Repa Review

# File Structure

* Data/Array/
  + Repa.hs
  + Repa/
    - Arbitrary.hs
    - Index.hs
    - Slice.hs
    - Unsafe.hs
    - Base.hs
    - Eval.hs
    - Shape.hs
    - Stencil.hs
    - Eval/
      * Chunked.hs
      * Cursored.hs
      * Elt.hs
      * Gang.hs
      * Interleaved.hs
      * Load.hs
      * Reduction.hs
      * Selection.hs
      * Target.hs
    - Operators/
      * IndexSpace.hs
      * Interleave.hs
      * Mapping.hs
      * Reduction.hs
      * Selection.hs
      * Traversal.hs
    - Repr/
      * ByteString.hs
      * Delayed.hs
      * HintInterleave.hs
      * Partitioned.hs
      * Undefined.hs
      * Cursored.hs
      * ForeignPtr.hs
      * HintSmall.hs
      * Unboxed.hs
      * Vector.hs
    - Specialised/
      * Dim2.hs
    - Stencil/
      * Base.hs
      * Dim2.hs
      * Partition.hs
      * Template.hs

# Repa summary

Manifest, Delayed.

Use of computeP after fusing operations together.

# File summary

## Data/Array/Repa.hs

Provides all definitions – imports all modules and exports relevant identifiers.

Class Declarations

## Data/Array/Repa/Base.hs

Definition of class Source r e where:

* ‘r’ is a tag and indicates which instance to use in ./Repr/\*.hs
  + For example ‘Delayed.hs’ defines instance Source D …, ‘Unboxed.hs’ defines instance Source U …, and ‘Vector.hs’ defines instance Source V ….
* ‘e’ is the element type of the Array. Instinct says ‘e’ must be an instance of Elt but not sure how this ties in yet. Certain instances restrict element type. In particular, to store array elements unboxed U we need to know the width of each element and how to box them.”
* Stop press: e may also be the composed functions.

Source defines (amongst others):

* data Array r sh e – which adds an instance of class Shape sh defined in ./Shape.hs to the parameters of Source
* extent – provide the parameter sh (shape)
* index / ! and other variations (shape -> a)
  + Index takes a shape for the array structure and another shape denoting the position in the array. Shape can mean both the size of the array (Z :. 10 :. 10) and a position in the array (Z :. 5 :. 5) for the index of ~half the array.
* linearIndex (Int -> a)
* toList – Efficiency note: O(n). Iterates using index and cons elements together.

Instances of class Source are available in ./Repr/\*.hs

Source contains only operations that read (index et al) or consume (toList) an array.

## Data/Array/Repa/Shape.hs

Definition of class Eq sh => Shape sh where:

* ‘sh’ is an array dimension. Available dimensions take the form Z, Z :. Int, Z :. Int :. Int, … and are defined in Shape’s instance ./Index.hs
  + It is easiest to use the aliases: DIM0, DIM1, DIM2, DIM3, DIM4, DIM5 where the numeral suffix denotes dimension (DIM2 is a 2D array).

Shape defines (amongst others):

* rank – number of dimensions
* size – number of elements

## Data/Array/Repa/Eval/Load.hs

Definition of class (Source r1 e, Shape sh) => Load r1 sh e where:

* ‘r1’ is the array tag, one of D or C as only delayed arrays are applicable.
* ‘e’ is the element type
* ‘sh’ is the shape

Load defines:

* loadS – As loadP but sequentially.
* loadP – Pass a delay array with accumulated functions so that work can be chunked and given to threads.

As such, the Load class bridges the Source and Target. Functions are composed together and then when given an integer index a value will eventually be computed.

For example:

R.map g (R.map f (delay someArray))

will result in the following encapsulated in a delayed array:

g . f . fromIndex someArray

so that when a thread is given an integer range, for each integer it can yield a new value.

## Data/Array/Repa/Eval/Target.hs

Definition of class Target r e where

* ‘r’ is the array tag, one of U, F or V as certain manifest are supported.
* ‘e’ is the element type

This represents an array after computation has happened – the array has been constructed in parallel.

Target defines functions to create a manifest array.

This file also defines fromList.

Instances

## Data/Array/Repa/Index.hs

Sole instance of Shape.

Defines :., Z, DIM[0..5].

Defines ix[1..5]. These functions are useful to denote array size:

‘ix3 10 5 2’ is an alias for ‘Z :. (10 :: Int) :. (5 :: Int) :. (2 :: Int)’

## Data/Array/Repa/Repr/Unboxed.hs

Instance of a manifest Source / Target where the element type is an unboxed vector:

instance Data.Vector.Unboxed.Unbox a => Source U a

Underlying representation is as unboxed vectors.

data Array U sh a = AUnboxed !sh (Data.Vector.Unboxed.Vector a)

E.g. linearIndex simply calls out to Data.Vector.Unboxed.(!) and zip calls out to Data.Vector.Unboxed.zip

computeUnboxedP and fromListUnboxed are both aliases to Data.Array.Repa.Eval.computeP and Data.Array.Repa.Eval.Target.fromList respectively.

## Data/Array/Repa/Repr/Vector.hs

Instance of a manifest Source / Target where the element type is a boxed vector:

instance Source V a

Underlying representation is as boxed vectors.

data Array V sh a = AVector !sh !(Data.Vector a)

E.g. linearIndex calls out to Data.Vector.!

computeVectorP and fromListVector are both aliases to Data.Array.Repa.Eval.computeP and Data.Array.Repa.Eval.Target.fromList respectively.

## Data/Array/Repa/Repr/Delayed.hs

Instance of a delayed Source / Load.

Underlying representation is slightly different – as a function from the shape (indicies) to array elements.

Most interesting about Delayed.hs is the instance of Load. LoadP is defined to throw work at a chunker which throws the work at ganged threads to compute.

Operators

## Data/Array/Repa/Operators/IndexSpace.hs

Operators on Indicies to produce delayed arrays.

For example, to append two arrays, we can add the lowest rank dimension together (i+j), create a new array of that size and create a function that looks up the result in either the first or second array depending on whether the required index is 0 to i or i+1 to i+j.

Defines (amongst others):

* append – merge two arrays by increasing the size of the lowest rank dimension
* transpose – the last lowest two dimensions in the array (Z :. 2 :. 3) -> (Z :. 3 :. 2)
* extract – take from starting index to size of result elements from the array

## Data/Array/Repa/Operators/Mapping.hs

Map and Zip, producing delayed arrays.

## Data/Array/Repa/Operators/Reduction.hs

Fold variants which provide manifest arrays. These are wrapped in the IO Monad.

## Data/Array/Repa/Operators/Selection.hs

Implements a type of filter - selectP

Filter can be implemented as followed:

unsafePerformIO $ selectP f (linearIndex array) (size (extent array))

Scheduling

## Setup) Data/Array/Repa/Eval/Gang.hs

*A single, shared gang of threads is created at ‘start-up’. This is where the restriction of single level of parallelism comes from. The threads are instructed to ad nauseam: block awaiting an input function, compute that function and signal completion.*

A gang of threads is created. Each thread is created using GHC.Conc.forkOn and is given two Control.Concurrency.MVar’s for message passing communication. The MVars are retained ‘statically’ so that we can use primitives ‘putMVar’ and ‘takeMVar’ to communicate work to the threads. Work communicated is of the form (ThreadID :: Int -> IO ()).

Each thread:

1. Blocks on workRequest MVar
2. Evaluates work, passing its threadID
   * work includes writing to final resultant mutable vector, so the thread doesn’t need to communicate chunks of work back
3. Signal that we have completed the work
4. Continue from (1)

Uses GHC.Conc.numCapabilities (“the value passed to the +RTS –N flag”) as the number of threads to fork.

## 1) Data/Array/Repa/Repr/Eval.hs

*Computation of arrays is kickstarted by computeP. It turns a Delayed Array into a Manifest Array.*

A new mutable buffer is created to store the contents of the array.

## 2) Data/Array/Repa/Repr/Delayed.hs

*A Delayed Array is represented as a function from index to result. When array values are demanded we have to apply the function over every index. This can be done in parallel. After computeP, the next step in the chain is loadP.*

A delayed representation of a manifest array ‘arr’ might be Array D (extent arr) (index arr) – when given a index, we merely return that index of the original array.

When loadP is given a buffer, we pass on the number of elements in the new array, a function to write into the buffer when given an index and a value, and a function that yields a resultant value when given an index.

## 3) Data/Array/Repa/Eval/Chunked.hs

*Take the function that ‘computes a result and writes into a new array, given an index’ and groups the application of this function over a range of indicies to each thread. Each thread gets a chunk of indicies to evaluate. As such we have a new function, ThreadID -> work over index range.*

We know the number of elements to be computed and the number of threads. We can then work out an index range that each thread should be assigned to and convert the (index -> write result) function to a (thread index -> write results) function.

Now that we have a thread relevant function, we can call functions relative to thread gangs.

## 4) Data/Array/Repa/Eval/Gang.hs

*Threads have already been forked and each one is waiting for an MVar write of a (ThreadID -> Work) function. We can now communicate this function to every thread so that computation can actually be performed in parallel, merely by applying the thread ID.*

A check is made to ensure we are not doing nested parallelism (the threads will already be working).

Then, every thread is sent the function via MVar mechanism. Every thread will then evaluate the function, resulting every entry in the new array buffer being filled in by the function being evaluated by every index.

We await all thread signals and then carry on as usual, free to use the new array.