15-150 Fall 2013
Lecture 20:
Sequences in practice

Stephen Brookes *

November 5, 2013

*Building on notes by Dan Licata

1 Sequences

Last time we introduced the signature `SEQ` and told you about a structure `Seq` that implements this signature, and provides parallel-friendly sequence types and operations.

```
signature SEQ =
sig
  type 'a seq
  exception Range
  val nth : int -> 'a seq -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val empty : unit -> 'a seq
  val map : ('a -> 'b) -> ('a seq -> 'b seq)
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
  val length : 'a seq -> int
  val cons : 'a -> 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
end;
```

For today, assume that we've `open`-ed this structure, so we don't have to use qualified names to get at the components. When developing code it might be preferable to use fully qualified names — this may help you keep track of where the components come from. But in class we're going to work within a clearly defined framework, and we'll use the more succinct notation.

In today's lecture we'll discuss some examples using sequences to solve some non-trivial problems. We won't have time in class to prove correctness and/or analyze the work and span of each function that we introduce. Some of these will be tasks in lab this week, and some may reappear on homework. We'd like you to get in the habit of sketching correctness proofs and work/span analysis as a matter of course! We did present work/span specs in the previous lecture for the main sequence operations, and we used them there to analyze a code fragment for summing integers in a sequence of integer sequences. You can use similar techniques here.

2 Gravitational simulation

We can use sequences to perform an *n-body simulation* of the effect of gravity on celestial bodies. Given the mass, position, and velocity of n celestial bodies, we want to simulate their movement over time due to gravitational forces. Such n-body simulations are used by astrophysicists — for example, their results provide evidence that there is a black hole in the center of the Milky Way. Similar ideas apply in other settings involving other forces; for example, Coulomb's law has a similar mathematical form to Nweton's law of gravitation, and is used in simulations of protein folding.

Here, we consider the n-body problem for 2 dimensions. The key step in the simulation is to compute the acceleration on a body due to the combined gravitational attraction of a collection of other bodies. With n bodies this requires computing a quadratic number ($O(n^2)$) of pairwise interactions. However, this problem is highly parallelizable, as the acceleration on one body due to one other body can be computed independently of the remaining bodies; and each body can compute its own combined gravitational acceleration in parallel.

(The following explanatory paragraphs are based on Wikipedia.)
Newton's law of universal gravitation states that every point mass in the universe attracts every other point mass with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them. (Separately it was shown that large spherically symmetrical masses attract and are attracted as if all their mass were concentrated at their centers.) This is a general physical law derived from empirical observations by what Newton called induction. (*Not what we call induction!*) It was formulated in Newton's work Philosophi Naturalis Principia Mathematica ("the Principia"), first published in 1687.

In modern language, the law states the following:

> A point mass attracts another point mass by a force directed along the line between the points. The magnitude of the force is proportional to the product of the two masses and inversely proportional to the square of the distance between them.

As a formula,

$$F = Gm_1m_2/r^2$$

where F is the magnitude of the force between the masses, G is a gravitational constant, m_1 is the first mass, m_2 is the second mass, and r is the distance between the centers of the masses.

Newton's Law of Motion says that the *acceleration* on a body with mass m due to a force of magnitude F (in some direction) is (a vector in the same direction, with magnitude) F/m. This is usually abbreviated as "force = mass times acceleration", or $F = ma$. So the acceleration on a body of mass m due to the gravitational attraction of another body of mass m_2 is

$$a = Gm_2/r^2$$

where r is the distance between the bodies. [1]

We're thinking of "vectors" here as pairs (x, y) of real numbers, and such a pair determines a "direction" (all lines with the slope y/x) and a magnitude (the square root of $x^2 + y^2$). For example, the vector $(1.0, 1.0)$ has slope 1.0, so is directed at 45% (north-west), and has magnitude $\sqrt{2}$.

We're also thinking of points as (x, y) pairs — the "x"-coordinate and the "y"-coordinate; the "origin" is the point $(0.0, 0.0)$.

Given a point (x, y) and a vector (dx, dy), we can *displace* the point along the vector and obtain a new point: this will be the point $(x + dx, y + dy)$.

Two vectors can be added together to produce a vector. In math we usually use + to mean either vector addition or real number addition, relying on the context to disambiguate. So in math we might write

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

for the sum of two vectors (x_1, y_1) and (x_2, y_2).

Given two points (a_1, b_1) and (a_2, b_2) the vector that represents the line from (a_2, b_2) to (a_1, b_1) is obtained by subtraction, and again using math notation we might write

$$(a_2, b_2) - (a_1, b_1) = (a_2 - a_1, b_2 - b_1).$$

With these ingredients in place, we now have enough to explain how to implement some simple ML functions for use in developing an n-body simulation.

[1] The gravitational acceleration on a body is independent of the body's mass, an observation usually attributed to Galileo. There is a (probably apocryphal) story about Galileo dropping balls of different masses off the top of the Leaning Tower of Pisa to demonstrate this, and contradicting Aristotle, who thought that more massive objects fall faster.

The gravitational constant G is approximately $6.67x10^{-11}N(m/kg)^2$. However, to keep the math simple we'll assume that G is 1.0. This amounts (basically) to scaling all of our data appropriately. We will also not bother to keep track of dimensions such as N(ewtons) and so on.

In developing our simulation we will ignore what happens when bodies collide; actually when bodies get very close in space other forces than gravity may come into effect. So we'll design our code so that when objects get "too close" we ignore gravity.

Points and vectors

As we said, we will represent points in the plane by pairs of real numbers. And we represent vectors as pairs of real numbers. The velocity and the acceleration of a body will be modeled as vectors, and will be used to compute the displacement to apply when "stepping" the universe for a single time step.

```
type point = real * real
type vect = real * real
```

Even though these two types are the same, it makes sense to use different names; values of type `point` and values of type `vect` play different rôles in the above narrative, and we want our ML implementation to be faithful to that description.

We need the following operations on points and vectors.

- Vectors have a zero, addition, and scalar multiplication:

  ```
  val zero : vect = (0.0 , 0.0)
  fun add ((x1,y1):vect, (x2,y2):vect):vect = (x1 + x2 , y1 + y2)
  fun scale(c:real, (x,y):vect):vect = (c * x , c * y)
  ```

- To compute the magnitude of a vector:

  ```
  fun mag ((x,y) : vect) : real = Math.sqrt (x * x + y * y)
  ```

- Given two points, we can compute the vector from the first to the second: (note the order of subtraction)

  ```
  fun diff ((x1,y1):point, (x2,y2):point) : vect = (x2 - x1, y2 - y1)
  ```

- Note that the relative distance of two points p and q, the distance between them, is simply

```
mag(diff(p, q))
```

- To displace a point by a vector:

```
fun displace ((x,y):point, (x',y'):vect) : point = (x + x', y + y')
```

- To check if two points are too close for comfort we'll use a parameter (tweakable, depending on the data), which we'll take as `0.1`, and say that points have collided when their relative distance is less than this parameter:

```
fun small (r : real) : bool = (Real.abs r < 0.1)
```

- In summary, we have the following operations on vectors:

```
type point = real * real
type vect = real * real
val diff: point * point -> vect
val small : real -> bool
val displace : point * vect -> point
val add : vect * vect -> vect
val scale : real * vect -> vect
val zero : vect
val mag : vect -> real
```

Representing celestial bodies

A body is represented by its position (a point), mass (a real number), and velocity (a vector).

```
(* body = position, mass, velocity *)
type body = point * real * vect
```

Accelerations

- The acceleration on body b due to the gravitational attraction of body b' is given by `accel` b b', as in:

  ```
  (* accel : body -> body -> vect *)
  fun accel (p1, _, _) (p2, m2, _) =
     let
       val d = diff(p1, p2)
       val r = mag d
     in
        if (small r) then zero else scale(G * m2/(r * r * r) , d)
     end;
  ```

  (See how the ML code corresponds clearly to the intuitive description given earlier. We scale `d` using the inverse cube of its magnitude `r`, because that produces the same result as a vector of magnitude $1/\mathtt{r}^2$. Note the use of wildcard patterns to indicate that the acceleration doesn't depend on the mass of the first body, as Galileo said. Also we have curried this function, to make it easy to use by partially applying it – to a given "first" body.)

- The acceleration on body b due to the collective action of the bodies in a sequence s is `accels` b s. We can calculate this vector by mapping `accel` b along s and adding the gravitational contributions of each body in the sequence. Rather than a map followed by a reduce, it's very simple to do a mapreduce:

  ```
  (* accels : body -> body seq -> vect *)
  fun accels b s = mapreduce (accel b) zero add s;
  ```

Moving

Suppose we have a body $b = (p, m, v)$, and we have computed the acceleration on b to be the vector a. For a small timestep dt we can approximate the motion of this point during this time step by assuming that the acceleration stays constant. Then, according to standard Newtonian mechanics, at the end of this time interval the body will have been displaced by the (vector) sum of $v\,dt$ and $\frac{1}{2}a\,dt^2$. (Here of course $v\,dt$ is the vector obtained by scaling v by dt, and $\frac{1}{2}a\,dt^2$ is the vector obtained by scaling a by $\frac{1}{2}dt^2$.) After this time step the body will have the same mass, a new position obtained by this displacement, and a new velocity obtained by adding $a\,dt$ to v. Hence,

```
(* move : body -> vect * real -> body *)
fun move (p, m, v) (a, dt) =
   let
      val p' = displace(p, add(scale(dt, v), scale(0.5 * dt * dt, a)))
      val v' = add(v, scale(dt, a))
   in
      (p', m, v')
   end;
```

And we can again use a higher-order function to do parallel movement of a collection of bodies:

```
(* step : body seq -> real -> body seq *)
fun step s dt = map (fn b => move b (accels b s, dt)) s;
```

An example

```
val sun = ((0.0,0.0), 332000.0,  (0.0,0.0));
val earth = ((1.0, 0.0), 1.0, (0.0,18.0));
```

Assume for simplicity that the sun is so much more massive than the Earth that we expect it to stay pretty much fixed in space while the Earth revolves in orbit. And we're interested in the trajectory of the Earth, i.e. a list of points that represent successive snapshots of the Earth's progression around its orbit, taken in successive time steps.

```
- val s : body seq = tabulate (fn 0 => sun | 1 => earth | _ => raise Range) 2;
val s = [((0.0,0.0),332000.0,(0.0,0.0)),((1.0,0.0),1.0,(0.0,18.0))]
     : body seq

- step s 0.01;
val it = [((5E~05,0.0),332000.0,(0.01,0.0)),((~15.6,0.18),1.0,(~3320.0,18.0))]
     : body seq
```

(Note that the sun has hardly moved, but the Earth has.)

We can generate orbits using:

```
(* orbit : body -> int * real -> point list *)
(* orbit b (n, dt) = n steps of motion of b under gravity of sun *)

fun orbit b (n, dt) =
   if n=0 then [ ] else
     let
       val (p', m, v') = move b (accel b sun, dt)
     in
       p' :: orbit (p', m, v') (n-1, dt)
    end;

orbit earth (10, 0.01);
val it =
  [(~15.6,0.18),(~48.7318019171,0.359213099043),
   (~81.7884162248,0.537587775754),(~114.835559608,0.71589462109),
   (~147.878962872,0.894177309445),(~180.920348361,1.07244756107),
   (~213.960467679,1.25071021688),(~246.999717285,1.42896774708),
   (~280.03833222,1.60722158368),(~313.076463412,1.78547263142)] : point list
```

Work/Span The work of `accels` is $O(n^2)$. The span is $O(\log n)$: the only chain of dependencies is adding up the components of the acceleration.

Thus, this algorithm is highly parallelizable. Where did this parallelism come from? First, from expressing the algorithm with maps and reduces on sequences, rather than lists. Second, from expressing the inner computations as mathematical calculations (in this case, calculations on real numbers), which are all independent of one another. We hope you can see the close correspondence between the math in question and the code, which is important both for parallelism, and for elegance/readability of the code.

Barnes-Hut In homework, we're going to have you implement a faster but approximate way of computing accelerations. The idea is to summarize the action of clusters of far-away bodies by replacing them with a single point at their center of mass. After all, it doesn't seem sensible to compute the interaction between the Earth and each star in the Andromeda Galaxy; it's much easier to pretend that the Andromeda Galaxy is a single point. This produces a less precise but faster simulation. The idea behind the Barnes-Hut algorithm is to divide space up into regions, and compute summary information (center of mass, total mass) for each region. When you are calculating the acceleration on a body due to bodies that are "far enough" away, you use the summary information. The work for this algorithm is $O(n \log n)$ (assuming a good choice of "far enough") instead of quadratic.