)
```

Functions of two arguments can be used as "infix" operators

```
(+), (*) :: Float -> Float -> Float
```

(obviously these are predefined and overloaded in practise) so arithmetic expressions can be written in familiar ways (and other function applications can be written in unfamiliar ones)

```
3.0 + (4.0 * f 3)
3 `h` 4
```

Due to currying, functions can be partially applied

```
h 3
```

is a function from `Int` to `Float`, and

```
(2.0 *)
```

is a function from `Float` to `Float`. (These are elementary examples of function-valued expressions).

5 Data constructor function definition

"Algebraic types" are roughly equivalent to "variant records" or "Unions" in some languages. They have an associated set of "constructor functions", which are genuine functions, but with limited power.

```
data IntList = Node Int IntList | Nil
```

defines an algebraic type `List` with two constructor functions `Node` and `Nil`. `Node` is of type `Int -> IntList -> IntList`. `Nil` is a constant ("function of zero arguments") of type `IntList`.

The following expressions are of type `IntList`

```
Nil
Node 1 (Node 2 (Node 3 Nil)))
1 ‘Node‘ Nil
```

Not surprisingly, Haskell has built in syntax for representing general lists—this is just given as an example of algebraic types.

6 General function definition

General functions are defined by a straightforward syntax

```
double :: Real -> Real
double x = 2.0 * x

factorial :: Int -> Int
factorial n = if n == 0
                then 1
                else n * factorial (n - 1)
```

Functions on algebraic types are handled by "pattern-matching"

```
isEmpty :: IntList -> Bool
isEmpty (Node x xs) = False
isEmpty  Nil        = True

car :: IntList -> Int        -- ``head''
car (Node x xs) = x

cdr :: IntList -> Intlist    -- ``tail''
cdr (Node x xs) = xs
```

Note `car` and `cdr` are undefined if the argument is `Nil`.

`double` could also be defined by

```
double = (2.0 *)
double = (\x -> 2.0 * x)
```

7 Special syntax for lists

Lists can be defined for scratch—a restricted form (`Ints` only) was already given. They are used so commonly that some special syntax has been added to the language. The equivalent of `Node` is `(:)`, and it's normally used as an infix function. The equivalent of `Nil` is `[]`. Additionally a list of several items can be written as comma-separated sequences in square brackets. The following are equivalent:

```
(:) 1.0 ((:) 2.0 [])
1.0 : (2.0 : [])
[1.0, 2.0]
```

This is a list of two `Float` items. Its *type* is written: `[Float]`. This is an example of a parametrised type—in this case the parameter type is `Float`.

A "list comprehension" is another special piece of syntax for building lists. A simple example is the expression

```
[x^2 | x <- xs]
```

Here `xs` is a list of numeric values. The result of the expression is a list formed by instantiating `x` to the value of each item in `xs`, and computing the expression preceding the `|`. If `xs = [1, 2, 3]`, the result of the above expression is `[1, 4, 9]`. (Fortran users might interpret this as something like an implied do loop!) This syntax has a few extensions—multiple generators can be placed after the `|` (nested implied do loop!), and predicates can be added to filter out certain values of the "dummy indices".

8 Normal order evaluation

Lazy languages implement "normal order evaluation" of expressions. Evaluation of an expression starts "at the outermost level", and no sub-expression is evaluated unless it is actually needed. Eg, if we had

```
cond :: Bool -> a -> a -> a
cond True  x y = x
cond False x y = y

n = 0
```

then the expression `cond (n != 0) (1 / n) 0` is well-defined. Only the first argument has to be evaluated to make the pattern-match.

9 Infinite data structures

This means that we can deal in expressions which would take ∞ly long to evaluate in full, so long as we only choose to look at a parts of the result which can be evaluated in finite time. It often simplifies construction of data structures if one doesn't worry initially about termination.

An application of infinite lists is in modelling state transition. Define

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

This takes: a transition function from a state and a current "environment value" to a new state; an initial state; and an ∞ list of successive environment values. It returns an ∞ list of states.

As an example, take the "transition function" to be `(+)`. This adds the current value of the "environment" to the current value of the "state". Then we would have

```
process (+) 0 [1, 3, 5, 7, ...]  ==
                      [0, 1, 4, 9, 16, ...]
```

So long as the computation only ultimately requires a finite number of items from this list, the fact that it formally contains an ∞ number of items doesn't matter.

10 Some standard functions

Some standard functions:

```
map f xs
   = [f x | x <- xs]

zipWith f [x, y, ...] [a, b, ...]
   = [f x a, f y b, ...]

transpose [[a1, a2, ...], [b1, b2, ...], ...]
   = [[a1, b1, ...], [a2, b2, ...], ...]

(f . g) x
   = f (g x)
```

All these can easily be defined in terms of the operations given so far (and are so-defined in the prelude). All of them are applicable to finite or infinite lists.

11 Haskell arrays

An array can be constructed from an "association list". An association is a pair constructed with the infix data constructor `(:=)`, binding a subscript to a value. `array` is a function which takes a pair of bounds and an association list, and returns an array. Eg

```
a = array (1, 3) [1 := Red,
                  2 := Blue, 3 := Green]
```

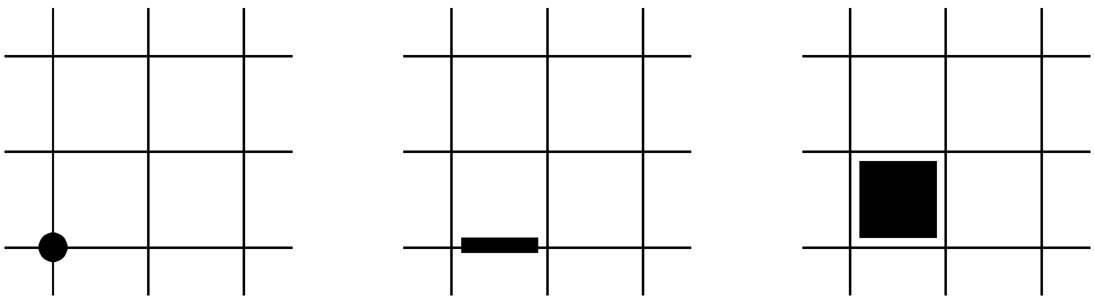
is an array with subscript type `Int` (tuples are also allowed, providing multi-dimensional arrays) and values in some algebraic type representing colours. The subscripting operator is the infix function `(!)`, so

```
a ! 2  ==  Blue
```

Arrays aren't "primitive" in Haskell. They can be implemented (inefficiently) in terms of function. They don't even have any special syntax, as lists do. So it's possible to implement ones own array system (provided efficiency isn't a overiding concern).

12 Lattices

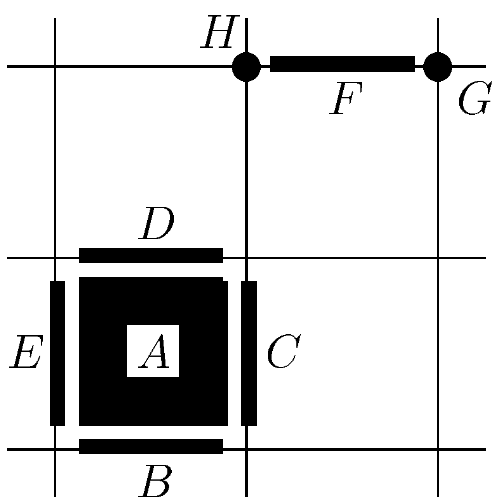
QCD nomenclature for the "cells" of a lattice



Site Link Plaquette

Variables in gauge theory and other lattice-based models "live" on different dimensional cells of the lattice.

Similar "cell complexes" arise in topology. The structure of a complex can be encoded in a "boundary operator", ∂.



$$\begin{aligned} \partial F &= G - H \\ \partial A &= B + C - D - E \\ \partial H &= 0 \end{aligned}$$

The boundary operation is powerful enough to encode the topology of an *arbitrary* cell complex.

13 Cells and generalised arrays in Haskell

Cells of a 2d square lattice can be represented by

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

A "form" is an array whose index space is the set of cells of a particular dimension in a particular lattice.

Suppose `l` is a data structure representing a 2×2 lattice

```
a = form l 0 [Cell2d0 (0, 0) := Red,
              Cell2d0 (0, 1) := Blue,
              Cell2d0 (1, 0) := Green,
              Cell2d0 (1, 1) := Yellow]
```

is a "0-form", taking values (colours, here) on the *sites* of `l`. Subscript by `(!#)`:

```
a !# Cell2d0 (1, 0)  ==  Green
```

Haskell has a class system, which allows the basic "form" operations to be *overloaded* over arbitrary instances of "cell" types and so arbitrary lattice shapes—`Cell2d` is just one example. The type signature of `form` is

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

14 More operations on cells and forms

- `cells l r` returns a list of all cells of dimension `r` in lattice `l`.
- `boundary l i` returns a list of the cells (paired with their incidence function) on the boundary of cell `i` in lattice `l`.
- `fmap` is like `map`. It takes a mapping function and a form, and returns a form.
- `fzipWith` is like `zipWith`. It takes a zipping function and a pair of forms, and returns a form.
- `nns` is complicated.

`nns` is a function which takes a mapping function and a 0-form, and returns a 0-form in which each element is a mapped list of the nearest neighbour elements of the original form. If $j_1, j_2, \ldots$ are the neighbours of site i, and `b = nns f a`, then

`a !#` i `== [f` i j_1 `(a !#` j_1`), f` i j_2 `(a !#` j_2`), ...]`

15 Implementation of nns

All operations are "local" except `boundary`. This works for the sites of *any* shape lattice.

```
mltply m ivs = [i := m * v | i := v <- ivs]

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]

nns :: (Cell a) => (a -> a -> b -> c) ->
                     Form a b -> Form a [c]
nns f a = fmapi (\i ivss ->
            [f i j (a !# j) | ivs <- ivss,
                              j := v <- ivs,
                              v < 0])
                (neighbours l 0)
          where l = lattice a
```

16 Poisson relaxation

Continuum Poisson equation:

$$(\nabla^2)(x, y) = c(x, y),$$

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.

A type to capture the state of a site

```
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
```

Given an array containing streams at each site, representing the history of that site, `env` returns an array of streams of each site's environment.

```
env = fmap transpose . nns (\_ _ -> map siteObs)
```

Given an array of initial states, evolve produces an array of site "history" streams

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

17 Red-black relaxation

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

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

18 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 "vortices".

Only changes to Poisson code are in definition of *State* type.

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:

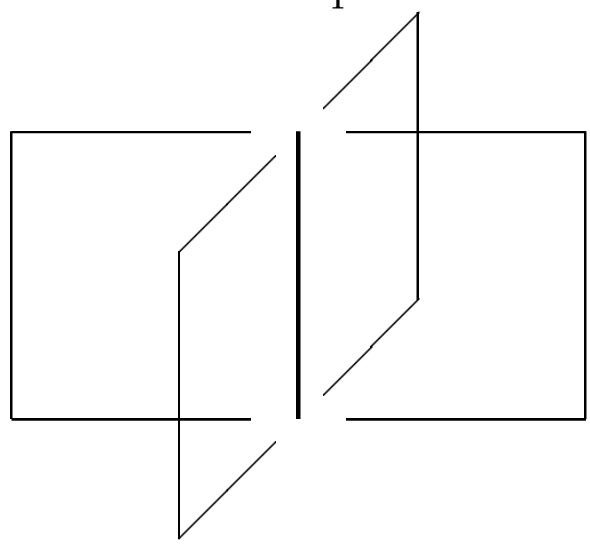
```
-- Energy difference between two spin states.
deltaE neighbs s s' = x
  where x :+ _ = - (conjugate (sum neighbs)) *
                   (s' - s)

-- 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)
```

19 Lattice gauge theory

The difference between $U(1)$ gauge theory (lattice QED) and the XY model is geometric. In gauge theory variables associated with links of the lattice, and the energy function is computed on the plaquettes.



Associate a group element U_P with any path P in the lattice by multiplying together the link variables along the path. Energy function is then

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

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

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

Replace `env` by

```
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

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])
```

and `colour` and `deltaE` by

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

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