Adding Support for Multi-Dimensional Arrays to Data Parallel Haskell

Gabriele Keller with
M. Chakravarty, S. Peyton Jones, R. Leshchinskiy
Programming Languages & Systems
School of Computer Sciences & Engineering
University of New South Wales
Sydney

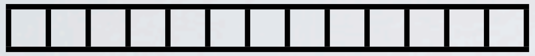
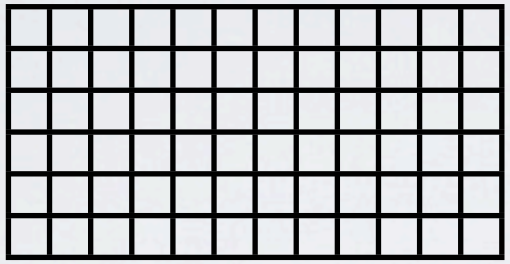
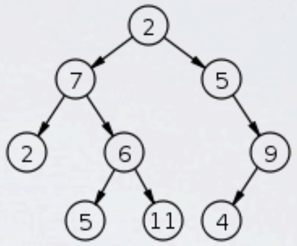
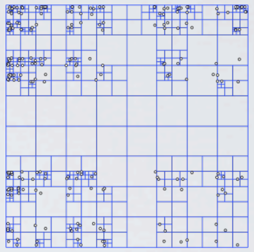
UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

PLS

DATA PARALLEL HASKELL

- Data Parallel Haskell (DPH) was designed with irregular parallel applications in mind:
  - structure of parallel computations/data structures impossible to predict statically
- Nested arrays as parallel data structure, elements and shape information distributed over processors
- Interface similar to list operations:
  - collective operations like map, fold, filter, array comprehension executed in parallel

Two forms of data parallelism

flat, regular	nested, irregular
 	 
limited expressiveness	covers sparse structures and even divide&conquer
close to the hardware model	needs to be turned into flat parallelism for execution
well understood compilation techniques	highly experimental program transformations

Example: Sparse matrix vector multiplication

- matrix represented in compressed row format
- every non-zero element represented as pair of column index and value
- every row as array of elements, matrix as array of rows

```
smvm' :: [:[: (Int, Double) :]:] -> [:Double:] -> [:Double:]
smvm' m v =
   [: sumP [: x * (v !: i) | (i,x) <- row :] | row <- m :]
```

Can we express regular computations in DPH?

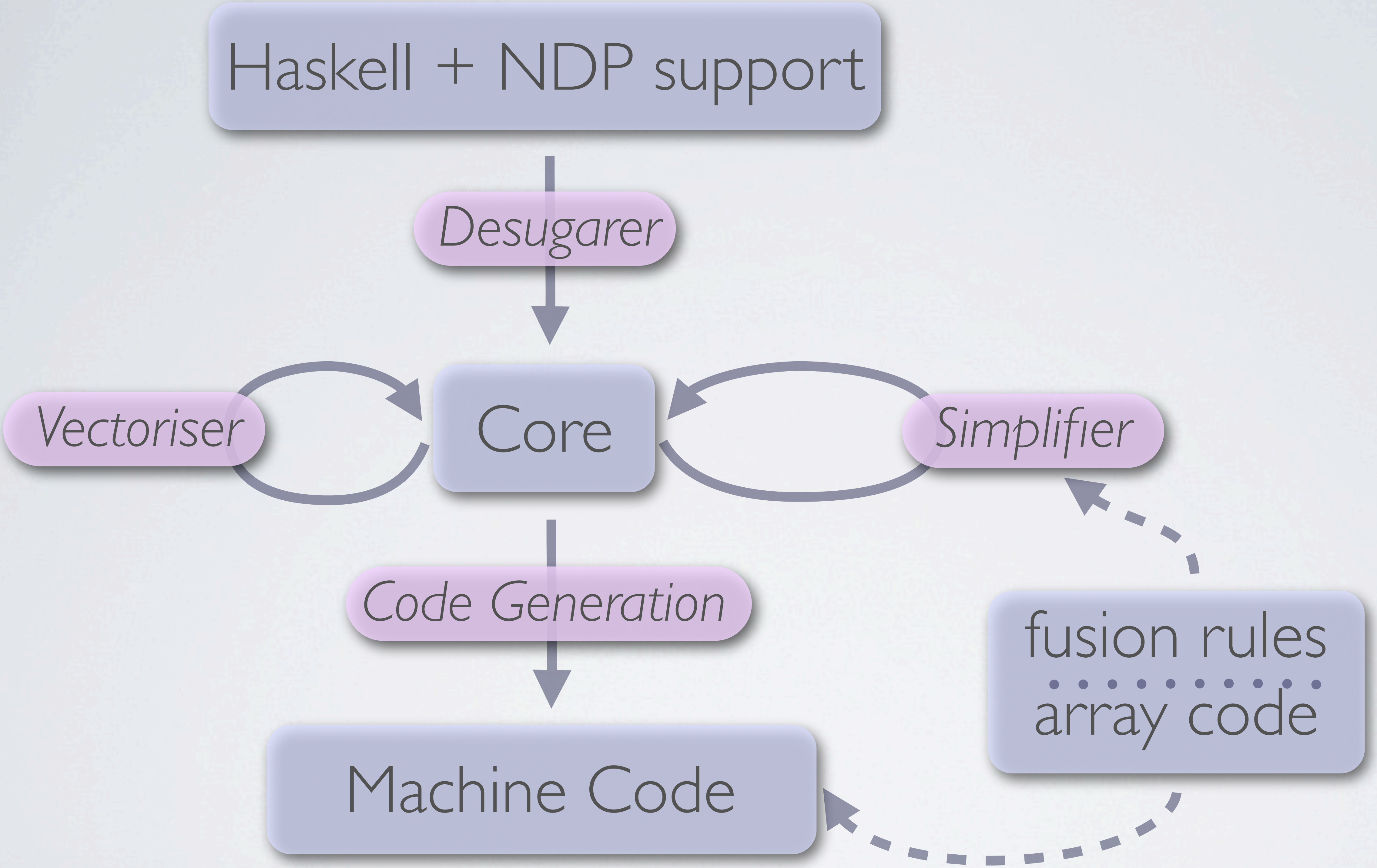
- nested arrays could be interpreted as n-dim arrays:

```
transpose:: [:[:a:]:] -> [:[:a:]:]
transpose m =
    [:[: v :! i | v <- m  :] | i <-[:0..(length m) -1:]
```

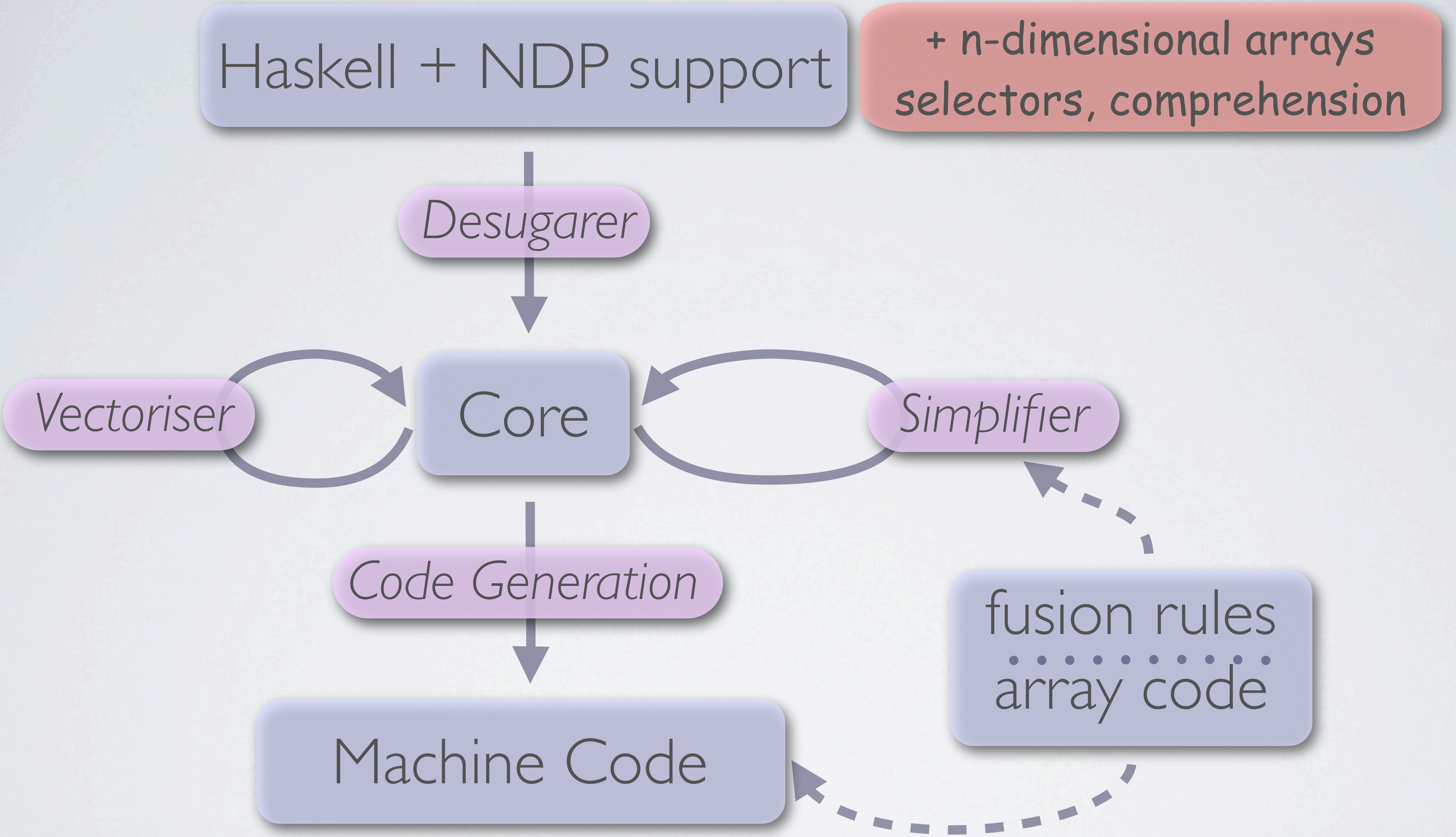
- awkward for more complicated operations (e.g., relaxation)

- wasteful, error prone, inefficient

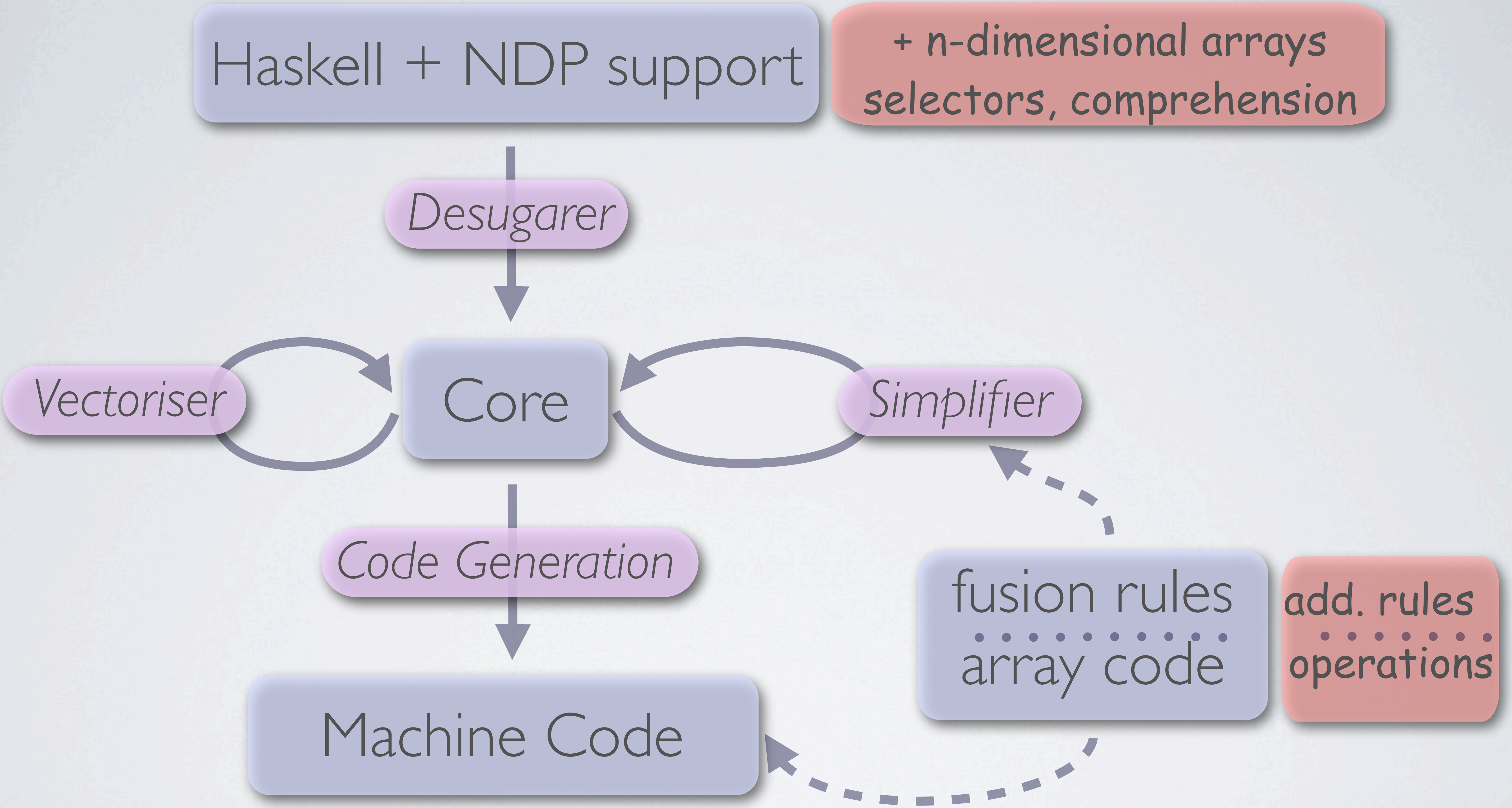
DPH Compilation



DPH Compilation



DPH Compilation



Haskell + NDP support

+ n-dimensional arrays
selectors, comprehension

Desugarer

Vectoriser

Core

Simplifier

Code Generation

Machine Code

fusion rules
array code

add. rules
operations

Design Questions

- How much syntactic support?
  - selection/indexing of subarrays
  - array comprehension
- How much static checking of shape information?
  - shape checking
  - shape polymorphic operations
- Which basic operations do we need?
- Interaction between regular and irregular computations

Tracking and checking of Shape Information

- Shape information:
  - dimensionality and length of each dimension
- Statically checked:
  - dimensionality
- Dynamically checked:
  - size of each dimension

N-Dim Arrays

- Arrays parametrised with shape descriptor type and element type:

```
Array dim e
```

- dimensionality on type level, size on value level

- element type restricted to basic types and pairs thereof

DIMENSIONALITY

- element-wise mapping works on arrays of any dim, leaves it unchanged:

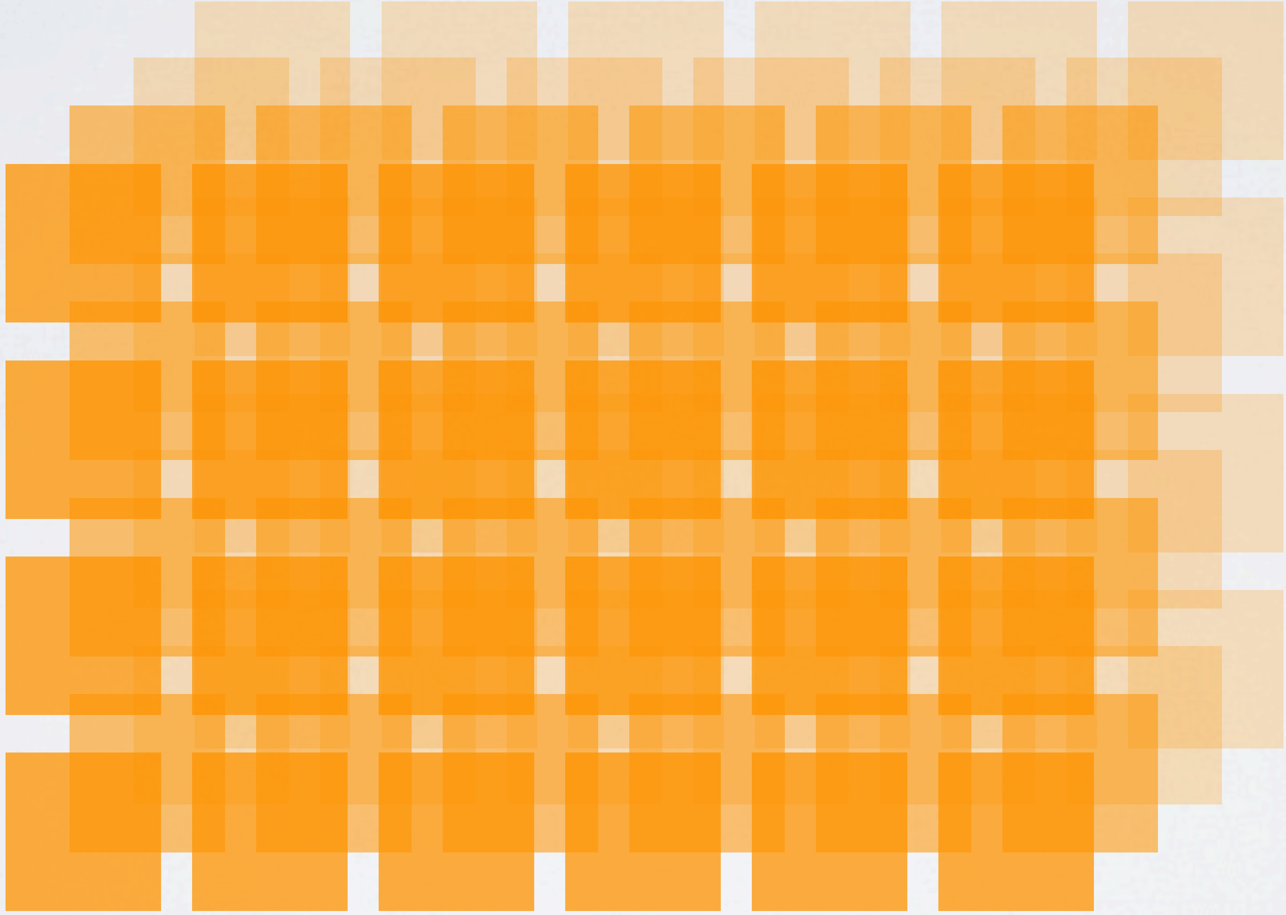
```
map:: (a -> b) -> Array dim a -> Array dim b
```

- some operations require the array to be of a specific dimensionality:

```
inverse:: Array DIM2 Double -> Array DIM2 Double
```

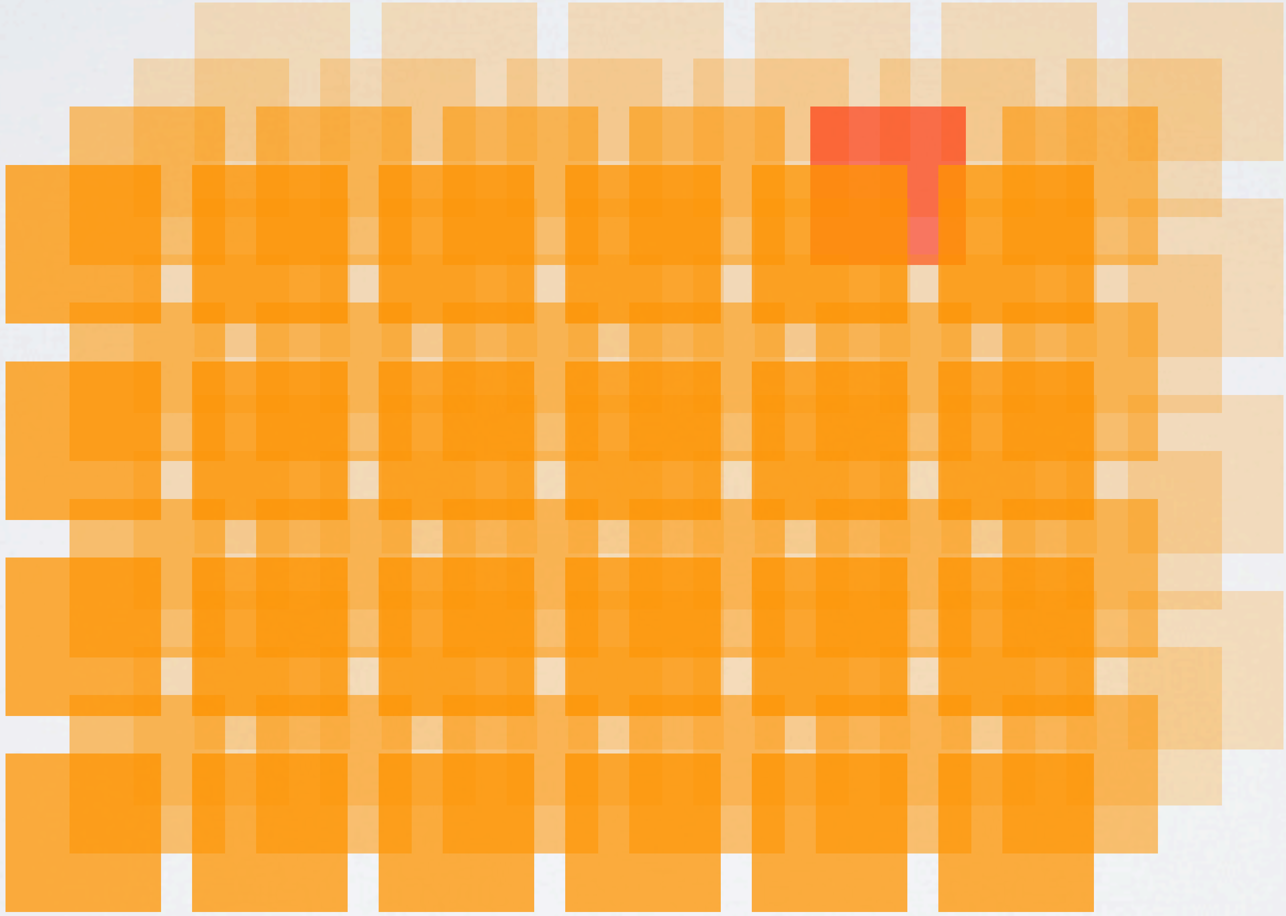
- for some operations, we want to express a more complex relationship between argument and result dimension

```
(!:):: Array dim a -> selector -> Array (depends on selector) a
```



- for some operations, we want to express a more complex relationship between argument and result dimension

```
(!:):: Array dim a -> selector -> Array (depends on selector) a
```

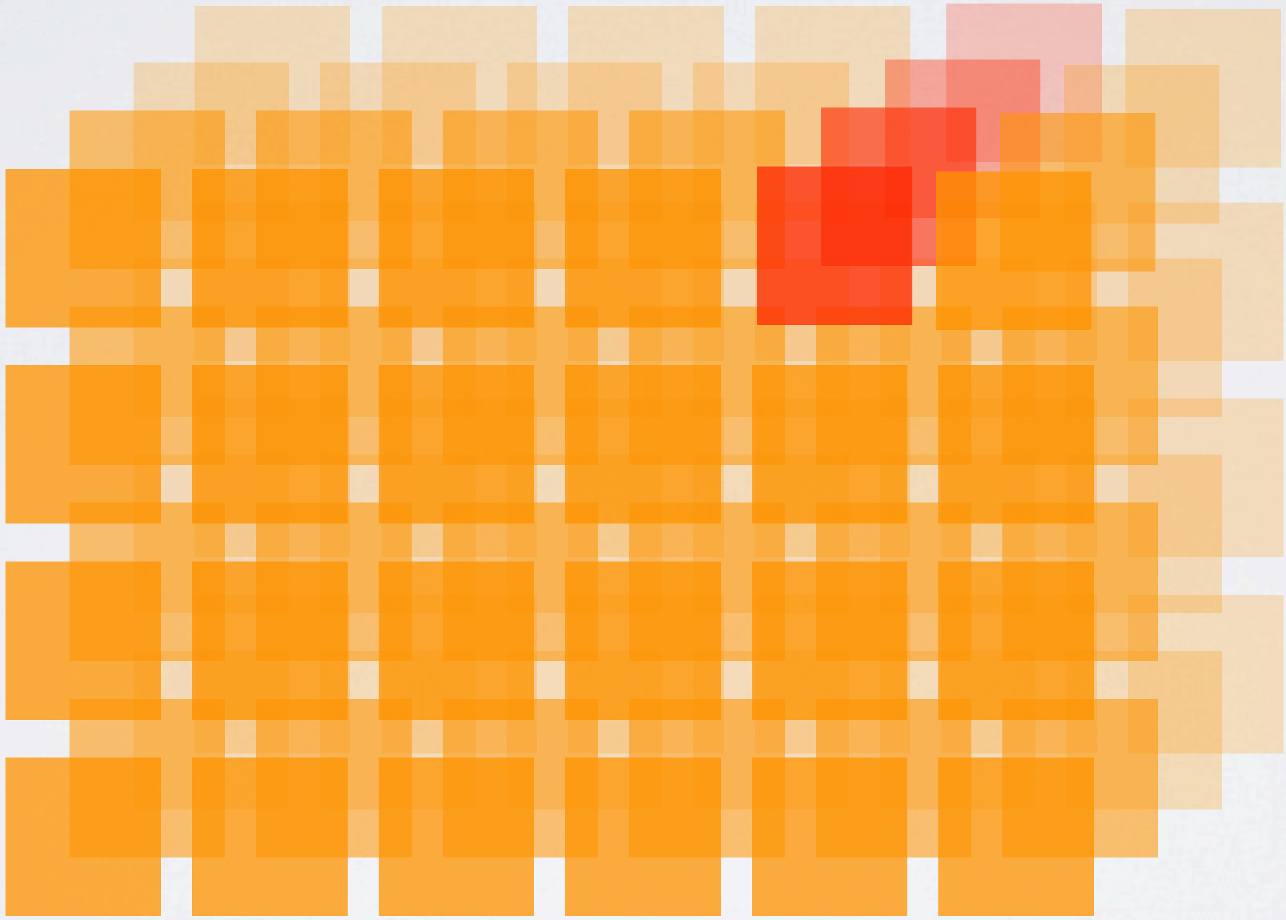


(4,0,1)

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

- for some operations, we want to express a more complex relationship between argument and result dimension

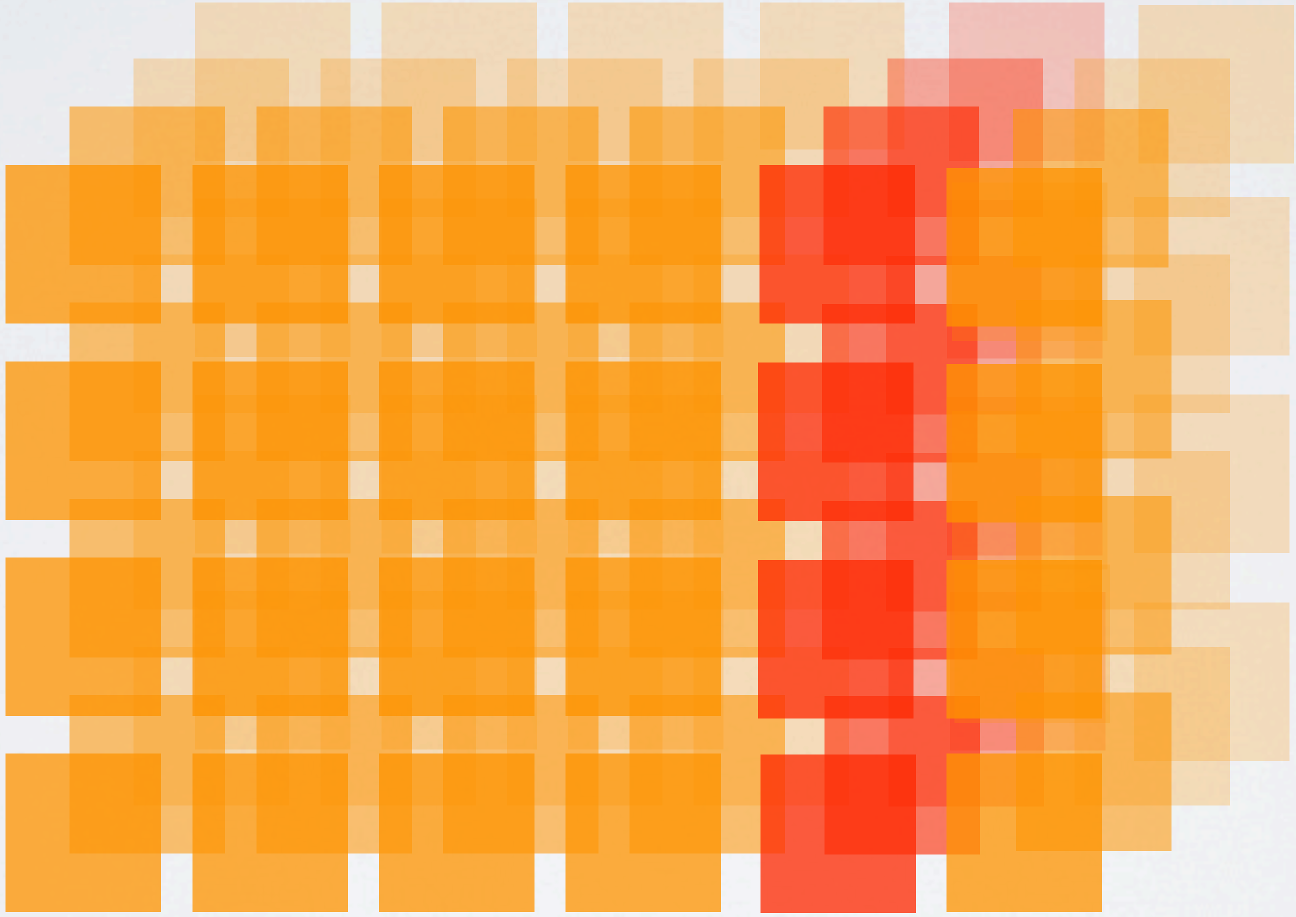
```
(!:):: Array dim a -> selector -> Array (depends on selector) a
```



(4,0,.)

- for some operations, we want to express a more complex relationship between argument and result dimension

```
(!:):: Array dim a -> selector -> Array (depends on selector) a
```



(4,.,.)

Representing the shape of an array:

- to do type level calculations on the dimensionality, we use internally an inductive definition

```
type DIM0 = ()
type DIM1 = (DIM0, Int)
type DIM2 = (DIM1, Int)
.....
```

- this is only used as internal representation type, the user should see them as n-tuples:

```
()
Int
(Int, Int)
.....
```

The Index type

- the generalised selection notation expresses an relationship between initial and projected dimension:

```
(4, 0, 3)
(4,  . , 3)
```

- The index type reflects this relationship on the type level:

```
data Index initialDim projectedDim where
  IndexNil   :: Index () ()
  IndexAll   :: Index init proj -> Index (init, Int) (proj, Int)
  IndexFixed :: Int -> Index init proj -> Index (init, Int)  proj
```

- terms of index typed only used internally

The Index type

- Some examples

```
IndexFixed 4 (IndexAll (IndexFixed 3 ())):: Index DIM3 DIM1
                      (4,  . , 3)

IndexFixed 4 (IndexAll (IndexAll ())):: Index DIM3 DIM2
                      (4, ., .,)
```

- With this definition, we can express the type of select as:

```
(!:):: Array dim e -> Index dim dim’ -> Array dim’
```

- for example

```
arr:: Array DIM3 Double
arr !: (IndexFixed 4 (IndexFixed 0 (IndexFixed 1 IndexNil)))
```

- similarly, we can use the index type to express the type of a generalized replicate:

```
replicate:: Array dim e -> Index dim‘ dim -> Array dim‘ e
```

- examples:

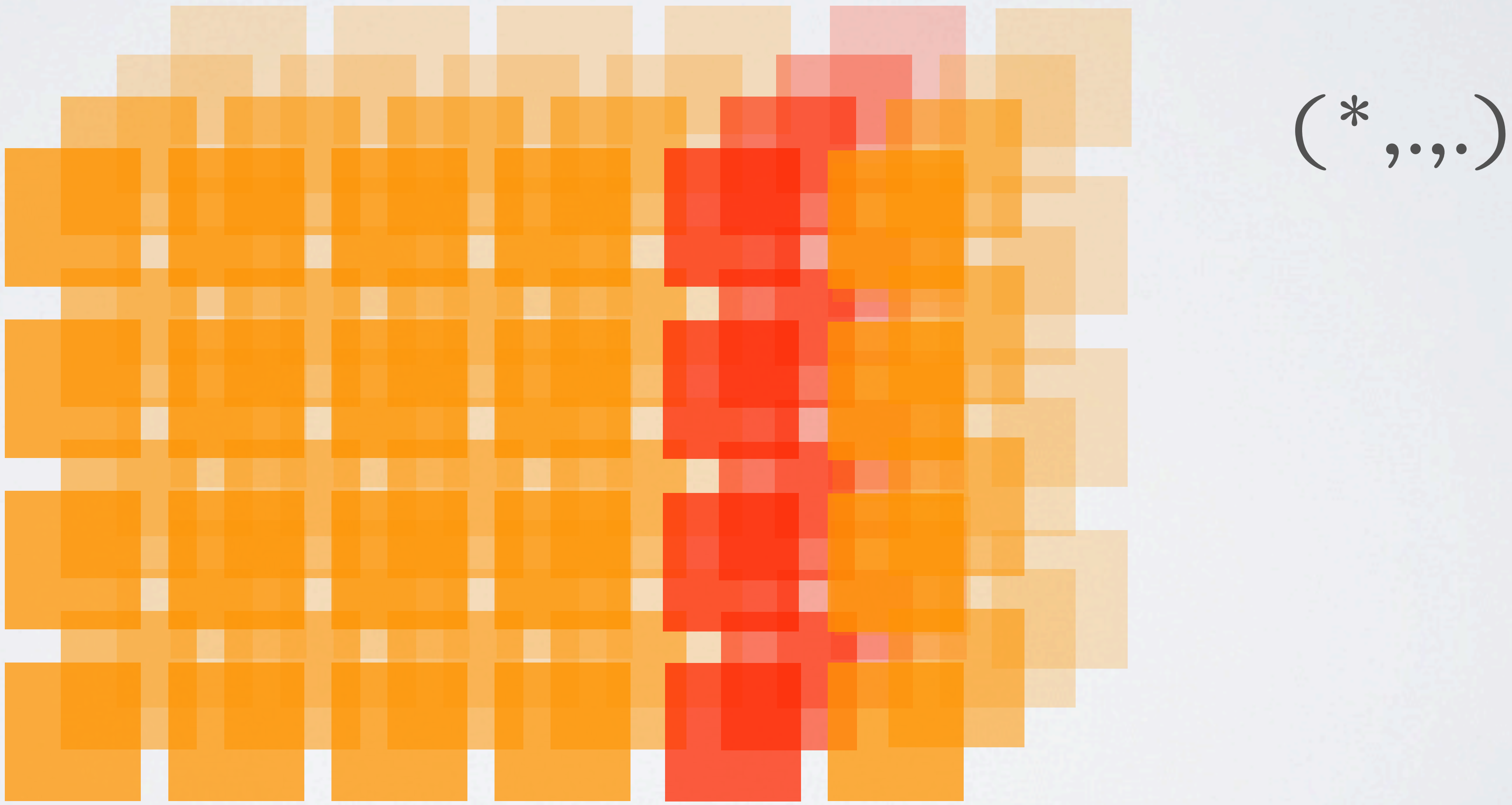
```
s:: Array DIM0 Int
replicate s (IndexFixed 5 ())
replicate s (IndexFixed 5 (IndexFixed 3 ())

v:: Array DIM1 Int
replicate v (IndexAll (IndexFixed 5 ())):: Array DIM2 Int
replicate v (IndexFixed 5 (IndexAll ())):: Array DIM2 Int
```

Mapping a reduction operation

- Collapsing all the elements along one or multiple dimensions into a scalar value

```
mapFold:: Array dim a -> Index dim dim’  -> (Array dim’ a -> b) ->  ?
```



The index type revisited

- we add an additional parameter to the index type

```
data Index a initialDim projectedDim where
  IndexNil   :: Index a () ()
  IndexAll   :: Index a init proj -> Index a (init, Int) (proj, Int)
  IndexFixed :: a -> Index a init proj -> Index a (init, Int)  proj
```

- and the type of indexing changes accordingly

```
(!:):: Array dim e -> Index Int dim dim’ -> Array dim’
```

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

PLS

- but still, what is the result type?

```
mapFold:: (Array dim a) ->
  Index () dim dim’ -> (Array dim’ a -> b)-> Array (dim - dim’) b
```

- to perform subtraction on the type level, we define the type family

```
type family (:-:) init proj
type instance (:-:) init () = init
type instance (:-:) (init,Int) (proj, Int) = (:-:) init proj
```

- but still, what is the result type?

```
mapFold:: (Array dim a) ->
  Index () dim dim’ -> (Array dim’ a -> b)-> Array (dim :-: dim’) b
```

- to perform subtraction on the type level, we define the type family

```
type family (:-:) init proj
type instance (:-:) init () = init
type instance (:-:) (init,Int) (proj, Int) = (:-:) init proj
```

Basic Operations

- Separating reordering/extraction of array elements and computations on elements

- Extraction/reordering:

```
bpermute::
  Array dim a -> (dim’ -> dim) -> Array dim’ a

defaultBpermute::
  Array dim a -> b -> (dim’ -> Maybe dim) -> Array dim’ a
```

UNSW

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

PLS

OPERATIONS

- Transposing, tiling, rotation, shifts can be easily expressed in terms of backpermute and default backpermute
  - relaxation in terms of shifts or backpermute straight forward
- No overhead if such a newly created array is immediately used as an argument to another function (stream fusion)
- element-wise map, scan, fold, zipWith to perform computations

Combining Regular & Irregular Computations

- Regular arrays as elements of irregular structures are useful to control the granularity of parallel computations
- Irregular structures insides regular arrays not allowed at the moment - should they be?

Status

- Implementation of library in progress
- Currently implementing examples to figure out if operations etc appropriate
- User level syntax not fixed yet