# Data-parallel arrays

## The Practice of Haskell Programming

Andres Löh

Well-Typed

May 17, 2012

# The plan for today

- Unboxed types (type internals, prerequisite).
- The Repa library.

# Unboxed types

# The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

# The internals of basic types

```
〉:i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

# The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

What is an `Int#`?

# The internals of basic types – contd.

To get names like `Int#` even through the parser, we have to enable the `MagicHash` language extension …

```
⟩ :i GHC.Prim.Int#
data GHC.Prim.Int#    -- Defined in ‘ GHC.Prim ’
```

So this one seems to be really primitive.

# Boxed vs. unboxed types

The type `Int#` is the type of **unboxed** integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

# Boxed vs. unboxed types

The type `Int#` is the type of **unboxed** integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

An `Int` is a **boxed** integer:

- it wraps the unboxed integer in an additional pointer,
- thereby introducing an indirection.

# Boxed vs. unboxed types – contd.

Pro unboxed:

- no indirection,
- faster,
- less space.

# Boxed vs. unboxed types – contd.

Pro unboxed:

- no indirection,
- faster,
- less space.

Pro boxed:

- only boxed types admit laziness,
- only boxed types admit polymorphism.

Boxing makes all types look alike, making it compatible with thunks and polymorphisms.

# Operations on unboxed types

Everything is monomorphic:

```
3#      :: Int#
3##     :: Word#
3.0#    :: Float#
3.0##   :: Double#
'c'#    :: Char#

(+#)       :: Int#    → Int#    → Int#
plusWord#  :: Word#   → Word#   → Word#
plusFloat# :: Float#  → Float#  → Float#
(+##)      :: Double# → Double# → Double#
```

# The kind of unboxed types

GHC uses Haskell's **kind** system to distinguish boxed from unboxed types:

```
⟩ :k Int
Int :: *
⟩ :k []
[] :: * → *
⟩ :k Int#
Int# :: #
```

- Kinds are the types of types.
- Just like programs are type-checked, they're also kind-checked.
- You can get kind errors.

# Kind errors

All these expressions produce kind errors:

```
〉 let x = undefined :: []
〉 3# +# 2
〉 id 3#
〉 [3#]
```

# Unpacking strict fields

You typically don't have to use unboxed types directly:

```
data X = C ... {-# UNPACK #-} !Int ...
```

If you have a strict, single-constructor field in a datatype, then the "unpack" pragma instructs GHC:

- to avoid the indirection introduced by the constructor,
- thereby in this case inlining the unboxed `Int#` inside.

# Repa

# Introducing Repa

A library for data-parallelism in Haskell:

- implemented as an EDSL,
- based on adaptive unboxed arrays,
- offers “delayed” arrays,
- arrays can be re-shaped,
- makes use of advanced type system features,
- offers high-level parallelism.

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;
- the final is the element type;

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;
- the second the **shape**.

# Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;
- the second the **shape**.

But what are **representation** and **shape**?

# Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index**.

# Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index**.

```
data Z = Z                -- similar to the  ()  type,  Z  for "zero"
data t :. h = !t :. !h    -- similar to  (,) , but strict
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
...
```

# Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index**.

```
data Z = Z                -- similar to the () type, Z for "zero"
data t :. h = !t :. !h    -- similar to (,), but strict
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
...
```

So `DIM2` is the type of strict pairs of integers.

# Array representations

Repa distinguishes two fundamentally different states an array can be in:

# Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;

# Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;
- a **delayed** array is not a real array at all, but merely a computation that describes how to compute each of the elements.

# Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;
- a **delayed** array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Let's look at the "why" and the delayed representation in a moment.

The standard **manifest** representation is denoted by a type argument `U` (for unboxed).

# Creating manifest arrays

$$\mathsf{fromListUnboxed} \\ \quad :: (\mathsf{Shape\ sh}, \mathsf{Unbox\ a}) \Rightarrow \mathsf{sh} \rightarrow [\mathsf{a}] \rightarrow \mathsf{Array\ U\ sh\ a}$$

# Creating manifest arrays

```
fromListUnboxed
  :: (Shape sh, Unbox a) ⇒ sh → [a] → Array U sh a
```

Example:

```
⟩ fromListUnboxed (Z :. 10 :: DIM1) [1 .. 10 :: Int]
AUnboxed (Z :. 10) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
⟩ fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
AUnboxed ((Z :. 2) :. 5) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The shape argument provides the dimensions and size of the array; the list must match the size of the shape:

```
⟩ size (Z :. 2 :. 5 :: DIM2)
10
```

# The Unbox class

The `fromListUnboxed` function creates an **adaptive unboxed** array.

The `Unbox` class is defined in the `vector` package:

```
class Unbox a
instance Unbox Int
instance Unbox Float
instance Unbox Double
instance Unbox Char
instance Unbox Bool
instance (Unbox a, Unbox b) ⇒ Unbox (a, b)
```

- Choose an efficient representation depending on element type.
- Represent arrays of tuples as tuples of arrays.

# What if our type is not in `Unbox`?

Two options:

- define an `Unbox` instance (tedious, but generally possible);
- use a less efficient manifest array representation ( `V` ).

For the purposes of this lecture, base types and `U` are sufficient.

# Array access

$$
\begin{array}{ll}
\mathsf{extent} & :: (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Repr}\ \mathsf{r}\ \mathsf{e}) \Rightarrow \mathsf{Array}\ \mathsf{r}\ \mathsf{sh}\ \mathsf{e} \rightarrow \mathsf{sh} \\
(!) & :: (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Repr}\ \mathsf{r}\ \mathsf{e}) \Rightarrow \mathsf{Array}\ \mathsf{r}\ \mathsf{sh}\ \mathsf{e} \rightarrow \mathsf{sh} \rightarrow \mathsf{e}
\end{array}
$$

# Array access

```
extent :: (Shape sh, Repr r e) ⇒ Array r sh e → sh
(!)    :: (Shape sh, Repr r e) ⇒ Array r sh e → sh → e
```

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
```

# Array access

```
extent :: (Shape sh, Repr r e) ⇒ Array r sh e → sh
(!)    :: (Shape sh, Repr r e) ⇒ Array r sh e → sh → e
```

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
```

```
⟩ extent example
(Z :. 2) :. 5
⟩ x ! (Z :. 1 :. 3)
9
```

# The Repr class

The class Repr keeps track which element types are allowed for which representation:

```
class Repr r e
instance Unbox a ⇒ Repr U a
instance           Repr V a
```

The unboxed representation is only valid for elements in the Unbox class.

# Operations on arrays

$$
\begin{array}{lll}
\mathsf{map} & :: & (\mathsf{Shape}\ sh, \mathsf{Repr}\ r\ a) \Rightarrow \\
 & & (a \rightarrow b) \rightarrow \mathsf{Array}\ r\ sh\ a \rightarrow \mathsf{Array}\ \mathsf{D}\ sh\ b \\
\mathsf{extract} & :: & (\mathsf{Shape}\ sh, \mathsf{Repr}\ r\ e) \Rightarrow \\
 & & sh \rightarrow sh \rightarrow \mathsf{Array}\ r\ sh\ e \rightarrow \mathsf{Array}\ \mathsf{D}\ sh\ e \\
(+\!\!+) & :: & (\mathsf{Shape}\ sh, \mathsf{Repr}\ r1\ e, \mathsf{Repr}\ r2\ e) \Rightarrow \\
 & & \mathsf{Array}\ r1\ (sh :. \mathsf{Int})\ e \rightarrow \mathsf{Array}\ r2\ (sh :. \mathsf{Int}) \rightarrow \\
 & & \mathsf{Array}\ \mathsf{D}\ (sh :. \mathsf{Int})\ e \\
(*\hat{}) & :: & (\mathsf{Num}\ c, \mathsf{Shape}\ sh, \mathsf{Repr}\ r1\ c, \mathsf{Repr}\ r2\ c) \Rightarrow \\
 & & \mathsf{Array}\ r1\ sh\ c \rightarrow \mathsf{Array}\ r2\ sh\ c \rightarrow \mathsf{Array}\ \mathsf{D}\ sh\ c
\end{array}
$$

Note:

- What does the shape requirement on $(+\!\!+)$ tell us?
- All these functions return **delayed** arrays ( D ).

# Why delayed arrays?

Recall "map fusion":

$$(\text{map } f \circ \text{map } g)\ xs \equiv \text{map } (f \circ g)\ xs$$

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.

# Why delayed arrays?

Recall "map fusion":

$$(\mathrm{map}\ f \circ \mathrm{map}\ g)\ xs \equiv \mathrm{map}\ (f \circ g)\ xs$$

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.
- However, lists can be traversed one by one. Even if we don't fuse the computations, we only allocate the intermediate cons-cells for the cons-cells we evaluate in the end.

# Why delayed arrays?

Recall "map fusion":

```
(map f ∘ map g) xs == map (f ∘ g) xs
```

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.
- However, lists can be traversed one by one. Even if we don't fuse the computations, we only allocate the intermediate cons-cells for the cons-cells we evaluate in the end.
- For arrays, we have to make a full intermediate copy for every traversal, so performing fusion becomes essential – so important that we'd like to make it **explicit** in the type system.

# Delayed arrays

Delayed arrays are internally represented simply as functions:

$$\textbf{data instance}\ \text{Array D sh e} = \text{ADelayed}\ !\text{sh}\ (\text{sh} \rightarrow \text{e})$$

- Delayed arrays aren't really arrays at all.
- Operating on an array does not create a new array.
- Performing another operation on a delayed array just performs function composition.
- If we want to have a manifest array again, we have to **explicitly force** the array.

# Creating delayed arrays

From a function:

$$\text{fromFunction} :: \text{sh} \rightarrow (\text{sh} \rightarrow \text{a}) \rightarrow \text{Array D sh a}$$

Directly maps to `ADelayed`.

From an arbitrary Repa array:

$$\text{delay} :: (\text{Shape sh}, \text{Repr r e}) \Rightarrow \text{Array r sh e} \rightarrow \text{Array D sh e}$$

# The implementation of `map`

```
map :: (Shape sh, Repr r a)
    ⇒ (a → b) → Array r sh a → Array D sh b
map f arr = case delay arr of
              ADelayed sh g → ADelayed sh (f ∘ g)
```

# The implementation of `map`

$$
\begin{aligned}
&\mathsf{map} :: (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Repr}\ \mathsf{r}\ \mathsf{a}) \\
&\qquad \Rightarrow (\mathsf{a} \to \mathsf{b}) \to \mathsf{Array}\ \mathsf{r}\ \mathsf{sh}\ \mathsf{a} \to \mathsf{Array}\ \mathsf{D}\ \mathsf{sh}\ \mathsf{b} \\
&\mathsf{map}\ \mathsf{f}\ \mathsf{arr} = \mathbf{case}\ \mathsf{delay}\ \mathsf{arr}\ \mathbf{of} \\
&\qquad\qquad\qquad \mathsf{ADelayed}\ \mathsf{sh}\ \mathsf{g} \to \mathsf{ADelayed}\ \mathsf{sh}\ (\mathsf{f} \circ \mathsf{g})
\end{aligned}
$$

Many other functions are only slightly more complicated:

- think about pointwise multiplication `(*^)`,
- or the more general `zipWith`.

# Forcing delayed arrays

Sequentially:

$$\begin{aligned}\mathsf{computeS} :: {} & (\mathsf{Fill\ r1\ r2\ sh\ e}) \Rightarrow \\ & \mathsf{Array\ r1\ sh\ e} \rightarrow \mathsf{Array\ r2\ sh\ e}\end{aligned}$$

# Forcing delayed arrays

Sequentially:

$$\text{computeS} :: (\text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{Array r2 sh e}$$

In parallel:

$$\text{computeP} :: (\text{Monad m}, \text{Repr r2 e}, \text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{m (Array r2 sh e)}$$

# Forcing delayed arrays

Sequentially:

$$\text{computeS} :: (\text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{Array r2 sh e}$$

In parallel:

$$\text{computeP} :: (\text{Monad m}, \text{Repr r2 e}, \text{Fill r1 r2 sh e}) \Rightarrow \\ \text{Array r1 sh e} \rightarrow \text{m (Array r2 sh e)}$$

The `Fill` class encodes which representations can be converted into which others. The interesting case is:

```
instance (Unbox e, Shape sh) ⇒ Fill D U sh e
```

# “Automatic” parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don’t have to concern the user.

# “Automatic” parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don’t have to concern the user.
- But: Repa **only** supports **flat** data-parallelism! If the delayed computations forced by `computeP` are themselves parallel, Repa will fall back to sequential computation.

# Reducing arrays

Reductions or folds are also available in both sequential and parallel variants:

```
sumS    :: (Num a, Shape sh, Repr r a, Unbox a, Elt a) ⇒
           Array r (sh :. Int) a → Array U sh a
sumP    :: (Monad m, Num a, Shape sh, Repr r a, Unbox a, Elt a) ⇒
           Array r (sh :. Int) a → m (Array U sh a)

sumAllS :: (Num a, Shape sh, Repr r a, Unbox a, Elt a) ⇒
           Array r sh a → a
sumAllP :: (Monad m, Num a, Shape sh, Repr r a, Unbox a, Elt a) ⇒
           Array r sh a → m a

foldS   :: (Shape sh, Repr r a, Unbox a, Elt a) ⇒
           (a → a → a) → a → Array r (sh :. Int) a → Array U sh a
foldP   :: (Monad m, Shape sh, Repr r a, Unbox a, Elt a) ⇒
           (a → a → a) → a → Array r (sh :. Int) a → m (Array U sh a)
```

The constraint `Elt` is comparable to `Unbox`.

# Examples

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5) [1 .. 10]
```

```
〉 computeS (map (+ 1) example) :: Array U DIM2 Int
AUnboxed ((Z :. 2) :. 5) (fromList [2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
〉 computeUnboxedS (extract (Z :. 0 :. 1) (Z :. 2 :. 3) example
AUnboxed ((Z :. 2) :. 3) (fromList [2, 3, 4, 7, 8, 9])
〉 sumS it
AUnboxed (Z :. 2) (fromList [9, 24])
〉 sumS it
AUnboxed Z (fromList [33])
〉 sumAllS example
55
```

# Larger example: Matrix multiplication

# Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-example` package which contains more than just this example.

# Start with the types

We want something like this:

$$
\begin{aligned}
\mathrm{mmultP} :: {} & \mathrm{Monad}\ m \Rightarrow \\
& \mathrm{Array\ U\ DIM2\ Double} \rightarrow \mathrm{Array\ U\ DIM2\ Double} \rightarrow \\
& m\ (\mathrm{Array\ U\ DIM2\ Double})
\end{aligned}
$$

- We inherit the `Monad` constraint from the use of a parallel compute function.
- We work with two-dimensional arrays, it's an additional prerequisite that the dimensions match.

# Strategy

We get two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`:

- we expect `w1` and `h2` to be equal,
- the resulting matrix will have shape `Z :. h1 :. w2`,
- we have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays,
- for each of these pairs, we have to take the sum of the products,
- and these results determine the values of the result matrix.

# Strategy

We get two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`:

- we expect `w1` and `h2` to be equal,
- the resulting matrix will have shape `Z :. h1 :. w2`,
- we have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays,
- for each of these pairs, we have to take the sum of the products,
- and these results determine the values of the result matrix.

Some observations:

- the result is given by a **function**,
- we need a way to **slice** rows or columns out of a matrix,

# Starting top-down

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction   (Z :. h1 :. w2)
                             (λ(Z :. r   :. c   ) → ...)
```

# Slicing

A quite useful function offered by Repa is `backpermute`:

$$
\begin{aligned}
\mathsf{backpermute} :: {} & (\mathsf{Shape}\ sh1, \mathsf{Shape}\ sh2, \mathsf{Repr}\ r\ e) \Rightarrow \\
& sh2 \rightarrow \qquad\qquad\qquad \text{-- new shape} \\
& (sh2 \rightarrow sh1) \rightarrow \quad \text{-- map new index to old index} \\
& \mathsf{Array}\ r\ sh1\ e \rightarrow \mathsf{Array}\ D\ sh2\ e
\end{aligned}
$$

- We compute a delayed array simply by saying how each index can be computed in terms of an old index.
- This is trivial to implement in terms of `fromFunction`.

# Slicing – contd.

We can use `backpermute` to slice rows and columns.

```
sliceCol  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol  c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

# Slicing – contd.

We can use `backpermute` to slice rows and columns.

```
sliceCol  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol  c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a

sliceRow  :: Repr r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

```
〉computeUnboxedS (sliceCol 3 example)
AUnboxed (Z :. 2) (fromList [4, 9])
```

Note that `sliceCol` and `sliceRow` do not actually create a new array unless we force it!

# Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

$$
\begin{aligned}
\mathsf{slice} ::\ & (\mathsf{Slice\ sl}, \mathsf{Shape\ (SliceShape\ sl)}, \mathsf{Shape\ (FullShape\ sl)}, \\
& \ \mathsf{Repr\ r\ e}) \Rightarrow \\
& \mathsf{Array\ r\ (FullShape\ sl)\ e} \rightarrow \mathsf{sl} \rightarrow \mathsf{Array\ D\ (SliceShape\ sl)\ e}
\end{aligned}
$$

A member of class `Slice`:

- looks similar to a member of class `Shape`,
- but describes **two** shapes at once, the orginal and the sliced.

# Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

```
slice :: (Slice sl, Shape (SliceShape sl), Shape (FullShape sl),
          Repr r e) ⇒
         Array r (FullShape sl) e → sl → Array D (SliceShape sl) e
```

A member of class `Slice`:

- looks similar to a member of class `Shape`,
- but describes **two** shapes at once, the orginal and the sliced.

```
sliceCol, sliceRow :: Repr r e ⇒
                      Int → Array r DIM2 e → Array D DIM1 e
sliceCol c  a = slice a (Z :. All :. c  )
sliceRow r a = slice a (Z :. r    :. All)
```

# Putting everything together

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction (Z :. h1 :. w2)
                (λ(Z :. r :. c) →
                   sumAllS (sliceRow r m1 *^ sliceCol c m2)
                ))
```

That's all. Note that we compute no intermediate arrays.

# Testing it

(Demo.)

# Summary

- The true magic of Repa is in the `computeP`-like functions, where parallelism is automatically handled.
- Haskell's type system is used in various ways:
  - Adapt the representation of unboxed arrays to element types.
  - Keep track of the shape of an array, to make fusion explicit.
  - Keep track of the state of an array.
- We have seen yet another embedded domain-specific language:
  - for efficient array computations,
  - allowing high-level deterministic parallelism,
  - where the types direct us towards correct use.
- A large part of Repa's implementation is actually quite understandable.