Expressing array programs

Robert Clifton-Everest
robertce@cse.unsw.edu.au
robeverest

Purely functional array languages/libraries

- Combinator based (map, fold, scan, filter, etc..)
- High-level
- Declarative
- Data-parallel
- Accelerate is a good example

Accelerate

An Accelerate computation

```
type Vector e = Array (Z:.Int) e
```

```
type Scalar e = Array Z e
```

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Array languages

	Multidimensional	**Nested**
Regular	Accelerate	
Irregular		

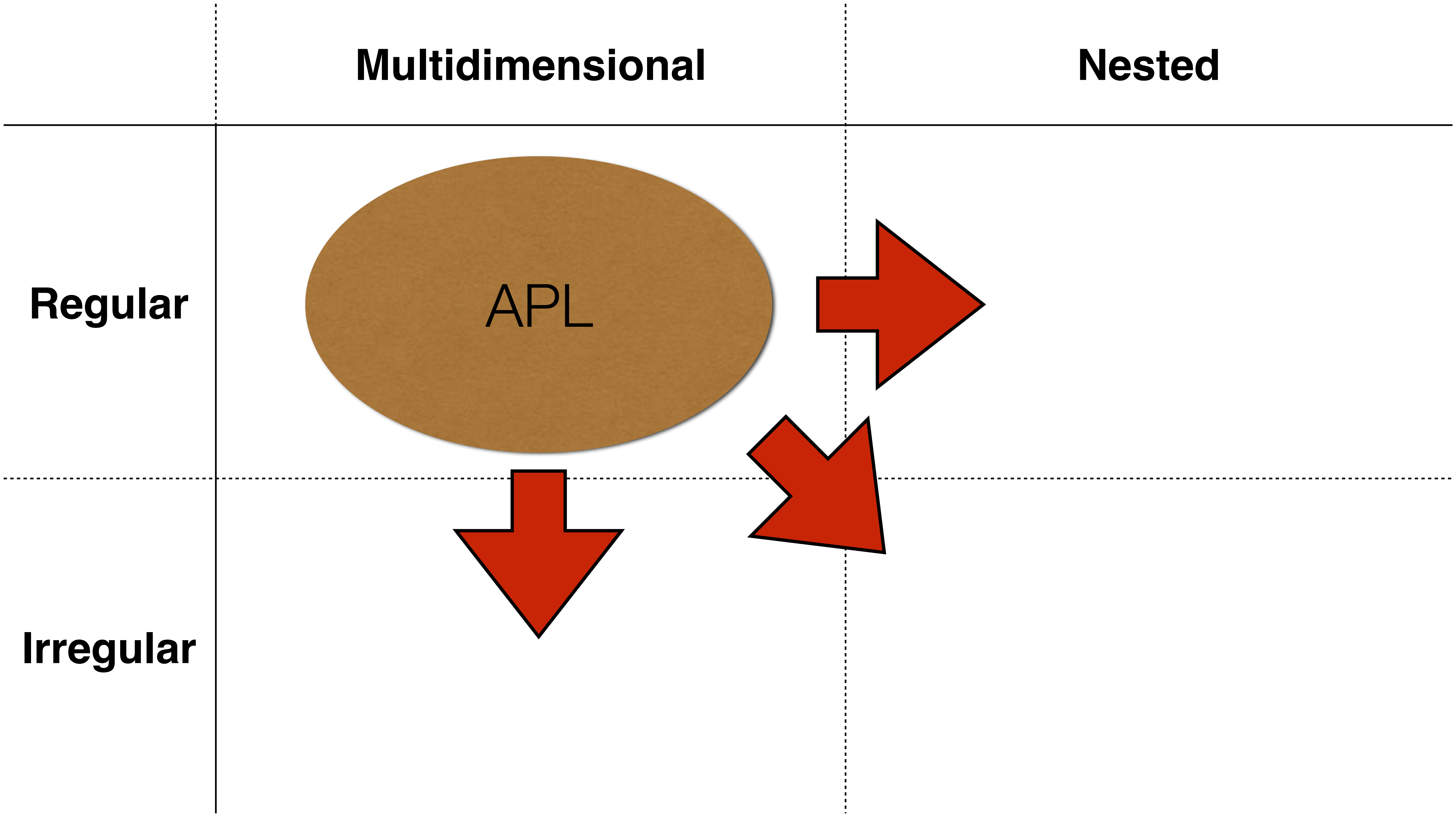
Modularity

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)

type Matrix e = Array (Z:.Int:.Int) e

mvm :: Acc (Matrix Float) -> Acc (Vector Float) -> Acc (Vector Float)
mvm m v = fold (+) 0 (zipWith (*) m (replicate (Z:. height m :. All) v)
```

mvm_{apl} `m v = dotp m v`*

* Not actual APL

Arrays



Modularity

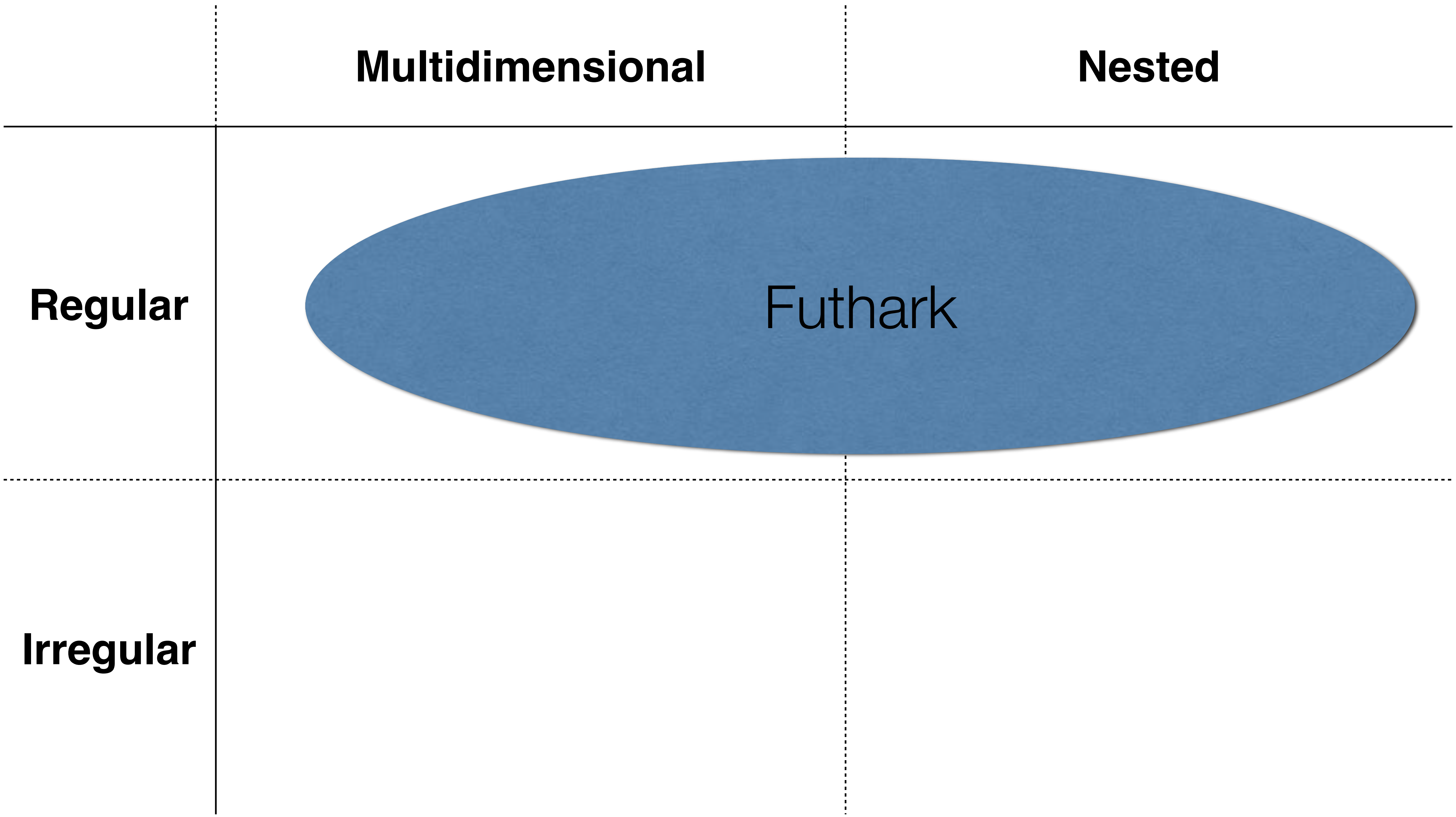
```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)

type Matrix e = Array DIM2 e

mvm :: Acc (Matrix Float) -> Acc (Vector Float) -> Acc (Vector Float)
mvm m v = fold (+) 0 (zipWith (*) m (replicate (Z:. height m :. All) v)
```

~~mvm_apl m v = dotp m v*~~

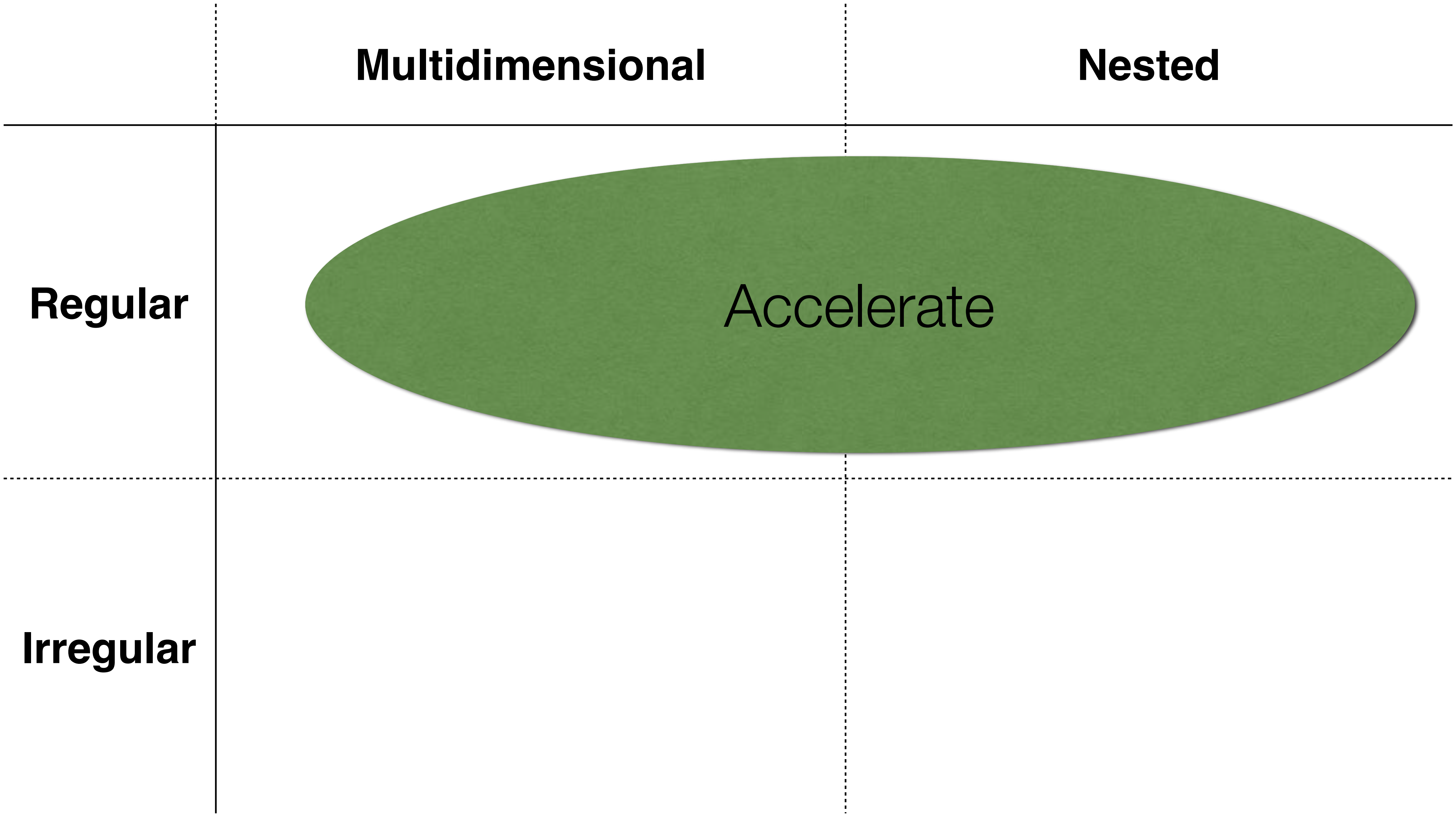
* Not actual APL

```
mvm_n :: Acc (Matrix Float) -> Acc (Vector Float) -> Acc (Vector Float)
mvm_n m v = concat_Z (map (dotp v) (rows m))

rows :: Acc (Array (sh:.Int) e) -> Acc (Array sh (Vector Int))

concat_Z :: Acc (Array sh (Scalar e)) -> Acc (Array sh e)
```

Modularity



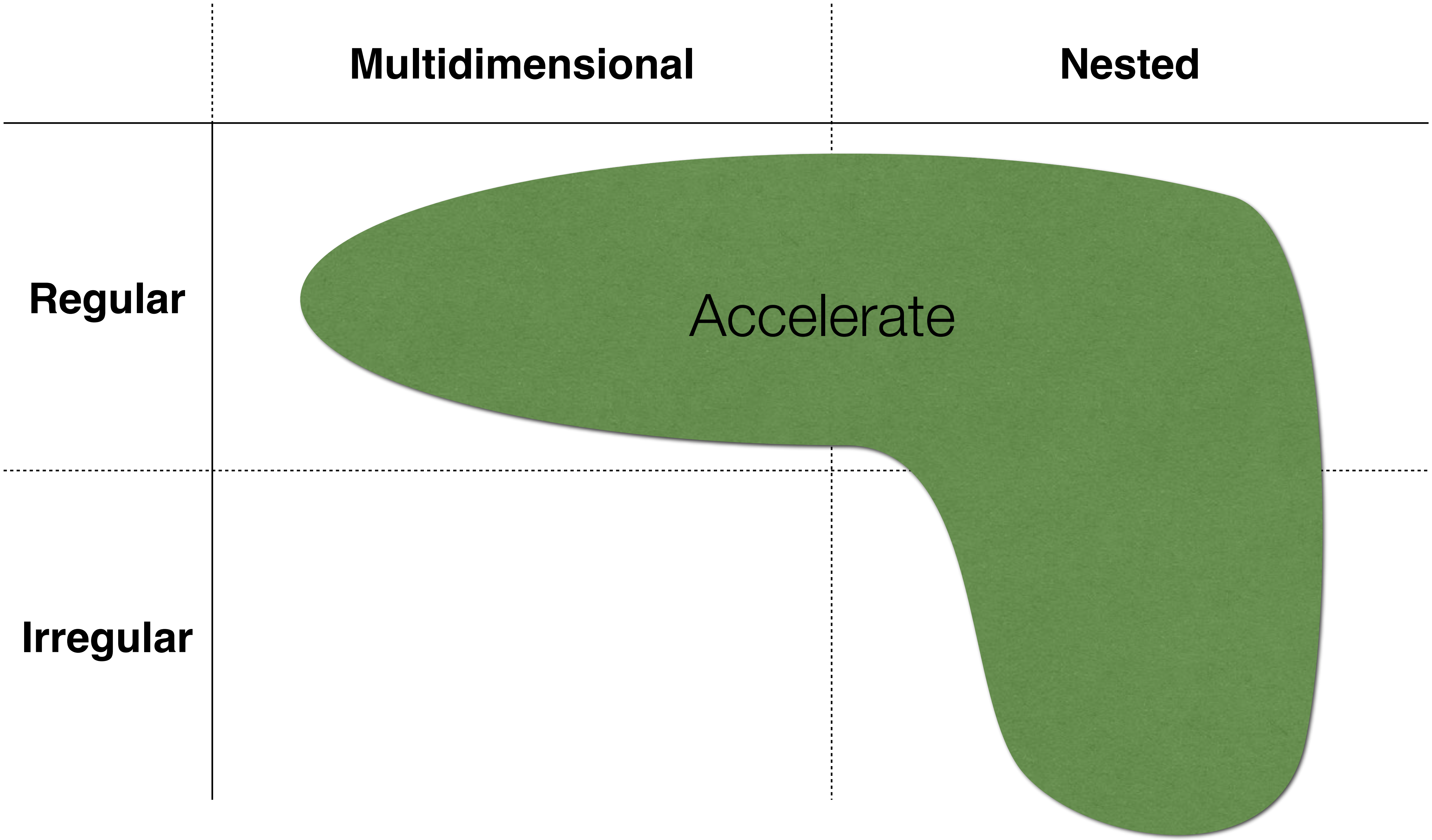
Modularity



Irregularity

```
type SparseVector e = Vector (Int, Vector e)
type SparseMatrix e = Vector (SparseVector e)

smvm :: SparseMatrix e -> Vector e -> Vector e
smvm m v = map (\sv -> let (ixs, vals) = unzip sv
                       in dotp vals (gather ixs v))
```

Two camps



Array representation

- Big topic

- The basics

$$\texttt{Vector (Vector e)} \Longrightarrow \texttt{(Vector Int, Vector e)}$$

Two camps

	Multidimensional	Nested
Regular		
Irregular		DPH NESL

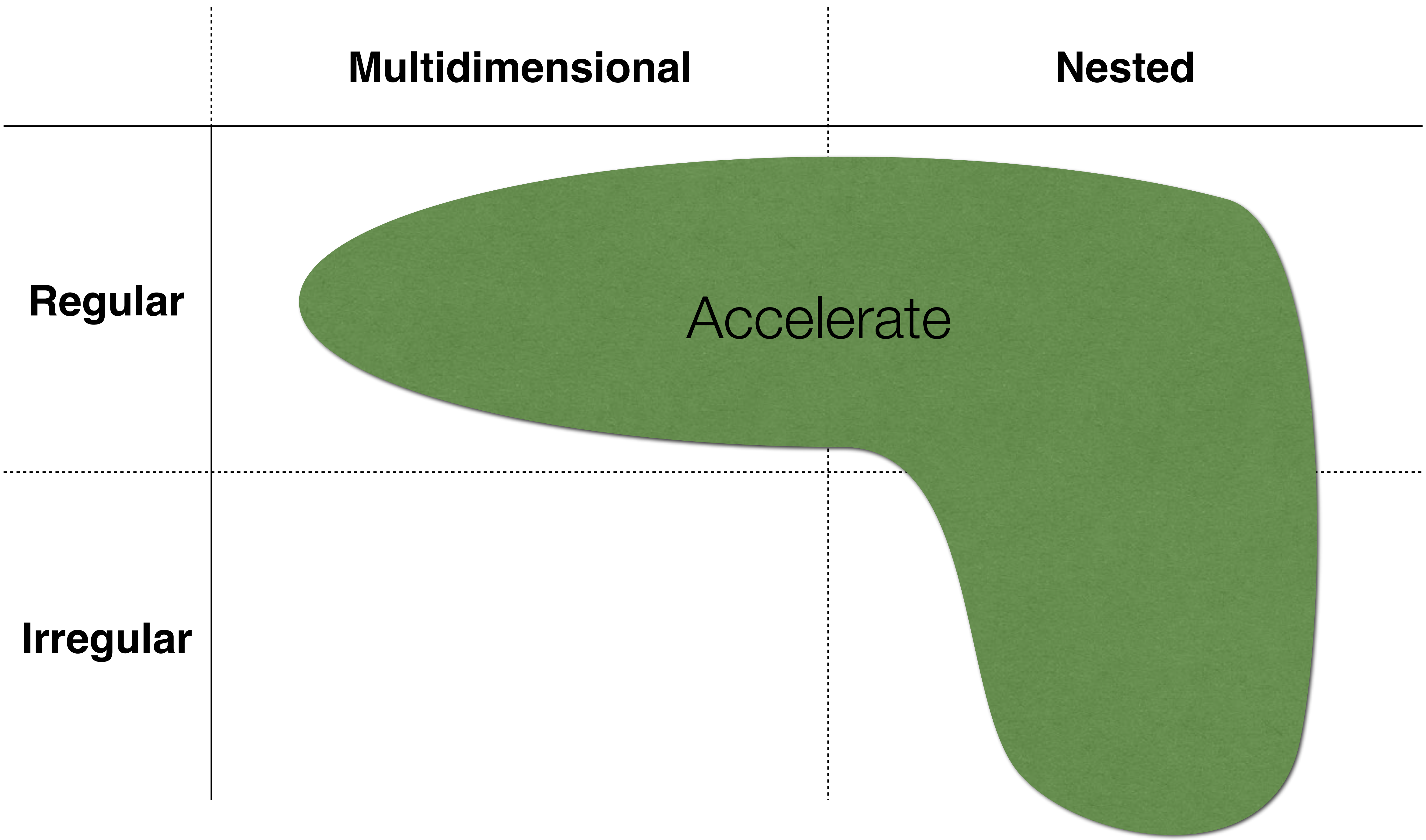
Array representation

- Big topic
- The basics

```
Vector (Vector e) ⟹ (Vector Int, Vector e)
```

- Not so good for regular
- Two representations?
- What about one for both?

Array representation



Irregular multidimensional arrays

```
type Ints ≈ Vector Int

Array (Z:.Int:.Ints) e

Array (Z:.Int:.Ints:.Int) e
```

Regularity aware flattening

```
rows m :: Vector (Vector Float)
```

↓

```
Array (Z:.Int:.Int) Float
```

```
map (filter (>0)) rows m
    :: Vector (Vector Float)
```

↓

```
Array (Z:.Int:.Ints) Float
```

MVM again

```
mvmn :: Acc (Matrix Float) -> Acc (Vector Float) -> Acc (Vector Float)
mvmn m v = concatZ (map (dotp v) (rows m))


mvmn m v = (dotp v)↑ m

dotp v :: Acc (Vector Float) -> Acc (Scalar Float)

(dotp v)↑ :: Acc (Array (sh:.i) Int) -> Acc (Array sh Int)
(dotp v) ys = let sh :. _ = shape ys
              in fold (+) 0 (zipWith (replicate (sh:.All) v) ys)
```

This approach

- Advantages
  - Only pay for irregularity when it's really needed
  - Defers scheduling decisions

```
Array (Z:.Int:.Ints:.Int)
```

- Disadvantages
  - Requires a richer implementation
  - The result type of a transformed function is unknown till after transformation