Repa 3 tutorial for curious haskells

Daniel Eddeland
d.skalle@gmail.com

Oscar Söderlund
soscar@student.chalmers.se

1 What is Repa?

Repa is a library for Haskell which lets programmers perform operations in parallel on regular arrays. The name Repa comes from REgular PArallel arrays. "Regular arrays" mean that the arrays are dense, rectangular, and every element in the same Repa array must be the same type.

1.1 Repa Arrays

Arrays in Repa have the type

```
Array r sh a
```

Here the sh stands for shape parameter, i.e. the size and dimensionality of the array. The type `a` stands for the type of element that is stored in the array, for instance `Int`. The parameter `r` stands for the type of the array representation. For example, one array could have the type:

```
Array U (Z :. 4 :. 5) Double
```

This would then be a 4x5 two-dimensional array of Doubles. The U here stands for Unboxed manifest array; unboxed means that the value of every element is calculated and stored (so it doesn't use lazy evaluation for elements).

1.2 Manifest arrays and delayed arrays

In Repa it is important to distinguish between manifest arrays and delayed arrays. Of the two, manifest arrays are the most simple; the concrete value of every element has been computed and is stored in memory. The following is the type of a simple, one-dimensional manifest arrays of Ints:

```
Array U DIM1 Int
```

As mentioned above, the U stands for unboxed manifest array.

Delayed arrays are a bit different; a delayed array may not have actually calculated every value before an element is retrieved. The values are instead represented by functions that are computed every time an element needs to be accessed. Therefore, if values of an array are used repeatedly, it might be a good idea to convert that array to manifest so that calculations need not be

repeated. On the other hand, having a delayed array may be useful for list fusion; we might want to perform several functions in order on every element of a list. In this case it might be beneficial to have a delayed array, so that we don't need to actually compute concrete values until we have our "final" array. Therefore Repa allows us to use delayed arrays, and we can convert delayed arrays to manifest using the `computeP` monadic function.

Delayed arrays are simply represented by the letter D, so the following could be the type of a one-dimensional delayed array.

```
Array D DIM1 Int
```

Some basic repa operations may give a delayed array as a return type, which means we will have to convert to manifest whenever we need or want to. As an example, the Repa `map` function has the following type:

```
map :: (Shape sh, Source r a) =>
       (a -> b) -> Array r sh a -> Array D sh b
```

The map function creates a new array by applying a function to every element of an array. The input array may be of any type (either manifest or delayed), but the resulting array will be of delayed type (shown by the D). The functionality is as expected similar to its Prelude counterpart (which applies to normal Haskell arrays).

1.3 Indexing

In Repa, the Shape type is used both to define the bounds of an array, and is also used when indexing in an array. For instance, let's consider a simple one-dimensional array of integers, of length 4. This array has the shape `(Z :. 4)`, i.e. the type for the array could be:

```
Array U (Z :. 4) Int
```

Here the Z represents the zero dimension, and the 4 after it represents the size of the first dimension. Similarly, a 3x3 two-dimensional array has the shape:

```
(Z :. 3 :. 3)
```

We can use Shapes for indexing as well. We can get the element at a certain index in a repa array using the indexing function `!`, with indices starting at 0. If `arr` is a one-dimensional array of ints of length 4 we can get the first element using:

```
arr ! (Z :. 0)
```

And likewise, we can get the last element using `arr ! (Z :. 3)`. Note that if we try to access the array out of bounds, we will get a runtime exception.

We can also use shapes to define n-dimensional arrays of arbitrary size. For instance, the type

```
(Z :. Int)
```

Repa has defined type synonyms for n-dimensional arrays, for n up to 5. For instance

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
```

Repa also defines help functions for creating indices from Ints. For instance, we can use the function ix1 to create a one-dimensional shape of a given size:

```
arr ! (ix1 0)
```

2 Using Repa

Before using Repa, be sure to install the library and import it into your Haskell file. It may also be helpful to use qualified imports since otherwise some functions may conflict with prelude functions (for instance the zipWith function).

2.1 Creating arrays

Repa arrays can be created from normal Haskell lists using the function $fromListUnboxed$. For instance:

```
larr = fromListUnboxed (ix1 8) [1..8]
```

Delayed arrays can also be constructed directly from functions using the $fromFunction$ function, which takes as parameters a shape (i.e. size of the array) and a function which maps every index to a value.

```
farr = fromFunction (ix1 4) $ \i -> (size i) ^ 2
```

The first argument here is simply a one-dimensional shape of size 4. The second argument is a function which takes a shape (called i) and then converts that shape to an integer index, and finally squares it. This means that `farr` will be the array `[0, 1, 4, 9]` (i.e. `[0..4]` squared).

2.2 Basic functions

Repa contains basic functions that can be used to simply perform operations on arrays. For instance, the following array functions exist which are analogous to the Prelude functions with the same name:

- `map`, which takes a function and a Repa array, and produces a delayed array by applying the function to every element of the array.
- `zipWith`, which takes a binary operator and two Repa arrays of the same size, and procudes a delayed array by applying the operator to indices of the input arrays.
- `foldAllP` (analogous to `foldl`), which combines all the values of an array to yield a single value. This can for instance be used to calculate the sum over an array of integers.

Here is an example of a small program using map.

```
main :: IO ()
main = do
  let arr = fromListUnboxed (ix1 4) [1..4]
      arr' = R.map (*3) arr
      arr'' <- computeP arr' :: IO (Array U DIM1 Int)
  print arr''
```

First we create an unboxed array containing `[1..4]`. Then we map the function (∗3) to multiply each element with 3, which yields a delayed array. Finally we use computeP to convert the array to a manifest array; we do this in order to be able to print it. The printed array value will then be

```
[3,6,9,12]
```

Another helpful Repa function is traverse, which can traverse an existent array and create a new array.

```
R.traverse :: (Shape sh', Shape sh, Source r a) =>
    Array r sh a ->
    (sh -> sh') ->
    ((sh -> a) -> sh' -> b) ->
    Array D sh' b
```

We see that traverse takes as input arguments

1. A source array
2. A function that transforms the shape of the array
3. A function which takes as argument an indexing function to the original array, an index in the new array, and computes the value at the index in the new array.

To demonstrate how traverse works, we implement pairwise addition; i.e. take as input a one-dimensional array and output an array of half its size that has summed together elements pairwise

```
pairwiseAdd :: Array U DIM1 Int -> Array D DIM1 Int
pairwiseAdd arr = traverse arr halfsize indexfun
  where halfsize (Z :. i) = (Z :. i `div` 2)
        indexfun ixf (Z :. i) =
                  (ixf $ ix1 $ 2*i) + (ixf $ ix1 $ 2*i+1)
```

3 Stencil convolution

Anyone who has dabbled in Photoshop, or any other image processing application, knows that the need often arises to manipulate images by applying filters. Such filters could for example be used to blur or change the color or lightness of an image. Applying such a filter to an image is a joy to do in Repa, and we are about to understand why.

3.1 What is a stencil?

An image processing filter is in most cases a linear function that is mapped over every pixel in the processed image. Every pixel is updated to a linear combination of itself and its neighbors.

The linear equation for how every pixel is combined with its neighbors can be described using a matrix of coefficients. The center element of the matrix is the coefficient of the current pixel being updated, and the neighboring elements are the coefficients for the neighboring pixels.

Another name for this matrix of coefficients is a stencil, and to applying this stencil to a pixel is called a stencil convolution. An example of such a stencil can be seen in figure 1.

```
0.013 0.025 0.031 0.025 0.013
0.025 0.057 0.075 0.057 0.025
0.031 0.075 0.094 0.075 0.031
0.025 0.057 0.075 0.057 0.025
0.013 0.025 0.031 0.025 0.013
```

Figure 1: An example stencil.

To apply a filter to an image is equivalent to applying the stencil of the filter to every pixel in the image. This means that the new value for every pixel can be calculated in parallel.

Now we start to suspect that Repa is a good fit for calculating stencil convolutions. Maybe we could express the stencil as a function, and use Repa's mapping combinators to do it in parallel? We just have to figure out a way to express stencils as functions.

As we are about to find out, Repa can do this for us. Let us then dive straight into exploring the tools that Repa provides for us to specify and apply stencil convolutions.

3.2 Stencils in Repa

This section includes code that is heavily inspired by the official Repa examples. Anyone who is interested in learning more about stencils should definitely check them out!

```
cabal install repa-examples
```

3.2.1 Specifying stencils

One possible way of describing a stencil is to define a function that maps a relative index to the coefficient that belongs to that index. For example, the very simple stencil in figure 2 can be described using the function in 3:

```
0 1 0
1 0 1
0 1 0
```

Figure 2: A simple example stencil.

```
makeStencil2 (Z :. 3 :. 3)
(\ix -> case ix of
          Z :. -1 :.  0  -> Just 1
          Z :.  0 :. -1  -> Just 1
          Z :.  0 :.  1  -> Just 1
          Z :.  1 :.  0  -> Just 1
          _              -> Nothing)
```

Figure 3: A lambda modelling the simple stencil in figure 2.

We see that Repa provides the function `makeStencil2`, for specifying a 2-dimensional stencil of a given dimension, here 3 by 3. Since this tutorial focuses on image manipulation, we will only concern ourselves with the special case of 2-dimensional stencils.

This way of making stencils from lambdas is however somewhat cumbersome. It is not immediately apparent from looking at the lambda in 3 that it actually represents the stencil in figure 2. To remedy this, Repa provides us with the convenience method `stencil2`, which allows us to specify stencils like in figure 4.

```
[stencil2| 0 1 0
           1 0 1
           0 1 0 |]
```

Figure 4: A stencil defined using the QuasiQuotes language extension.

If this syntax looks foreign, do not fret. Just know that it exploits a language extension called `QuasiQuotes`, and that by putting the pragma:

```
{-# LANGUAGE QuasiQuotes #-}
```

at the top of your source file, the code in figure 4 will be preprocessed and converted into the code in figure 3 during compile time.

3.2.2 Applying stencils

So now that we know how to specify stencils, it only remains to apply them to something. Let us keep working with our simple stencil from the previous section, and see what happens when we apply it to an actual image.

Repa allows us to read bitmaps into arrays using the `readImageFromBMP` function, which reads a bitmap from a relative file path and produces a two-dimensional

array of `Word8` triples. Each of these triples represents a pixel in the source image, and the individual words in these triples represent the red, green and blue components of the pixel.

Figure 5: Example image.

To read in the image in figure 5, the following code is sufficient:

```
do
  arrImage <- liftM (either (error . show) id) $
                readImageFromBMP "image.bmp"
```

This gives us an array of `Word8` triples. We use the run function for the `Either` monad to handle the case when reading the image results in an error.

```
makeStencil2 :: Num a =>
  Int -> Int ->
  (DIM2 -> Maybe a) ->
  Stencil DIM2 a

mapStencil2 :: Source r a =>
  Boundary a ->
  Stencil DIM2 a ->
  Array r DIM2 a ->
  Array PC5 DIM2 a
```

If we look at the types of makeStencil2 and the corresponding mapStencil2, we see that the type of the array we map our stencil over must contain elemnts of the same type as we specified in our stencil. We also see that mapping a stencil requires specifying what happens in the boundaries of the array. In our case, it is sufficient to use `(BoundConst 0)`, which means that we pretend that there are zeroes everywhere outside the boundaries of our array.

To make our integer stencil work for arrays of `Word8`:s, we will unzip the array of triples into three separate arrays of numbers. After acquiring three separate arrays containing the red, blue and green components, we will map our stencil on these arrays separately as such:

```
do
  (r, g, b) <- liftM (either (error . show) R.unzip3) readImageFromBMP "in.bmp"
  [r', g', b'] <- mapM (applyStencil simpleStencil) [r, g, b]
  writeImageToBMP "out.bmp" (U.zip3 r' g' b')
```

After acquiring our transformed component arrays, we promptly zip them back

into an image and write it to a bitmap using the Repa provided `writeImageToBMP`, resulting in the image in figure 6.

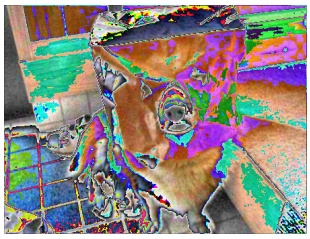

Figure 6: The stencil in figure 2 applied to the image in figure 5.

Our filter from the stencil in figure 2 has successfully spiced up the colors of our example image.

Let us now look at a couple of example stencils that actually result in something useful.

3.3 Some examples

Now that we know how to handle stencil convolution in Repa, let's try it out on a couple of real examples.

We are going to experiment with two fundamental image filters and their corresponding stencils, Gaussian blur and simple edge detection.

In order to easily experiment with different filters and stencils, let us use the primitives provided by Repa to define a couple of useful combinators.

First off, we represent our images as 2-dimensional arrays of numerals. However, the `Word8` type in which images are represented when read from files does not lend itself very well to non-integer arithmetic, so let us define images as arrays of floating point numbers, and a way to easily switch between representations.

Converting between different numeric representations in Haskell can be a real pain, but fortunately the authors of Repa have provided us with a way of doing it in their example files:

```
type Image = Array U DIM2 Double

promote :: Array U DIM2 Word8 -> IO (Image)
promote = computeP . R.map ffs
 where
ffs :: Word8 -> Double
ffs x = fromIntegral (fromIntegral x :: Int)

demote :: Image -> IO (Array U DIM2 Word8)
demote = computeP . R.map ffs
 where
ffs :: Double -> Word8
```

```
ffs x = fromIntegral (truncate x :: Int)
```

Now we can define a filter as a function from an image to a new image, and a way of creating filters by applying stencils:

```
type Filter = Image -> IO (Image)

applyStencil :: Stencil DIM2 Double -> Filter
applyStencil stencil = computeP . mapStencil2 (BoundConst 0) stencil
```

We use Repas `computeP` combinator to make sure that every filter is applied to individual pixels in parallel. This is why the resulting image from a filter is wrapped in the `IO` monad.

We can now use `applyStencil` to define our first filter. As it turns out, our example matrix from figure 1 is actually a stencil representing Gaussian blur with a 3-pixel radius.

Even though we represent images using floating point numerals, we must still use integers when specifying stencils using our special `stencil2` shorthand function. We can make an equivalent stencil by using a stencil of integers and a subsequent normalization filter:

```
gaussStencil :: Stencil DIM2 Double
gaussStencil =
  [stencil2| 2  4  5  4  2
             4  9 12  9  4
             5 12 15 12  5
             4  9 12  9  4
             2  4  5  4  2 |]

normalize :: Double -> Filter
normalize n = computeP . R.map (/ n)

gaussFilter :: Filter
gaussFilter = applyStencil gauss >=> normalize 159
```

Notice again the type of filter:

```
type Filter = Image -> IO (Image)
```

Now take a look at the type of the general monadic composition operator:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

Isn't that just a perfect fit for a filter composition combinator? Maybe that Kleisli guy is not so bad after all!

Before trying out our Gauss filter, it only remains to formalize the glue code from the previous section into an appropriate image processing combinator, carefully making sure that we please the type system by gently squeezing our filter in between our numeric coercion functions:

```
processImage :: String -> String -> Filter -> IO ()
processImage infile outfile filter = do
  (r, g, b) <- liftM (either (error . show) U.unzip3) (readImageFromBMP infile)
```

```
  [r',g',b'] <- mapM (promote >=> filter >=> demote) [r,g,b]
  writeImageToBMP outfile (U.zip3 r' g' b')
  return ()
```

We can now observe the result of :

```
processImage "fish.png" "fish_blurry.png" gaussFilter
```

Figure 7: Original image.

Figure 8: Image with one pass of gaussian blur filter applied.

That was not very impressive. Did it even work? Lets define another combinator for applying a filter multiple times and see:

```
passes :: Int -> Filter -> Filter
passes 1 filter = filter
passes n filter = filter >=> (passes (n-1) filter)
```

We can see the result of:

```
processImage "fish.png" "fish_blurrier.png" (10 `passes` gaussFilter)
```

Figure 9: Image with 10 passes of gaussian blur filter applied.

That's more like it. And now that we have a nifty little DSL for creating filters from stencils and applying them to images, there is no stopping us!

Another interesting activity in image processing is detecting edges. A very simple strategy for this is to apply the following stencil to an image:

```
edgeStencil :: Stencil DIM2 Double
edgeStencil =
  [stencil2| 0  1  0
             1 -4  1
             0  1  0 |]
```

We see that the resulting pixel will be black if it has the same color as all of its neighbors, which when applied to an entire image will result in enhanced edges. Lets try and apply it to an image:

Figure 10: Original image.

Figure 11: Image with edge detection filter applied.

That looks pretty strange. What happened to handsome Mr. Haskell? It seems like there is too much noise in the image for the edge detection filter to actually produce something useful. What a coincidence then that we just happen to have a noise reducing blur filter lying around.

We see that the result of applying 3 passes of gaussian blur before the edge filter is much more to our satisfaction:

```
3 `passes` gaussFilter >=> edgeFilter
```

Figure 12: Image with 3 passes of gaussian blur and edge detection filter applied.

3.4 Stencil benchmarking

So now we know how to use Repa:s stencil primitives to define image filters, but do we actually get any benefit from the parallelism?

This is our results from benchmarking our gaussian filter:

```
processImage "fish.bmp" "fish_blurrier.bmp" (n `passes` gaussFilter)
```

```
benchmarking 1 passes of gauss, 1 core
mean: 6.401326 s, lb 6.389732 s, ub 6.419607 s, ci 0.950
std dev: 18.87974 ms, lb 7.724321 ms, ub 27.25800 ms, ci 0.950

benchmarking 4 passes of gauss, 1 core
mean: 21.86578 s, lb 21.85245 s, ub 21.88628 s, ci 0.950
std dev: 21.50901 ms, lb 9.319194 ms, ub 30.88440 ms, ci 0.950

benchmarking 1 passes of gauss, 2 cores
mean: 3.746440 s, lb 3.739277 s, ub 3.756456 s, ci 0.950
std dev: 10.73748 ms, lb 5.337870 ms, ub 16.15573 ms, ci 0.950

benchmarking 4 passes of gauss, 2 cores
mean: 13.20408 s, lb 12.90553 s, ub 13.82378 s, ci 0.950
std dev: 510.8692 ms, lb 51.16636 ms, ub 671.8831 ms, ci 0.950
```

We see an almost double speedup when using two cores instead of one.

What a glorious day to be a Haskell programmer!