Streaming and Nested Parallelism in Accelerate

Robert Clifton-Everest
University of New South Wales

Jointly with
Frederik M. Madsen
Trevor L. McDonell
Manuel M. T. Chakravarty
Gabriele Keller

GPUs

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz

- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz

- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data
- How can we take advantage of this power?

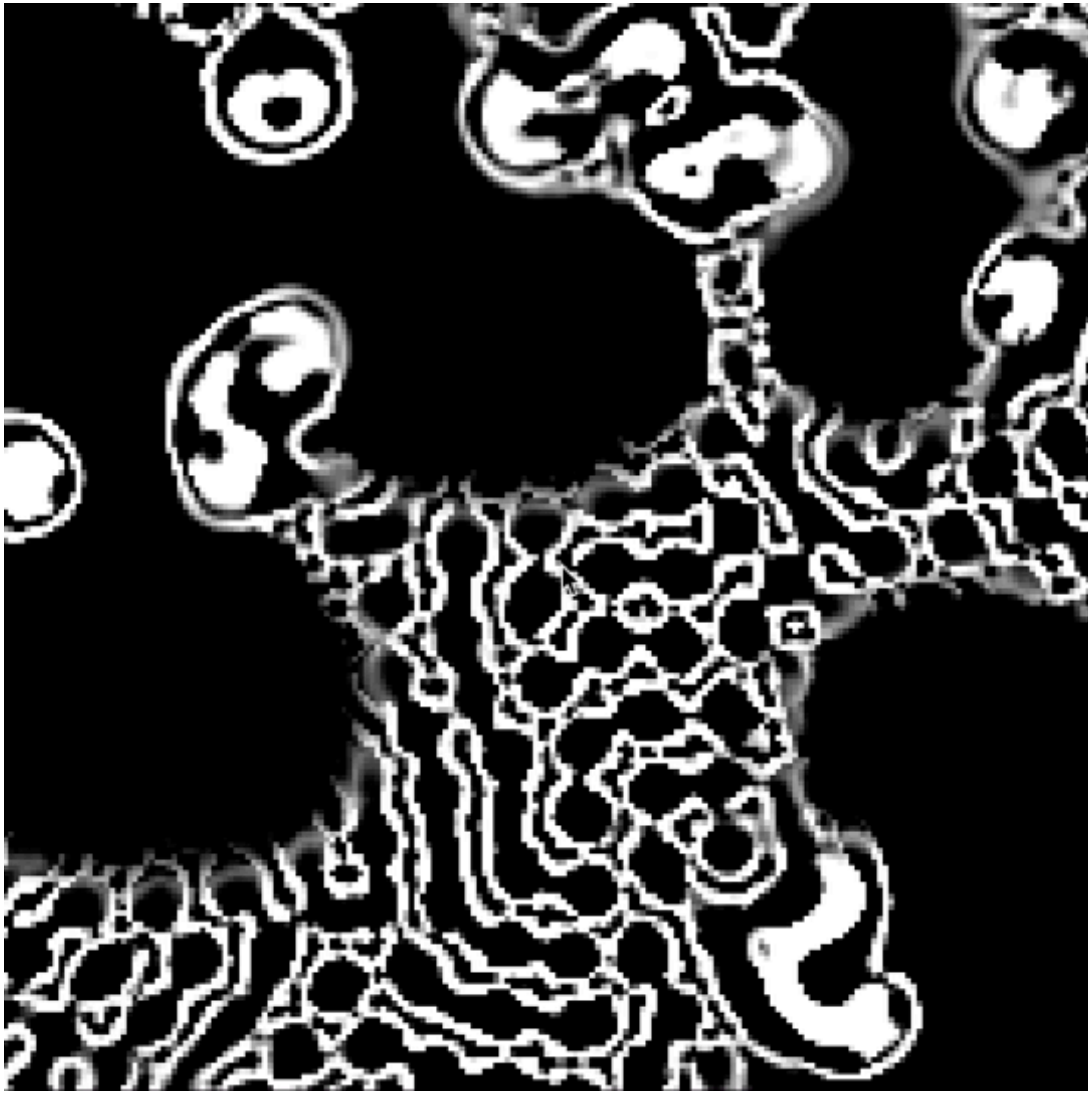
GPUs

- Lots of raw computing power
    - This one: 2688 cores @ 867 MHz
- Different hardware design
    - Limited instruction set
    - SIMD: Cores run the same program, but on different data
- How can we take advantage of this power?

With a high-level embedded language of course!

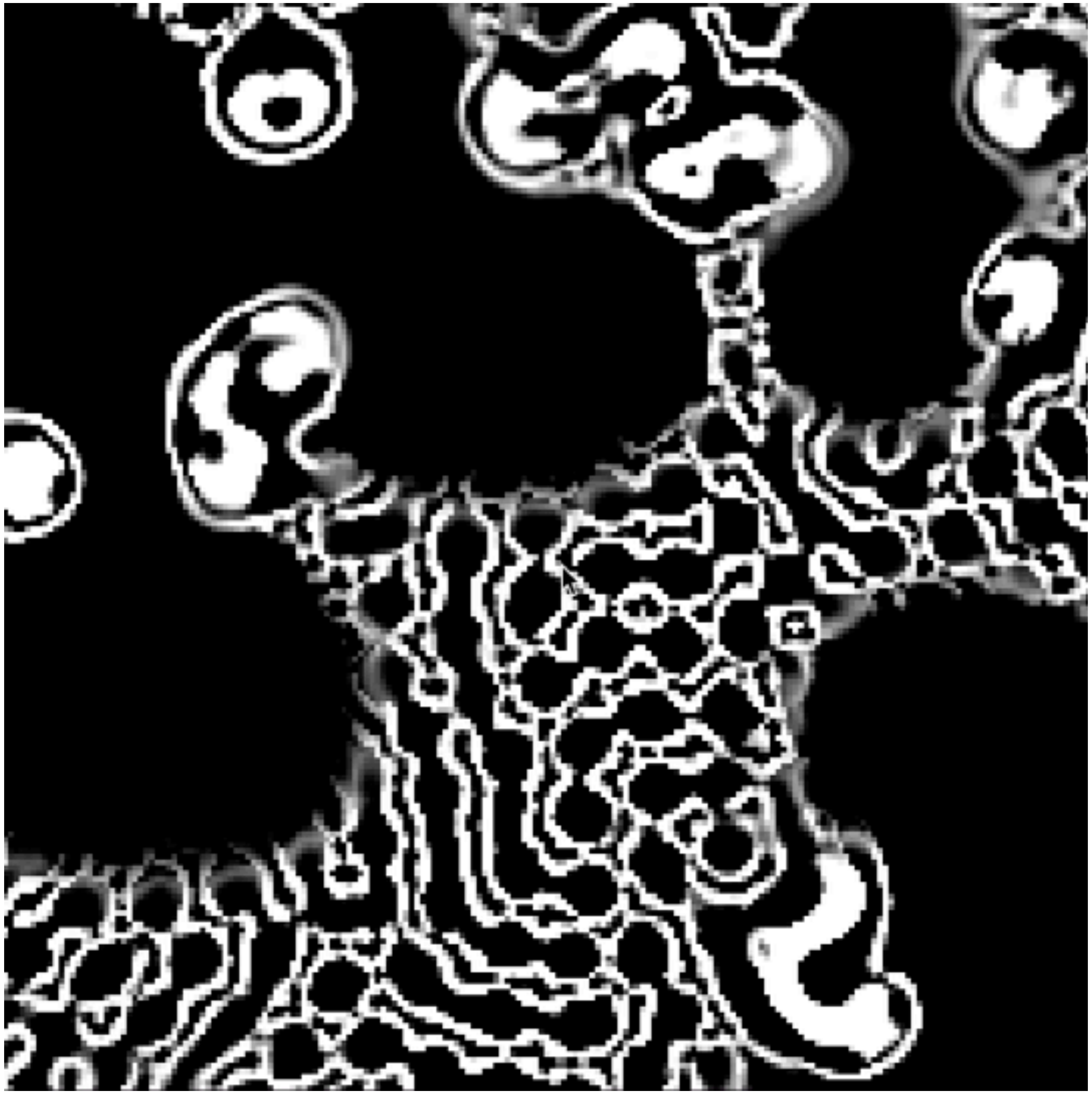
Accelerate

An embedded language for GPU programming



Accelerate

An embedded language for GPU programming



```
dotp xs ys =
```

Embedded
language arrays

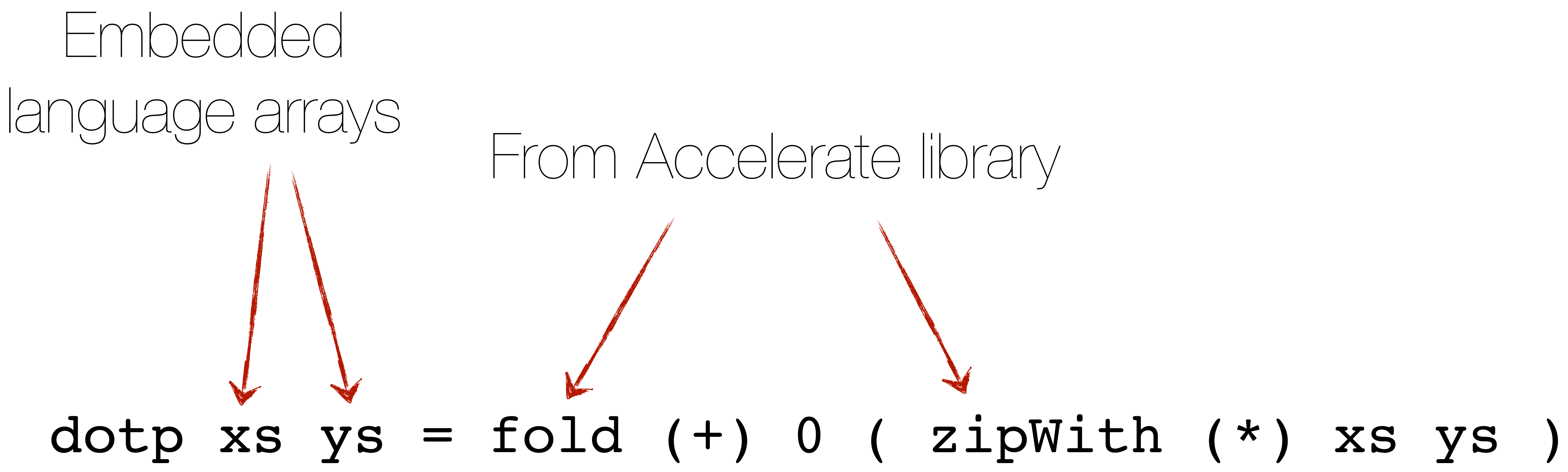
```
dotp xs ys =
```

Embedded
language arrays

```
dotp xs ys =                 zipWith (*) xs ys
```

Embedded
language arrays

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```



Embedded
language arrays

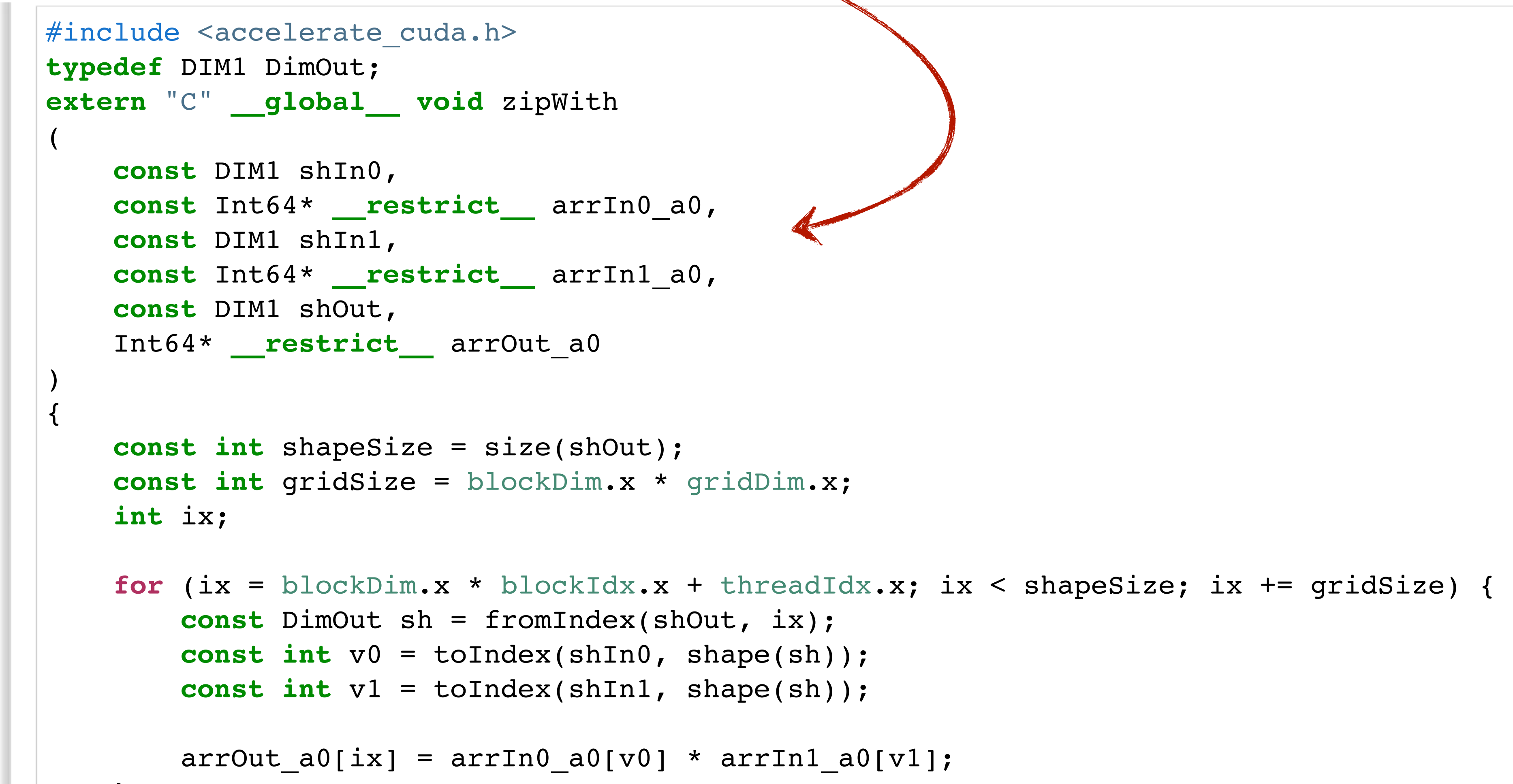
From Accelerate library

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

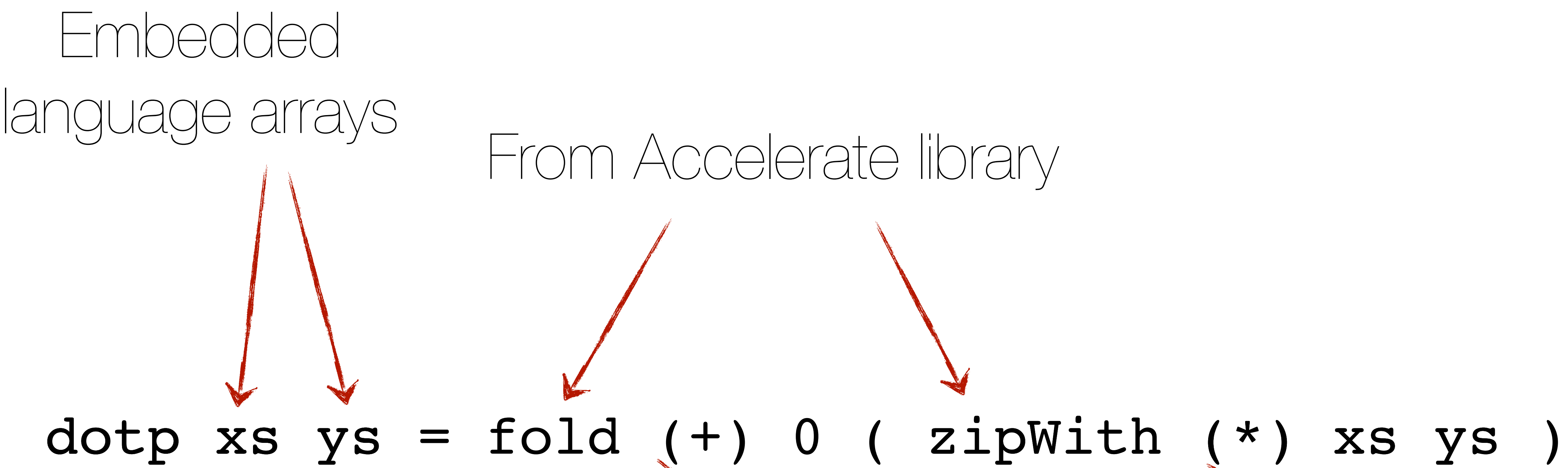
Embedded
language arrays

From Accelerate library

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```



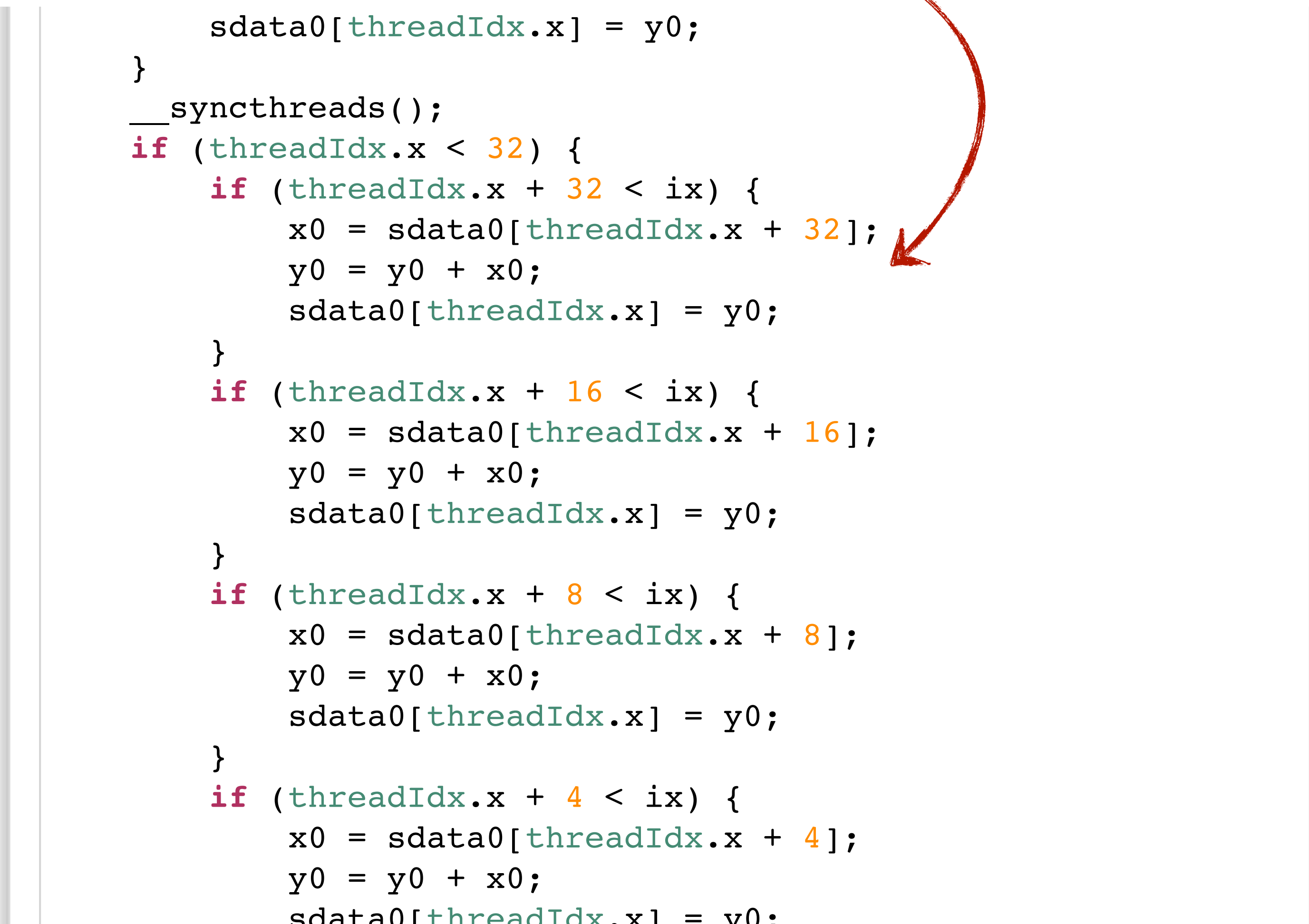
```
#include <accelerate_cuda.h>
typedef DIM1 DimOut;
extern "C" __global__ void zipWith
(
    const DIM1 shIn0,
    const Int64* __restrict__ arrIn0_a0,
    const DIM1 shIn1,
    const Int64* __restrict__ arrIn1_a0,
    const DIM1 shOut,
    Int64* __restrict__ arrOut_a0
)
{
    const int shapeSize = size(shOut);
    const int gridSize = blockDim.x * gridDim.x;
    int ix;

    for (ix = blockDim.x * blockIdx.x + threadIdx.x; ix < shapeSize; ix += gridSize) {
        const DimOut sh = fromIndex(shOut, ix);
        const int v0 = toIndex(shIn0, shape(sh));
        const int v1 = toIndex(shIn1, shape(sh));

        arrOut_a0[ix] = arrIn0_a0[v0] * arrIn1_a0[v1];
```



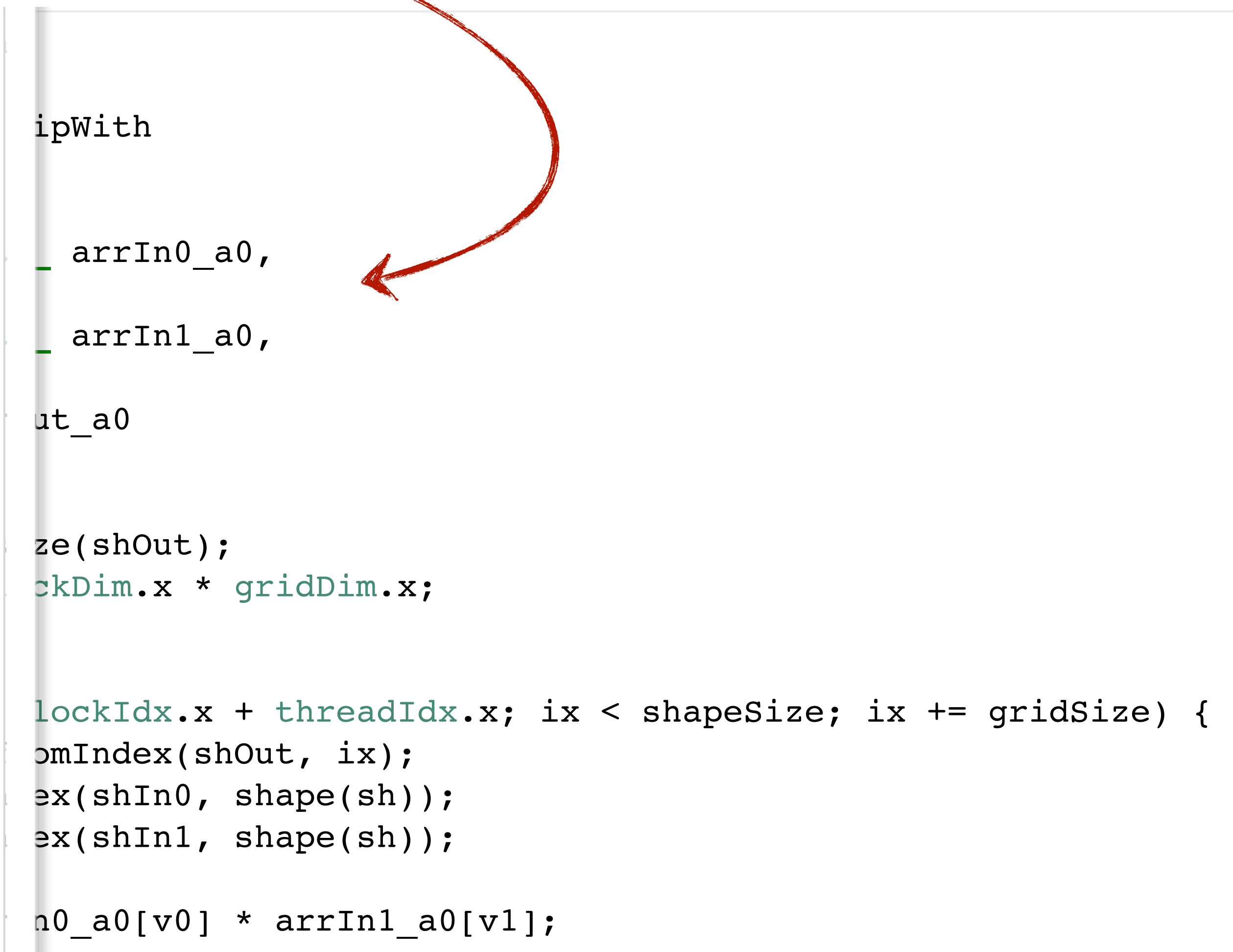
Embedded language arrays

From Accelerate library

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```



```
        sdata0[threadIdx.x] = y0;
    }
    __syncthreads();
    if (threadIdx.x < 32) {
        if (threadIdx.x + 32 < ix) {
            x0 = sdata0[threadIdx.x + 32];
            y0 = y0 + x0;
            sdata0[threadIdx.x] = y0;
        }
        if (threadIdx.x + 16 < ix) {
            x0 = sdata0[threadIdx.x + 16];
            y0 = y0 + x0;
            sdata0[threadIdx.x] = y0;
        }
        if (threadIdx.x + 8 < ix) {
            x0 = sdata0[threadIdx.x + 8];
            y0 = y0 + x0;
            sdata0[threadIdx.x] = y0;
        }
        if (threadIdx.x + 4 < ix) {
            x0 = sdata0[threadIdx.x + 4];
            y0 = y0 + x0;
            sdata0[threadIdx.x] = y0;
```



```
ipWith


_ arrIn0_a0,

_ arrIn1_a0,

ut_a0


ze(shOut);
ckDim.x * gridDim.x;


lockIdx.x + threadIdx.x; ix < shapeSize; ix += gridSize) {
omIndex(shOut, ix);
ex(shIn0, shape(sh));
ex(shIn1, shape(sh));

n0_a0[v0] * arrIn1_a0[v1];
```

Accelerate

Accelerate

- A deep embedding

Accelerate

- A deep embedding

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

Accelerate

- A deep embedding

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

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

Accelerate

- A deep embedding

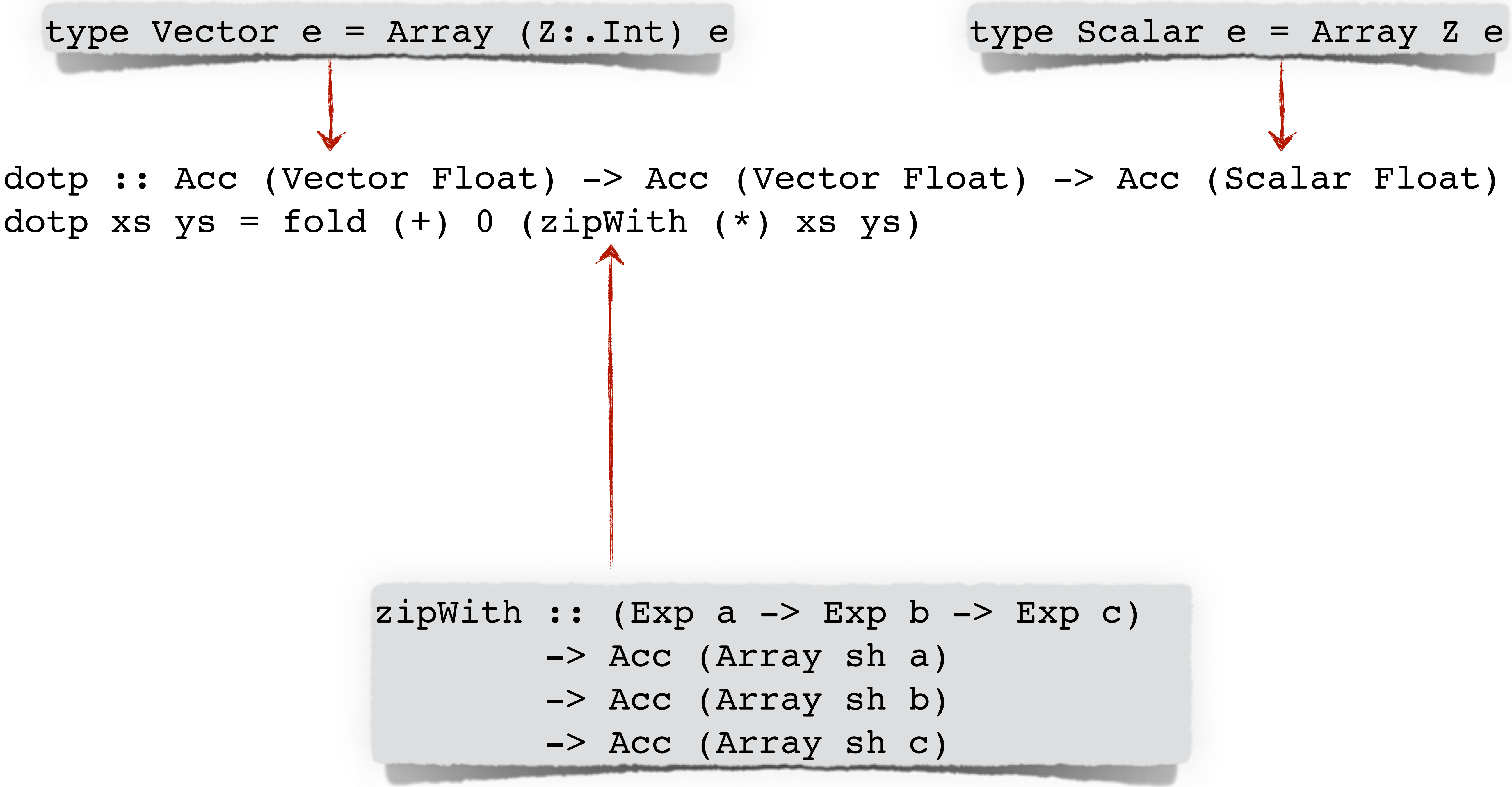
```
type Vector e = Array (Z:.Int) e
```

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

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

Accelerate

- A deep embedding



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

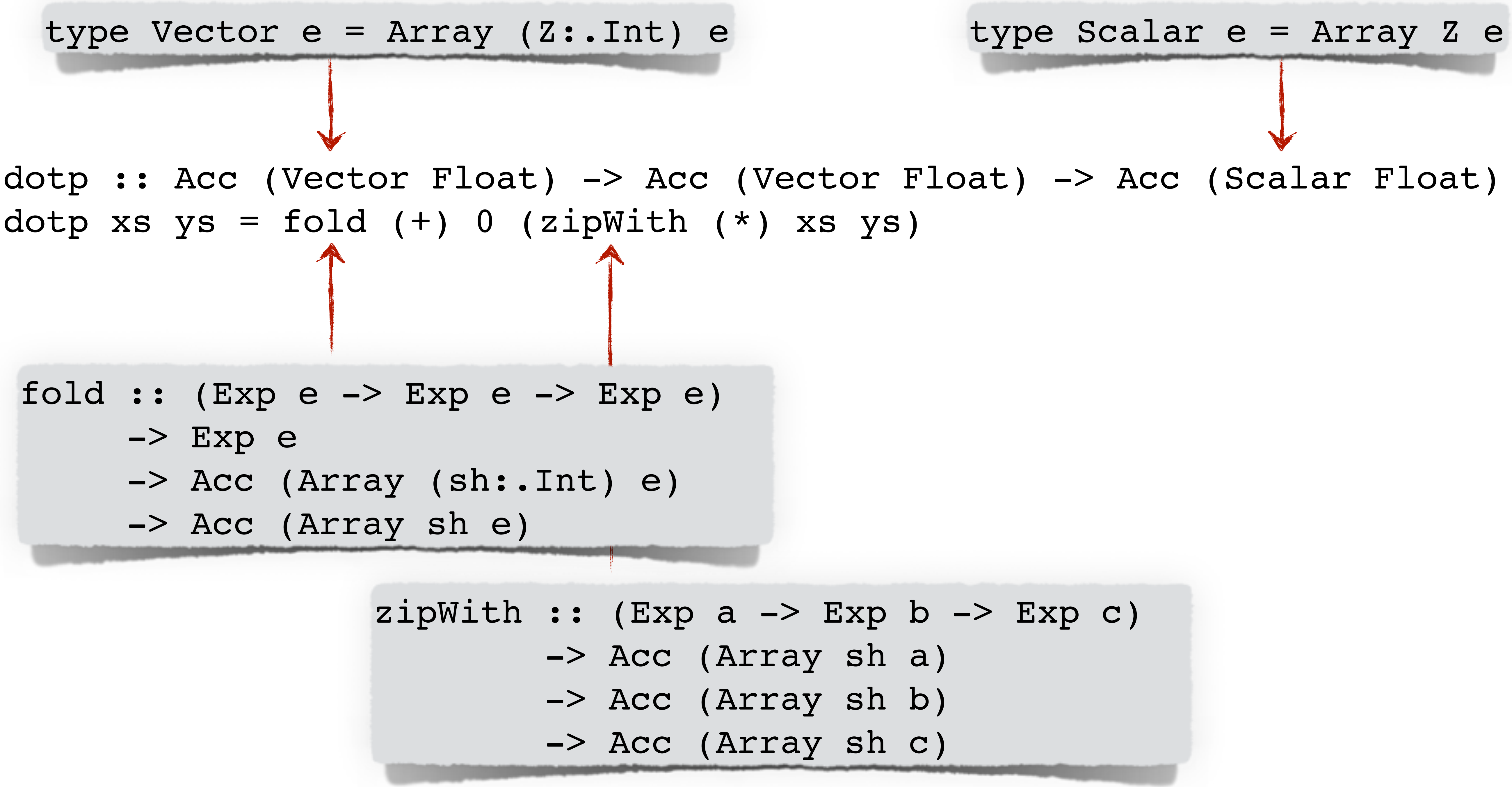
```
type Scalar e = Array Z e
```

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

```
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

Accelerate

- A deep embedding



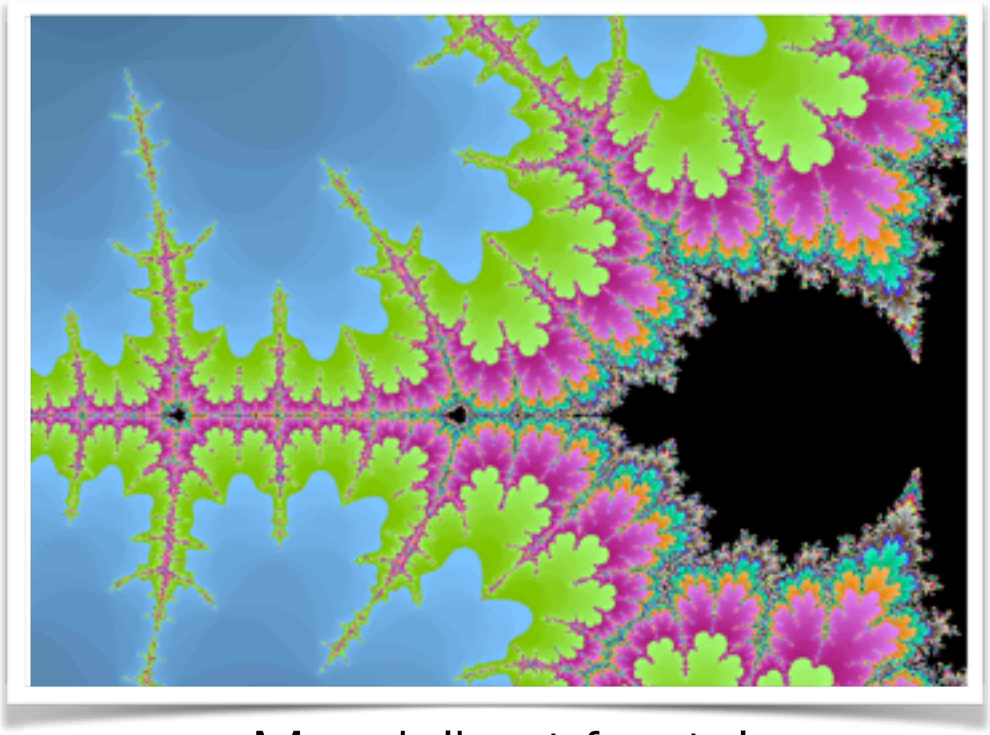
```
type Vector e = Array (Z:.Int) e
```

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

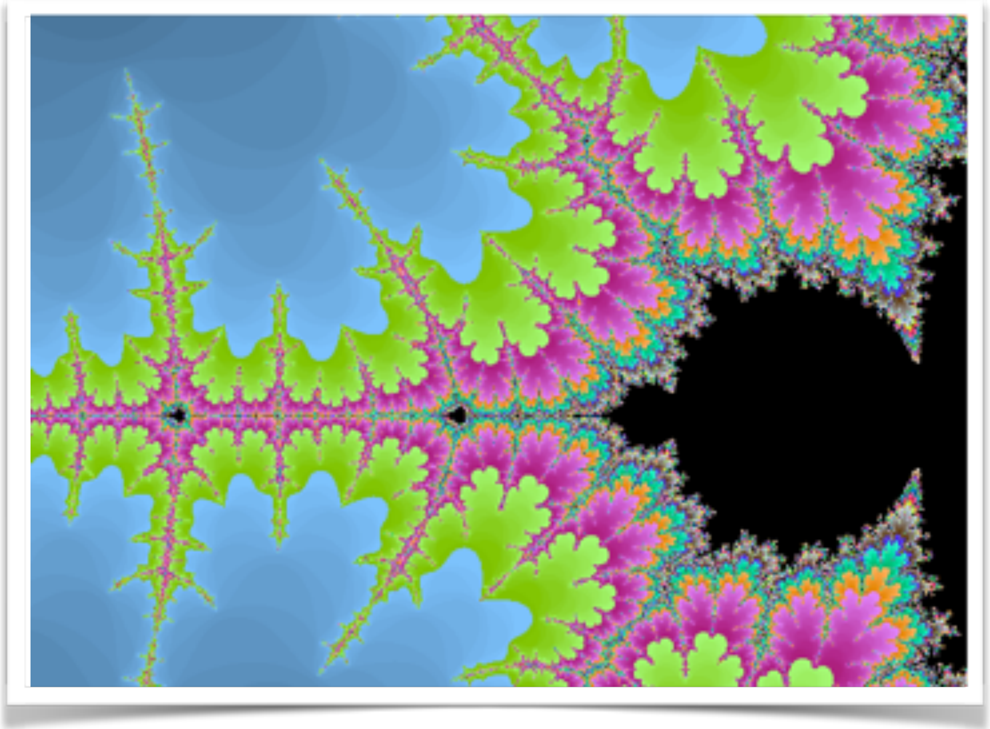
```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
fold :: (Exp e -> Exp e -> Exp e)
     -> Exp e
     -> Acc (Array (sh:.Int) e)
     -> Acc (Array sh e)
```

```
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

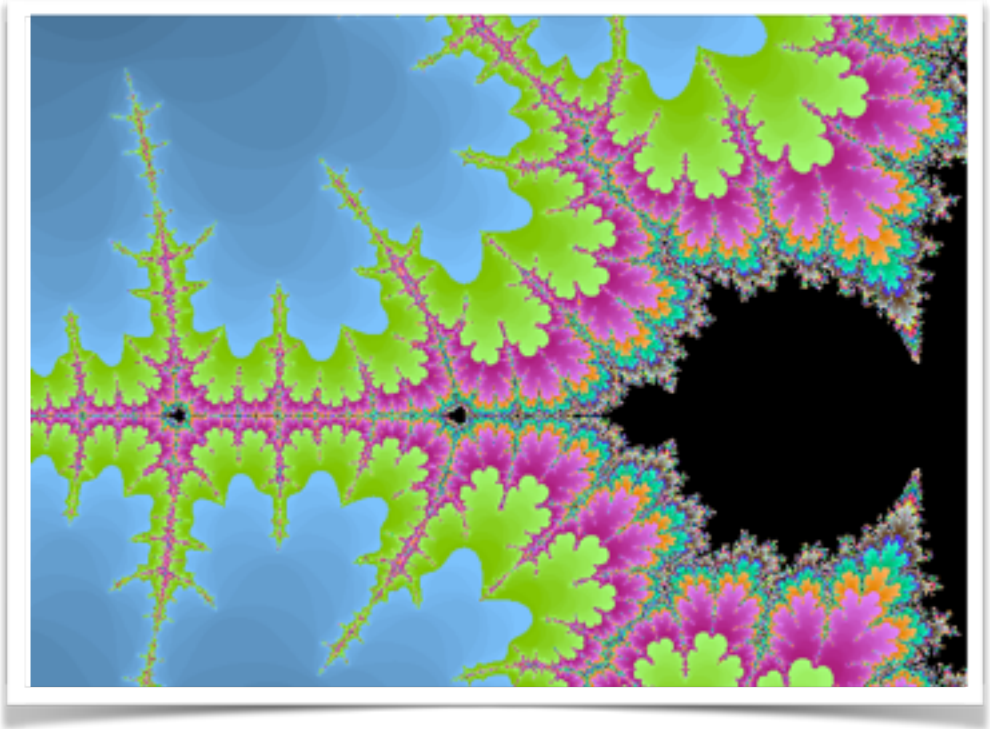


Mandelbrot fractal

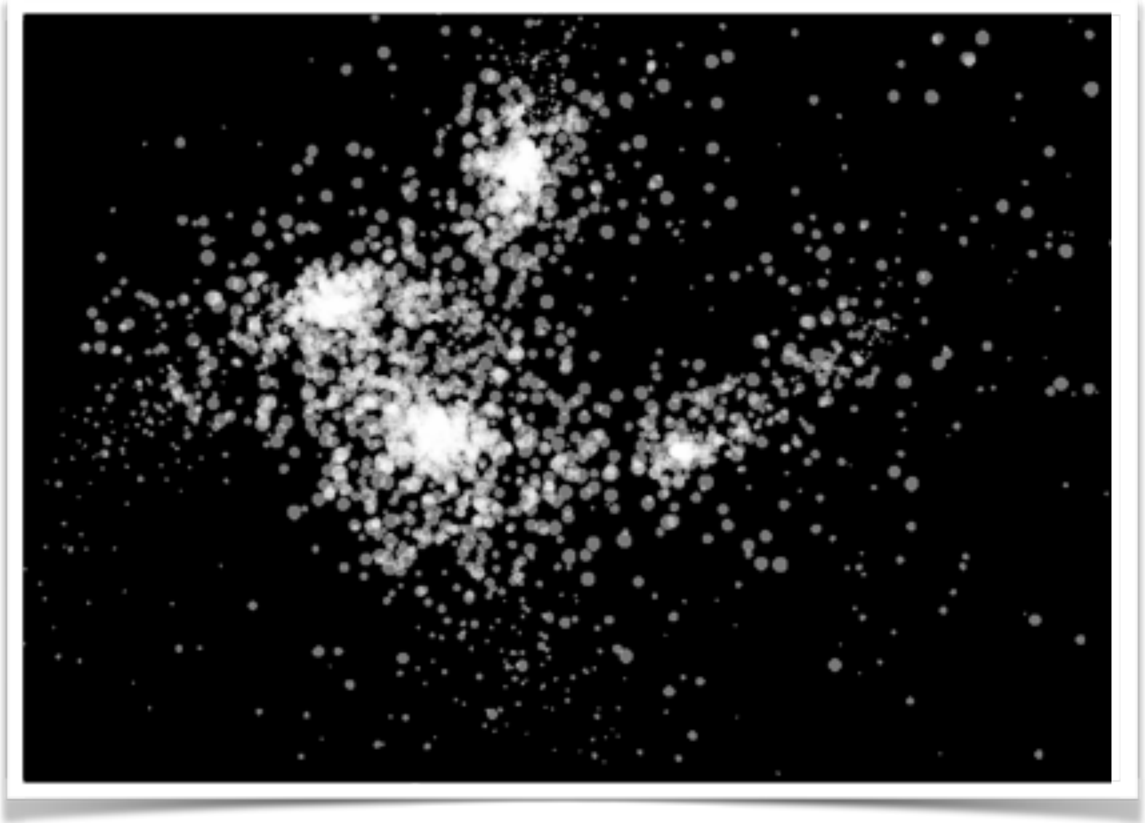


Mandelbrot fractal

n-body gravitational simulation

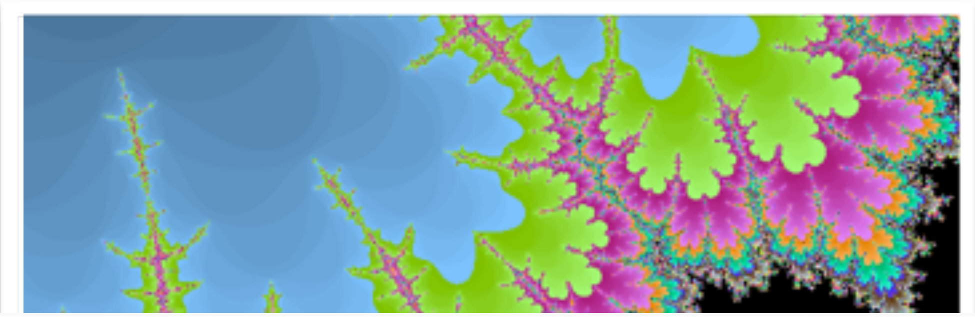


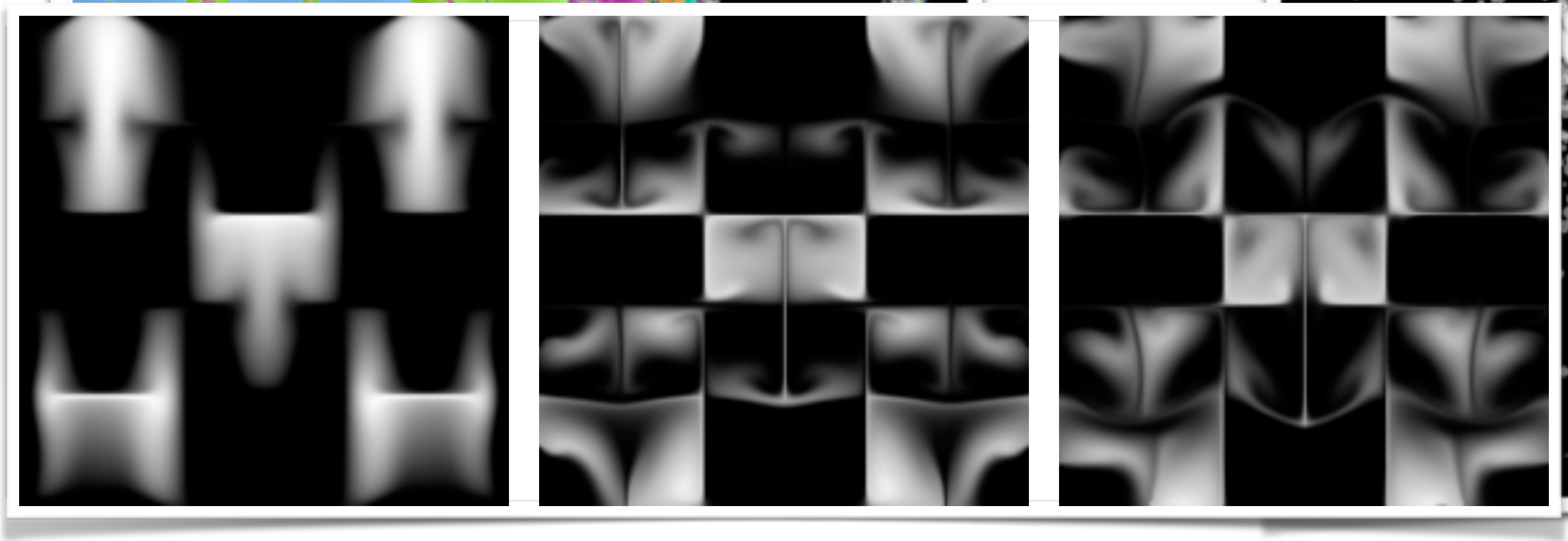
Mandelbrot fractal



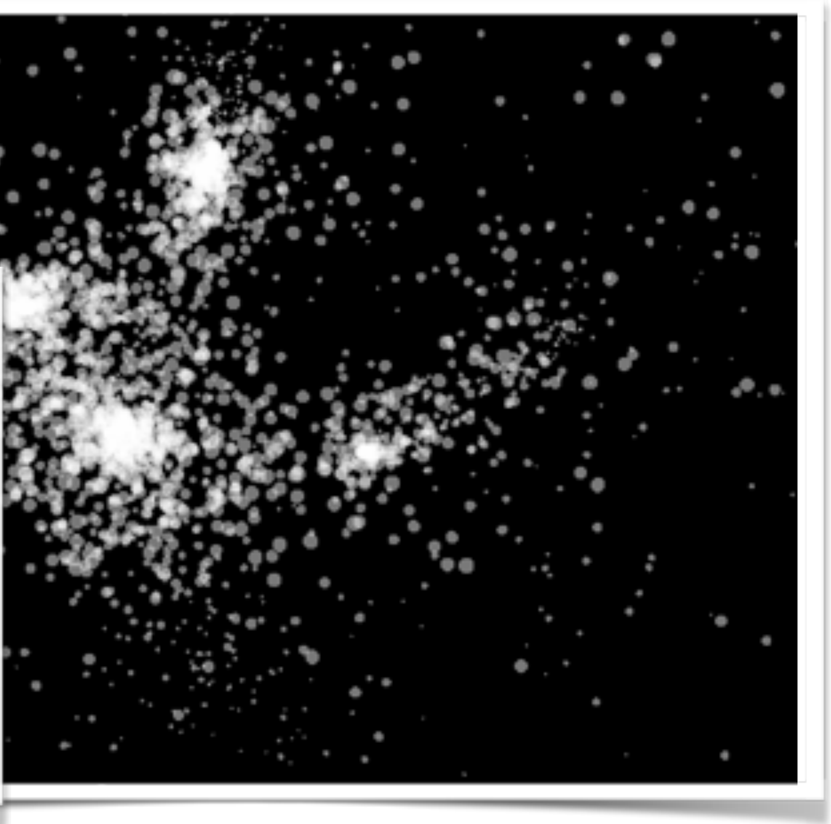
n-body gravitational simulation

Canny edge detection



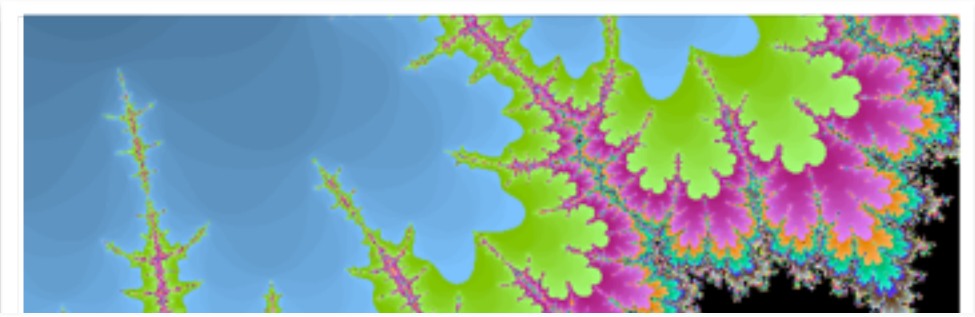


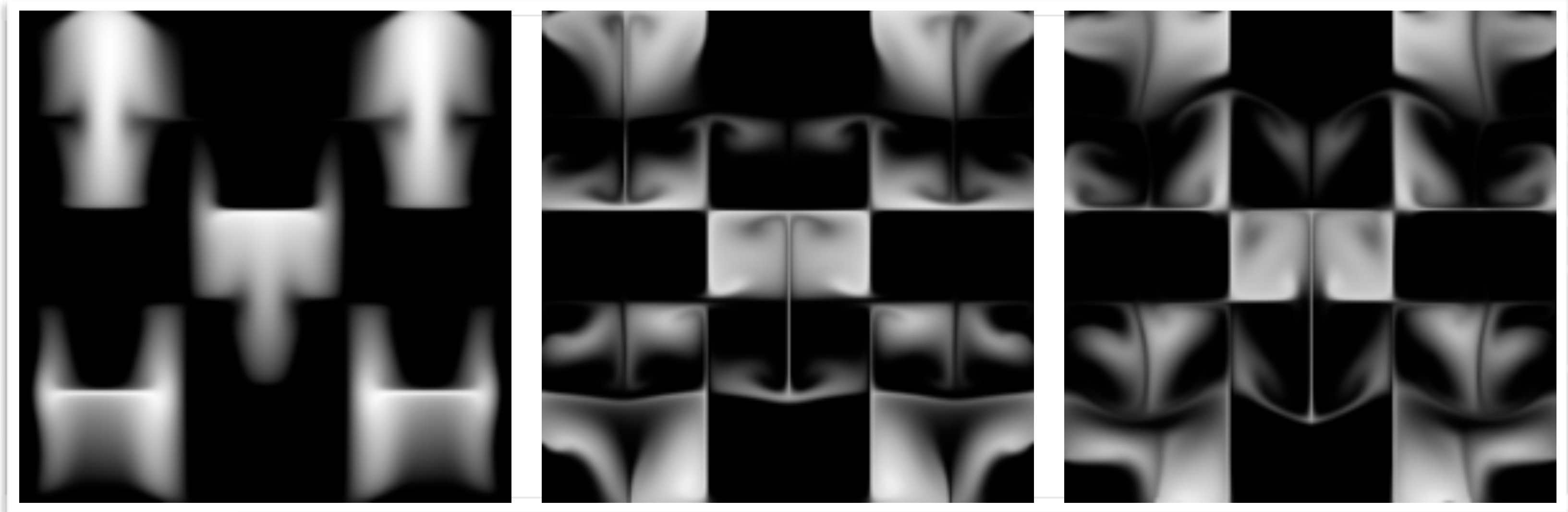
stable fluid flow



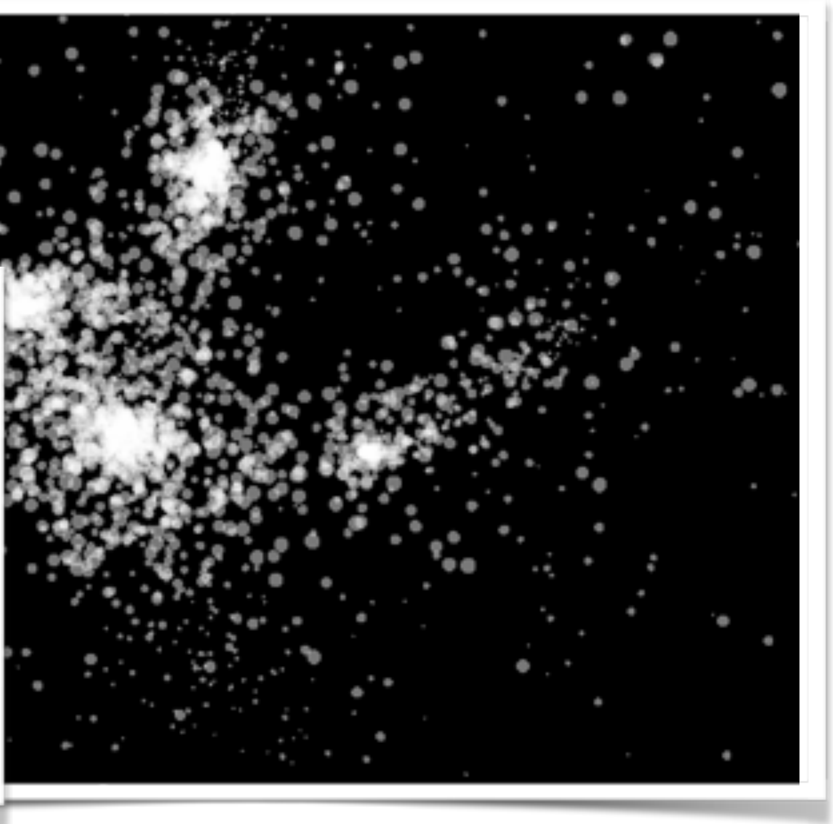
n-body gravitational simulation

Canny edge detection

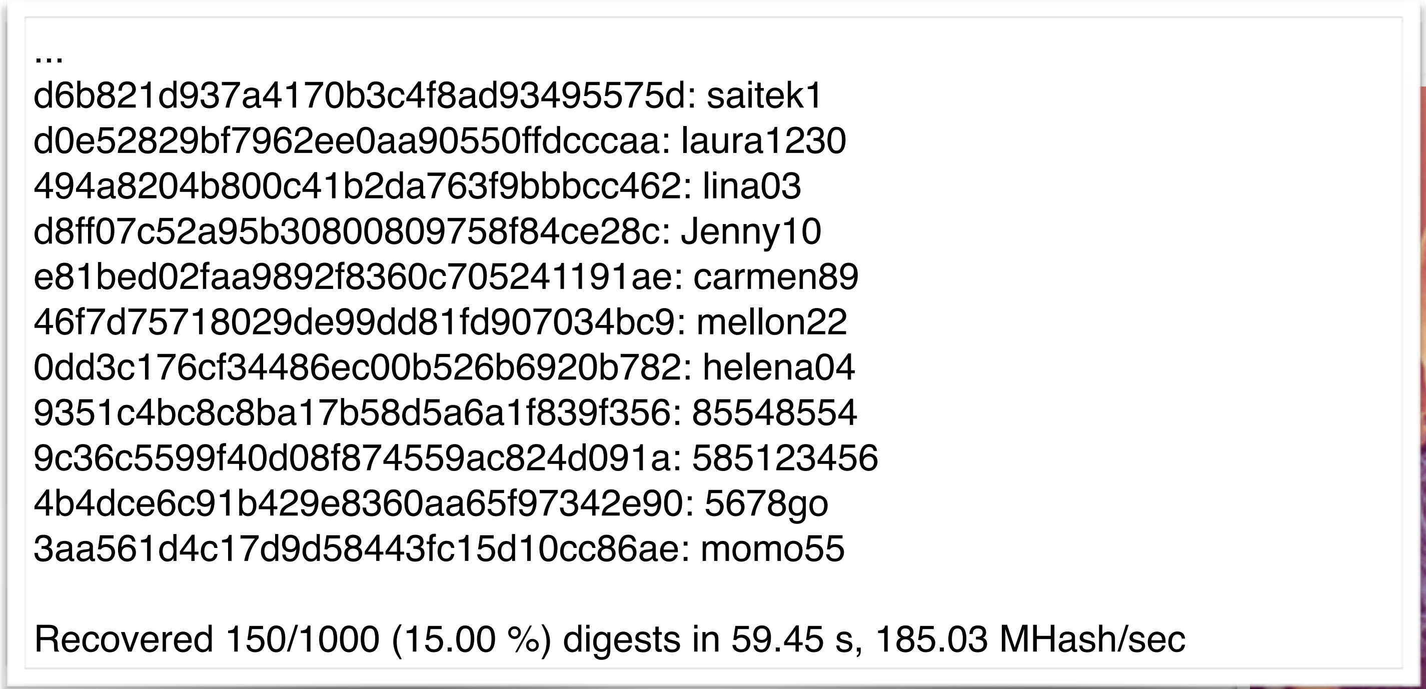




stable fluid flow

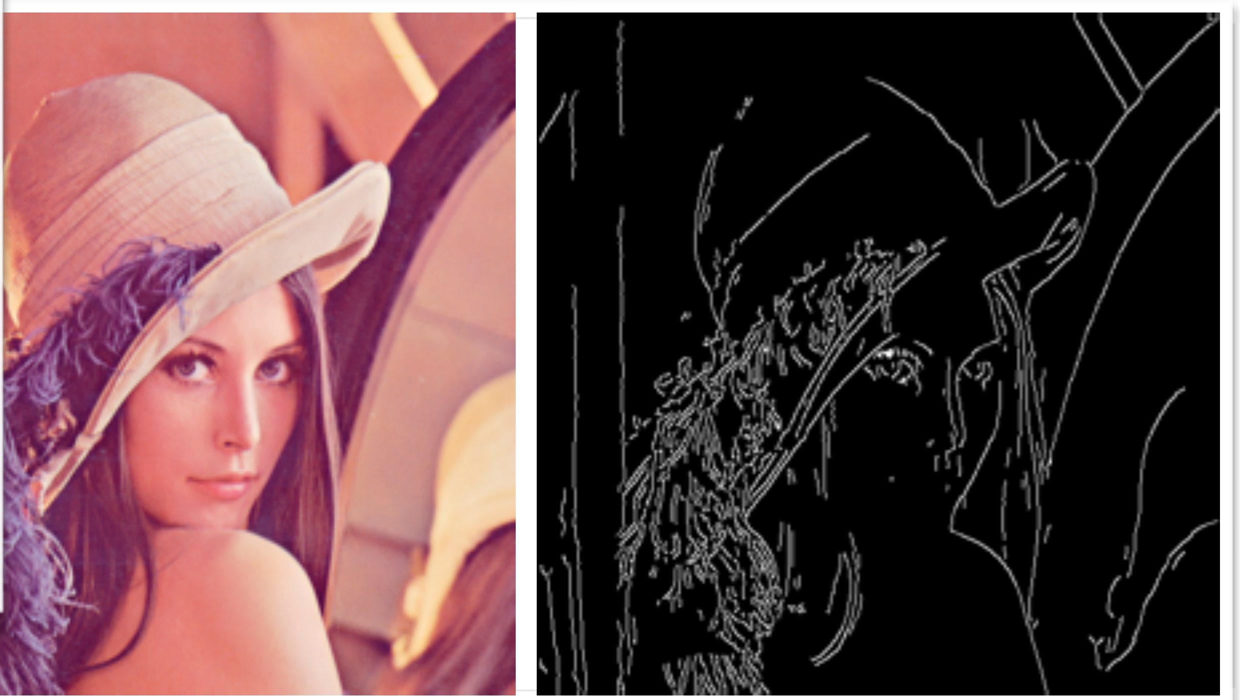


n-body gravitational simulation



```
...
d6b821d937a4170b3c4f8ad93495575d: saitek1
d0e52829bf7962ee0aa90550ffdcccaa: laura1230
494a8204b800c41b2da763f9bbbcc462: lina03
d8ff07c52a95b30800809758f84ce28c: Jenny10
e81bed02faa9892f8360c705241191ae: carmen89
46f7d75718029de99dd81fd907034bc9: mellon22
0dd3c176cf34486ec00b526b6920b782: helena04
9351c4bc8c8ba17b58d5a6a1f839f356: 85548554
9c36c5599f40d08f874559ac824d091a: 585123456
4b4dce6c91b429e8360aa65f97342e90: 5678go
3aa561d4c17d9d58443fc15d10cc86ae: momo55

Recovered 150/1000 (15.00 %) digests in 59.45 s, 185.03 MHash/sec
```

Password “recovery” (MD5 dictionary attack)



Canny edge detection

What's missing?

What’s missing?

- Matrix-vector multiplication.

What’s missing?

- Matrix-vector multiplication.
- In terms of `dotp`?

What’s missing?

- Matrix-vector multiplication.

- In terms of `dotp`?

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

What’s missing?

- Matrix-vector multiplication.

- In terms of `dotp`?

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
generate :: Exp sh
         -> (Exp sh -> Exp e)
         -> Acc (Array sh e)
```

What's missing?

- Matrix-vector multiplication.

- In terms of `dotp`?

```
index1 :: Exp Int -> Exp (Z:.Int)
```

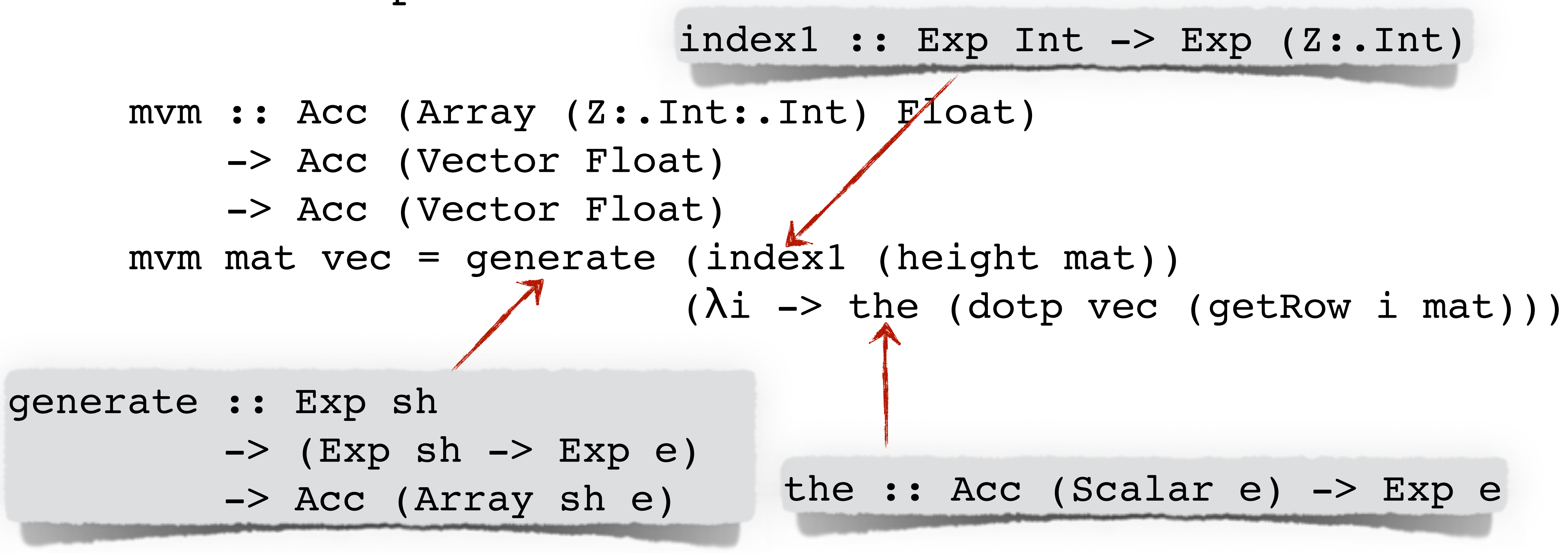
```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
generate :: Exp sh
         -> (Exp sh -> Exp e)
         -> Acc (Array sh e)
```

What's missing?

- Matrix-vector multiplication.

- In terms of `dotp`?



```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

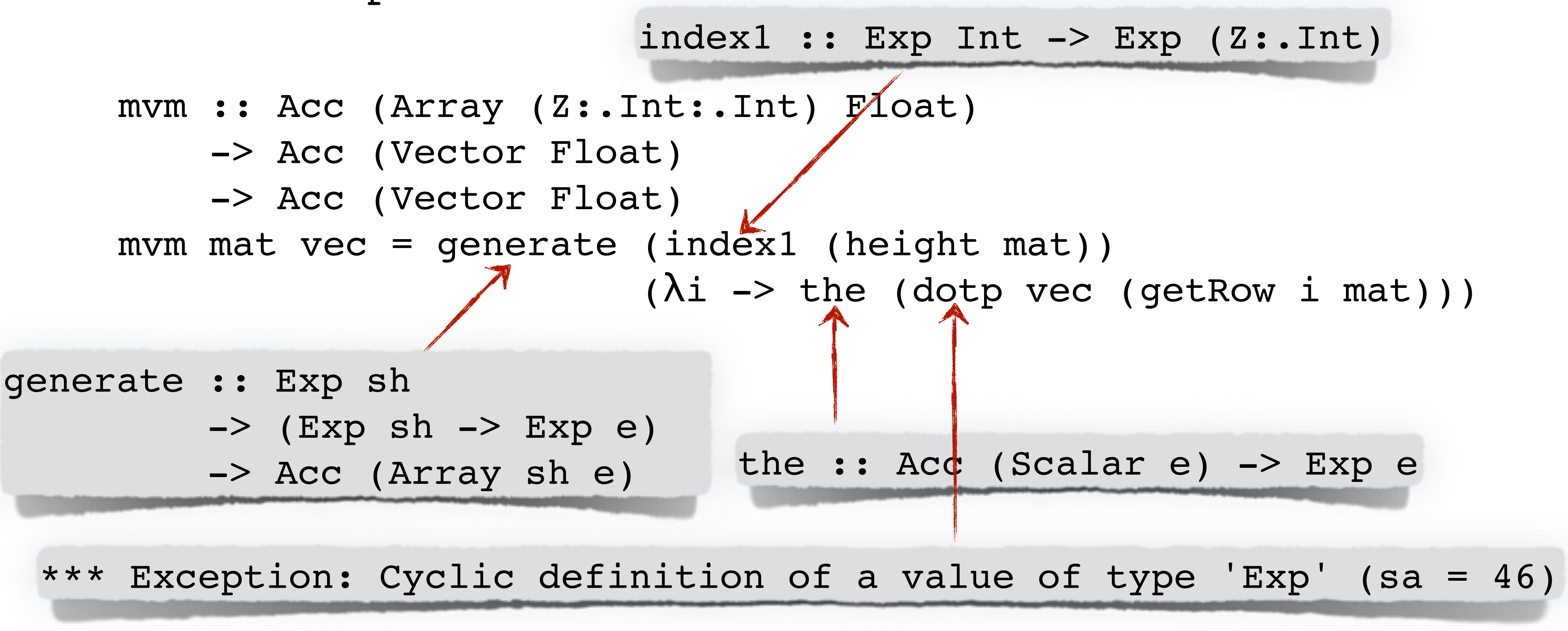
```
generate :: Exp sh
         -> (Exp sh -> Exp e)
         -> Acc (Array sh e)
```

```
the :: Acc (Scalar e) -> Exp e
```

What’s missing?

- Matrix-vector multiplication.

- In terms of `dotp`?



```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
generate :: Exp sh
         -> (Exp sh -> Exp e)
         -> Acc (Array sh e)
```

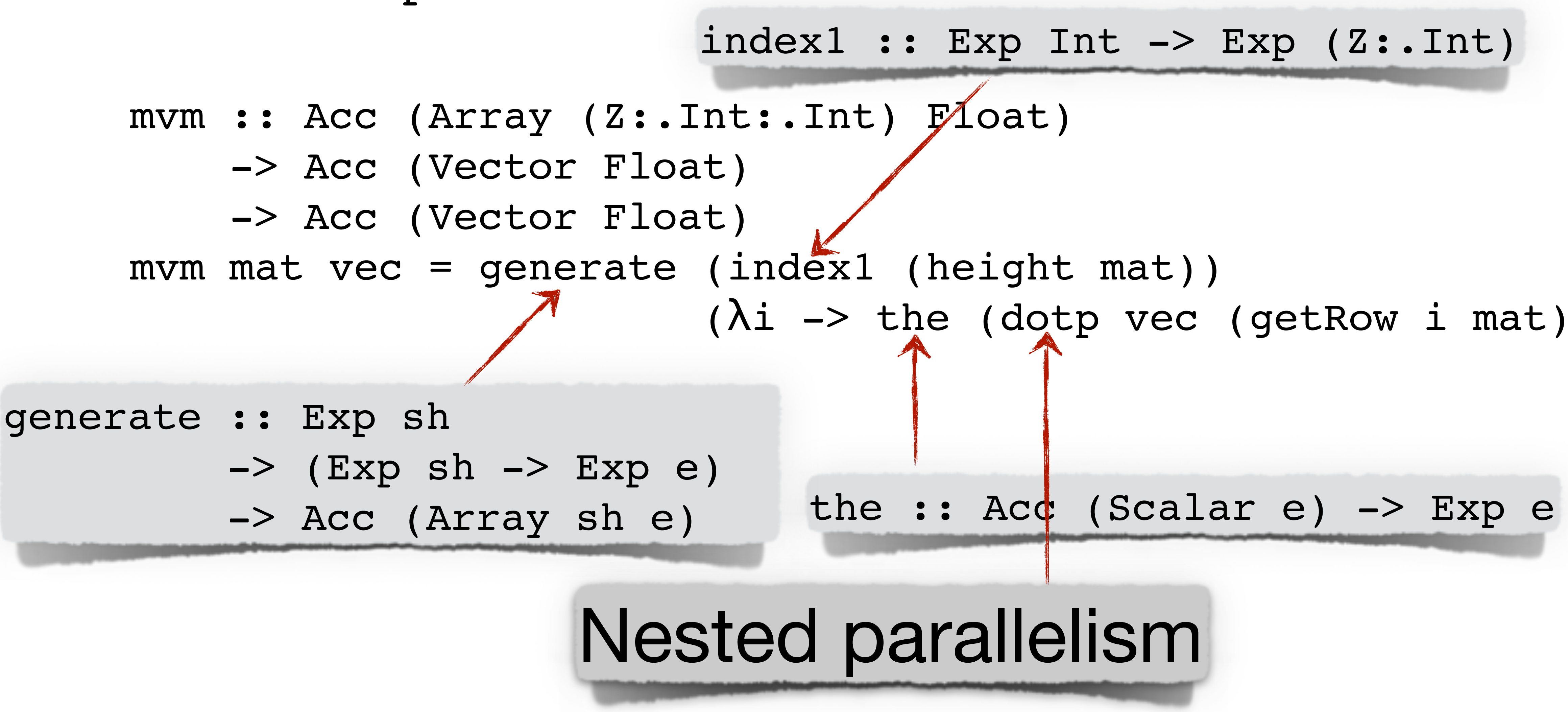
```
the :: Acc (Scalar e) -> Exp e
```

```
*** Exception: Cyclic definition of a value of type 'Exp' (sa = 46)
```

What's missing?

- Matrix-vector multiplication.

- In terms of `dotp`?



```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

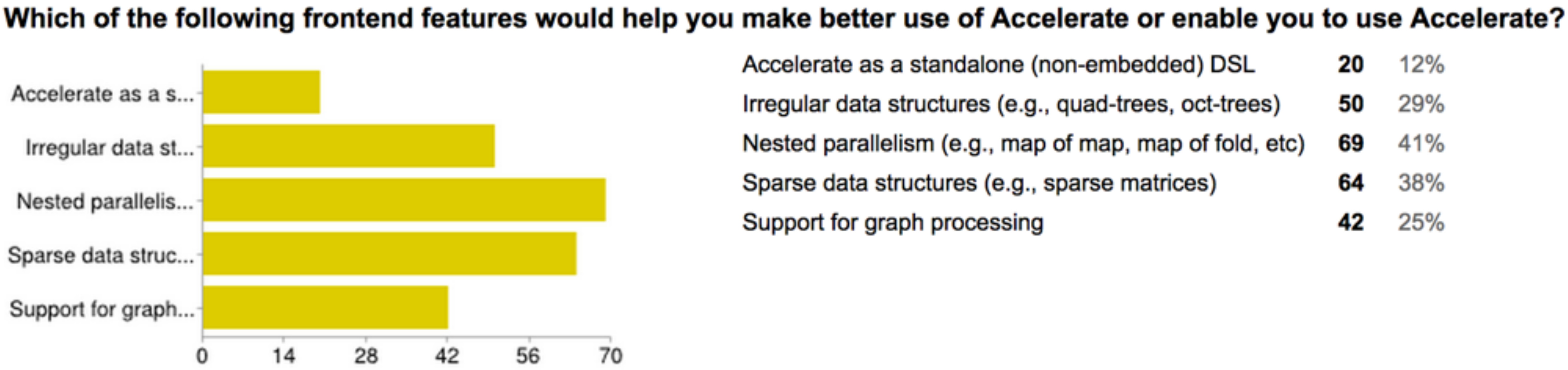
```
generate :: Exp sh
         -> (Exp sh -> Exp e)
         -> Acc (Array sh e)
```

```
the :: Acc (Scalar e) -> Exp e
```

Nested parallelism

Nested Parallelism

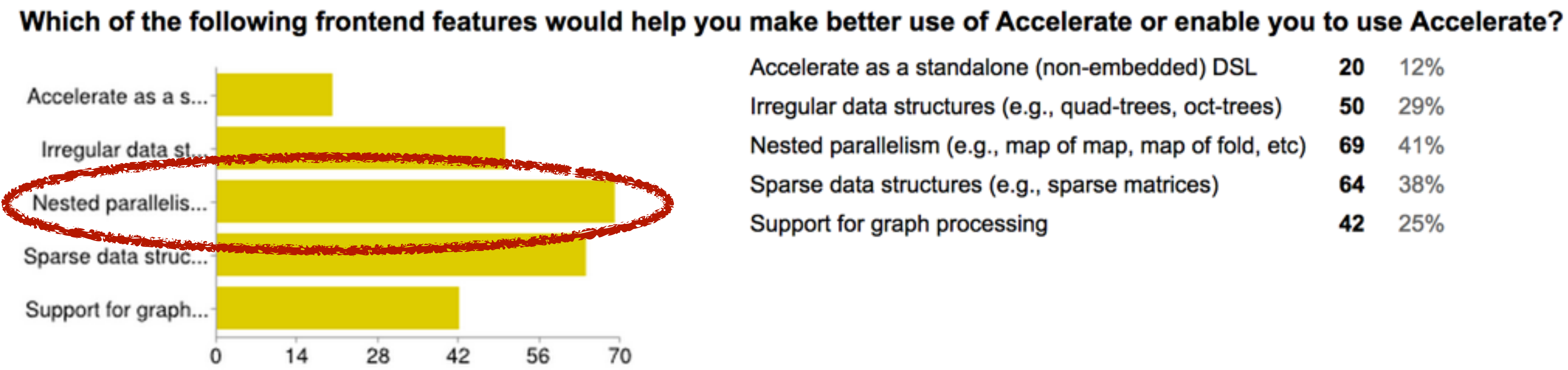
Nested Parallelism



Which of the following frontend features would help you make better use of Accelerate or enable you to use Accelerate?

Accelerate as a standalone (non-embedded) DSL	**20**	12%
Irregular data structures (e.g., quad-trees, oct-trees)	**50**	29%
Nested parallelism (e.g., map of map, map of fold, etc)	**69**	41%
Sparse data structures (e.g., sparse matrices)	**64**	38%
Support for graph processing	**42**	25%

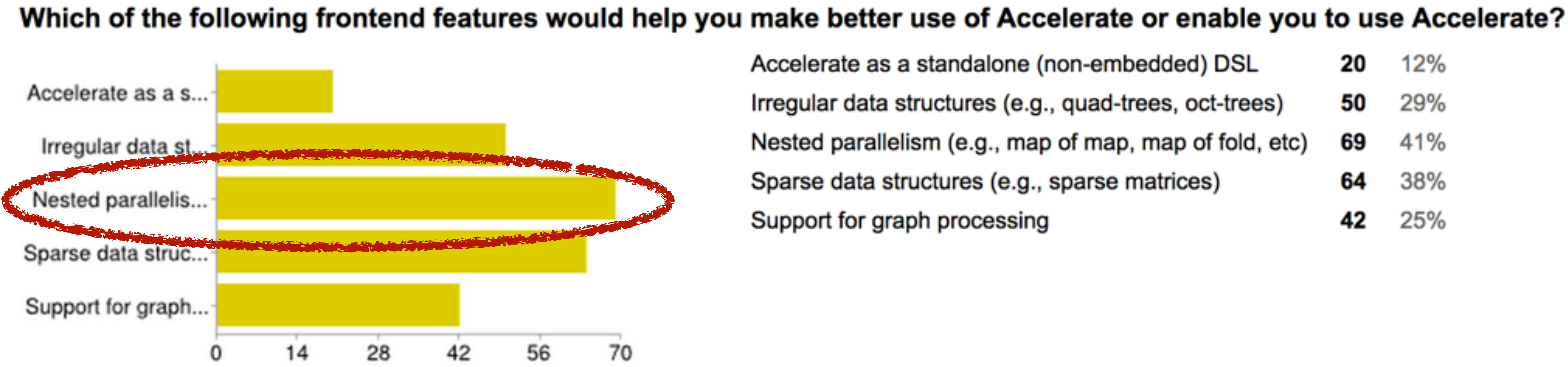
Nested Parallelism



Which of the following frontend features would help you make better use of Accelerate or enable you to use Accelerate?

Accelerate as a standalone (non-embedded) DSL	**20**	12%
Irregular data structures (e.g., quad-trees, oct-trees)	**50**	29%
Nested parallelism (e.g., map of map, map of fold, etc)	**69**	41%
Sparse data structures (e.g., sparse matrices)	**64**	38%
Support for graph processing	**42**	25%

Nested Parallelism

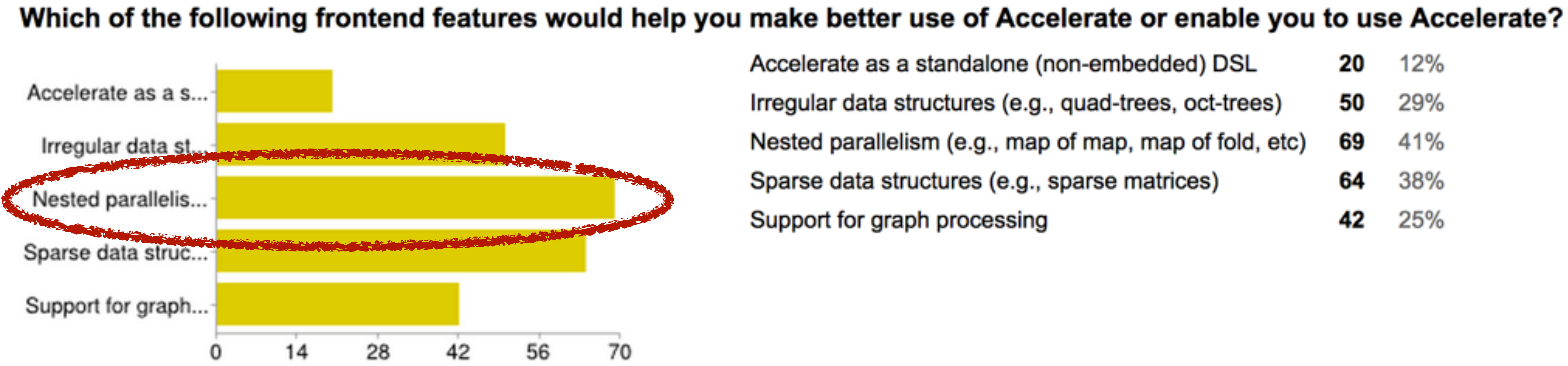


Which of the following frontend features would help you make better use of Accelerate or enable you to use Accelerate?

Accelerate as a standalone (non-embedded) DSL	**20**	12%
Irregular data structures (e.g., quad-trees, oct-trees)	**50**	29%
Nested parallelism (e.g., map of map, map of fold, etc)	**69**	41%
Sparse data structures (e.g., sparse matrices)	**64**	38%
Support for graph processing	**42**	25%

- NESL

Nested Parallelism



Which of the following frontend features would help you make better use of Accelerate or enable you to use Accelerate?

Accelerate as a standalone (non-embedded) DSL	**20**	12%
Irregular data structures (e.g., quad-trees, oct-trees)	**50**	29%
Nested parallelism (e.g., map of map, map of fold, etc)	**69**	41%
Sparse data structures (e.g., sparse matrices)	**64**	38%
Support for graph processing	**42**	25%

- NESL
- Data Parallel Haskell (DPH)

Nested Parallelism

Nested Operations

Nested Structures

Nested Parallelism

Nested Operations

MVM

Nested Structures

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)
```

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)
```

```
Array DIM2 (Vector e)
```

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)
```

```
Array DIM2 (Vector e)
```

Trees

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

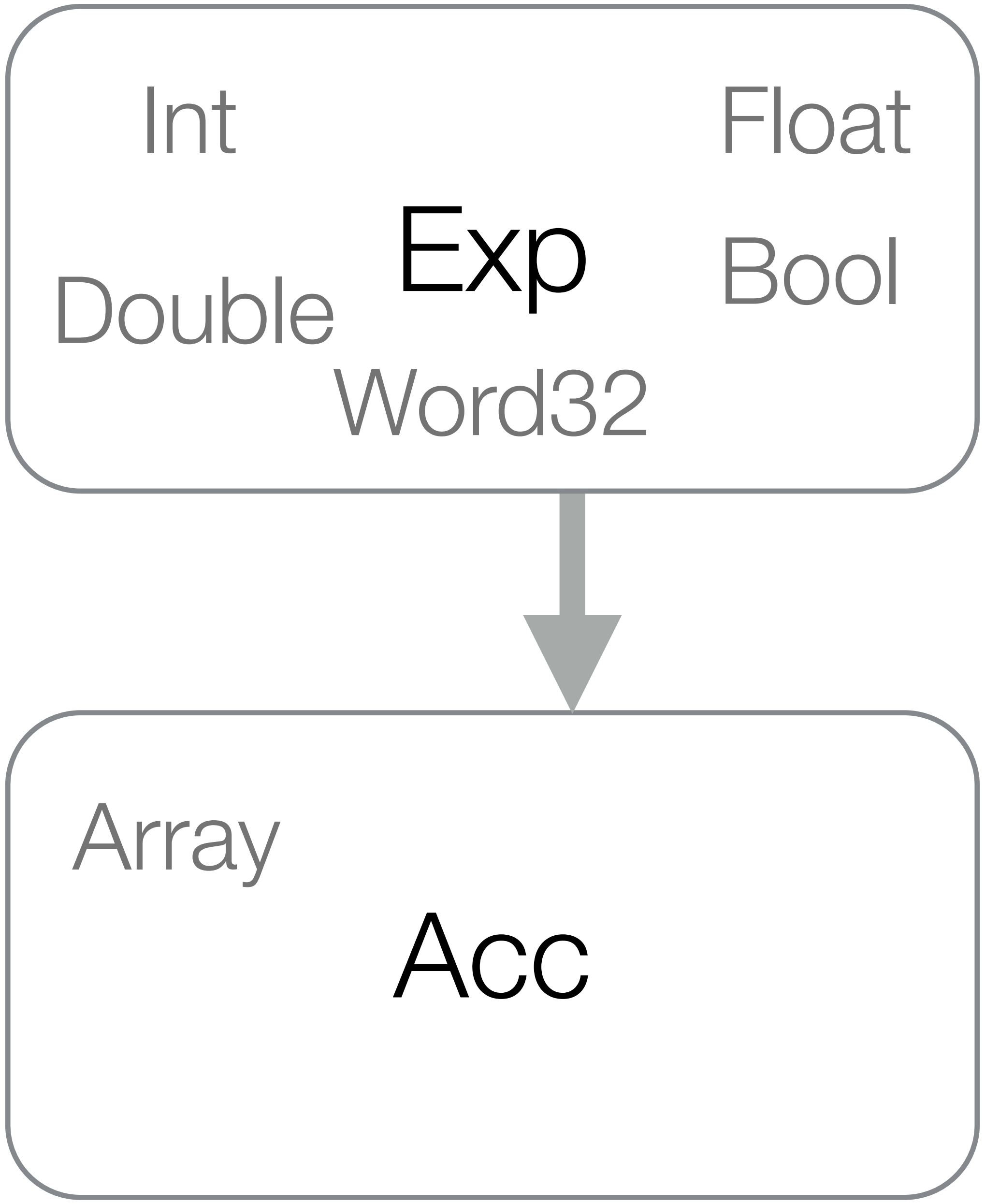
Nested Structures

```
Vector (Vector e)
```

```
Array DIM2 (Vector e)
```

Trees

Stratification


Int

Float

Exp

Double

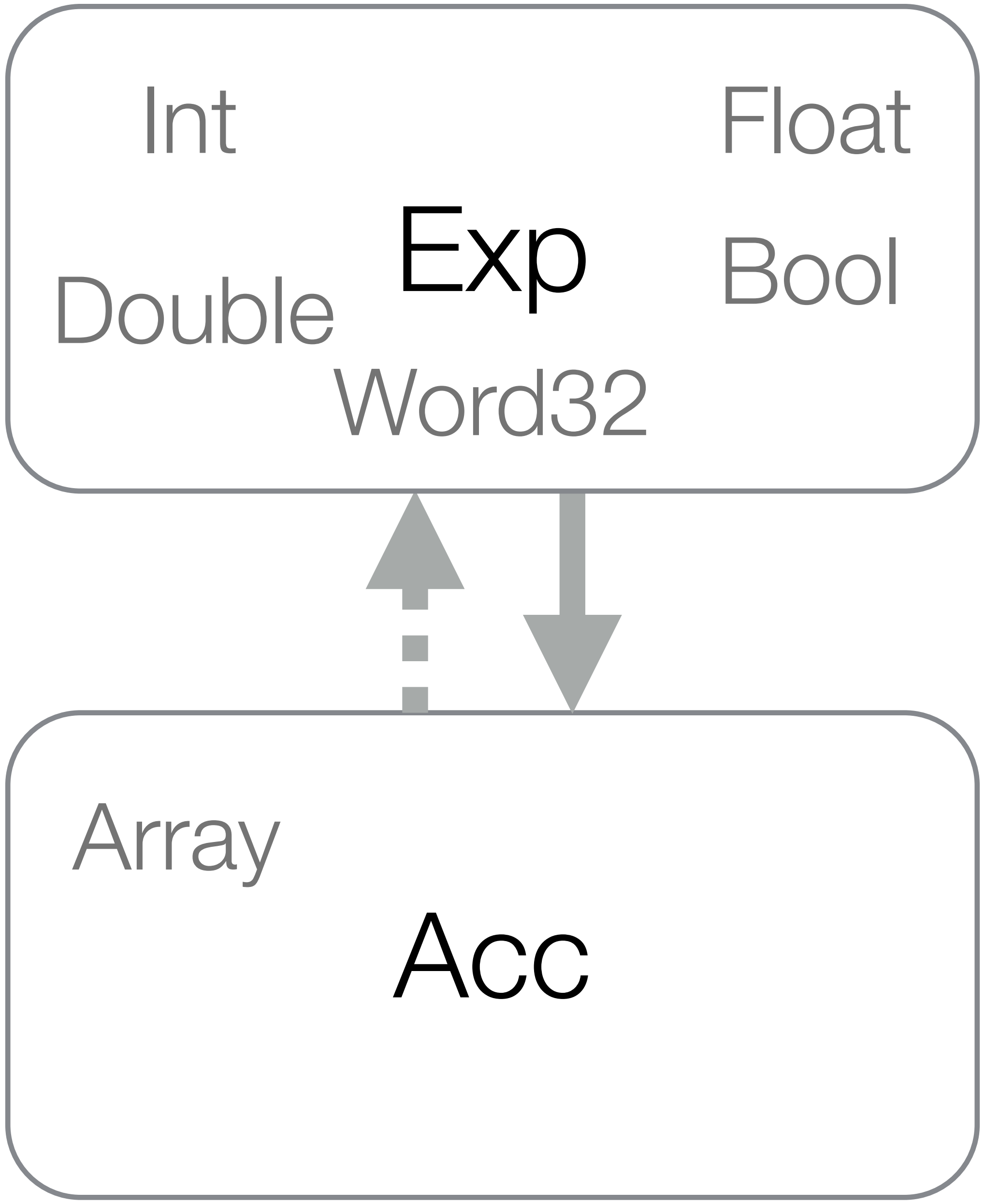
Bool

Word32

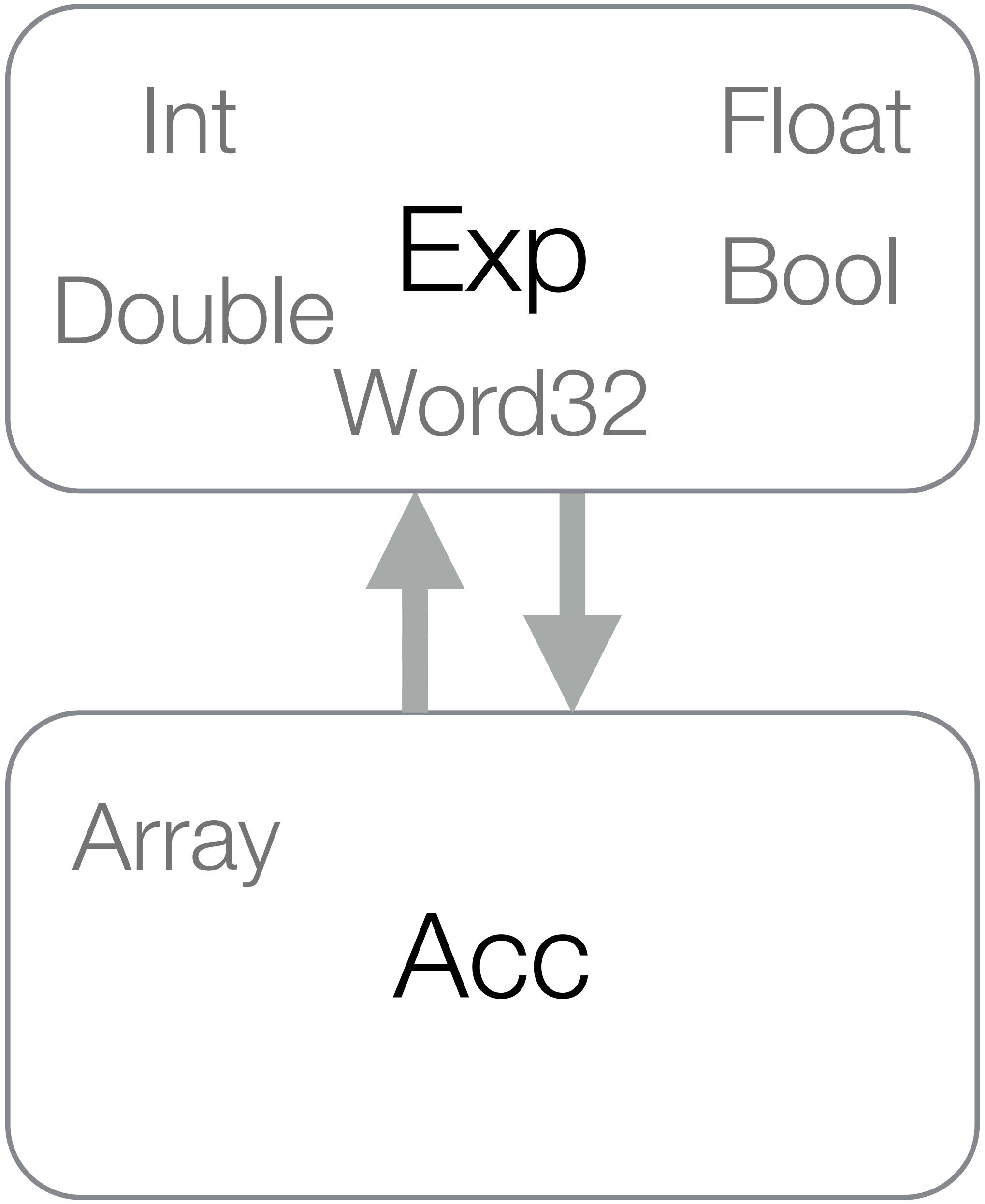
Array

Acc

Stratification



Stratification



Enabling nested parallelism

Enabling nested parallelism

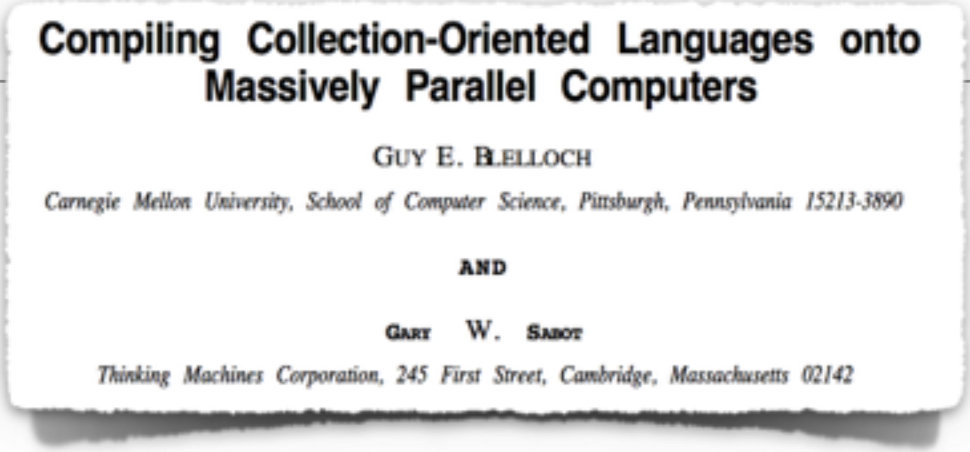
- Vectorisation (flattening)

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

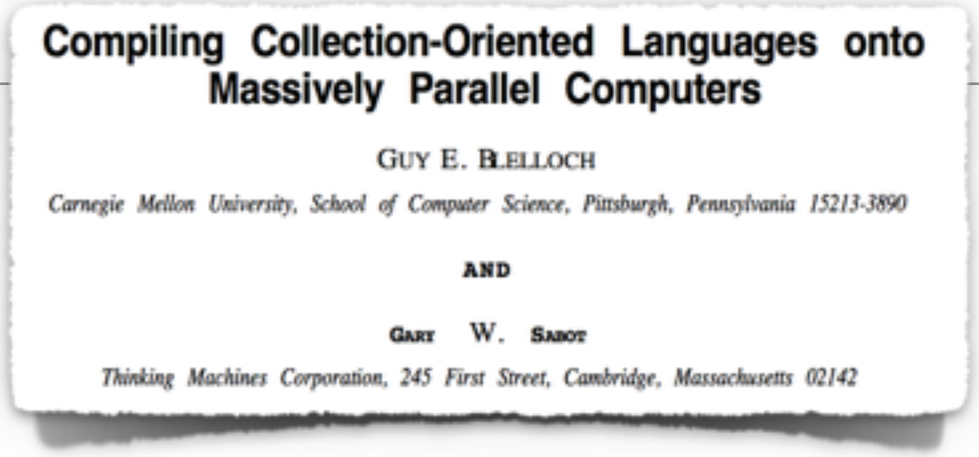
AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism

- Vectorisation (flattening)
    - First described by Blelloch and Sabot
    - Converts a nested parallel program into a flat parallel program



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

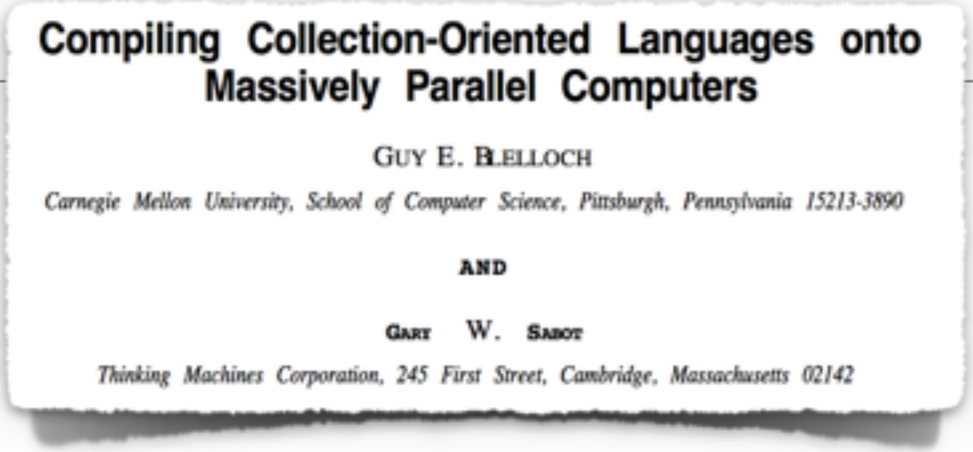
AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.



Compiling Collection-Oriented Languages onto Massively Parallel Computers

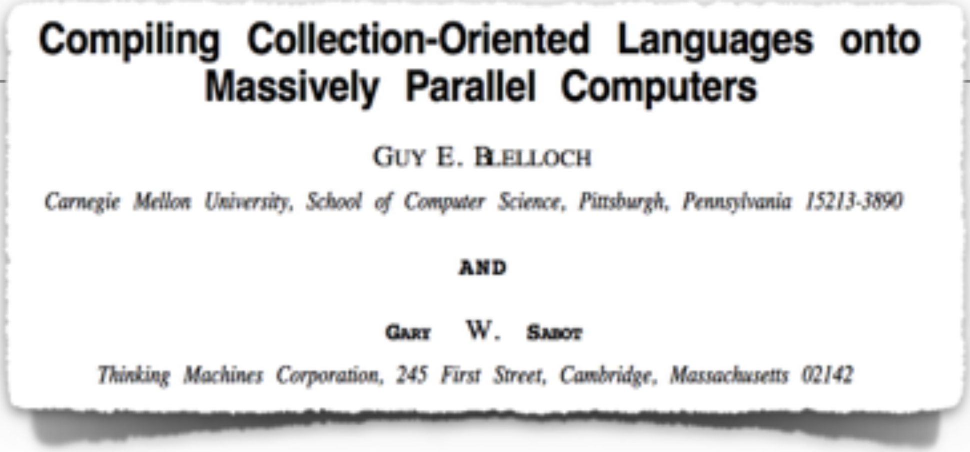
GUY E. BLELLOCH
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT
Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

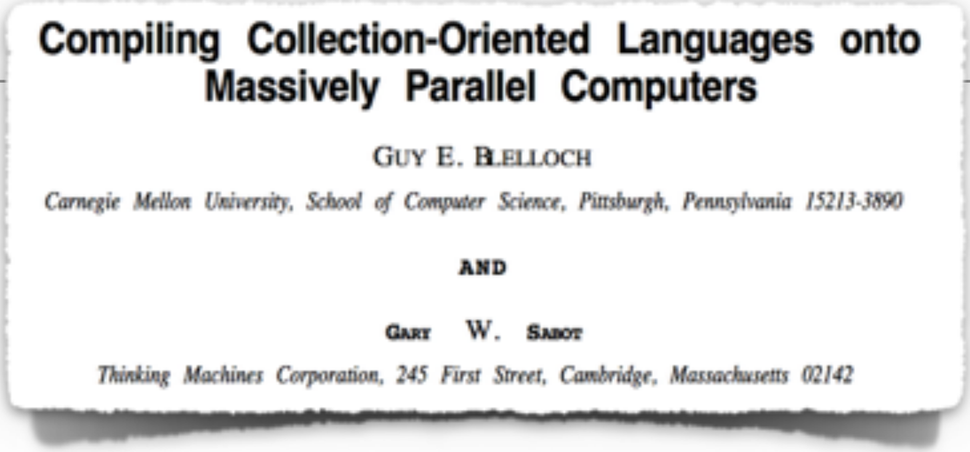
AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Complexity problems



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

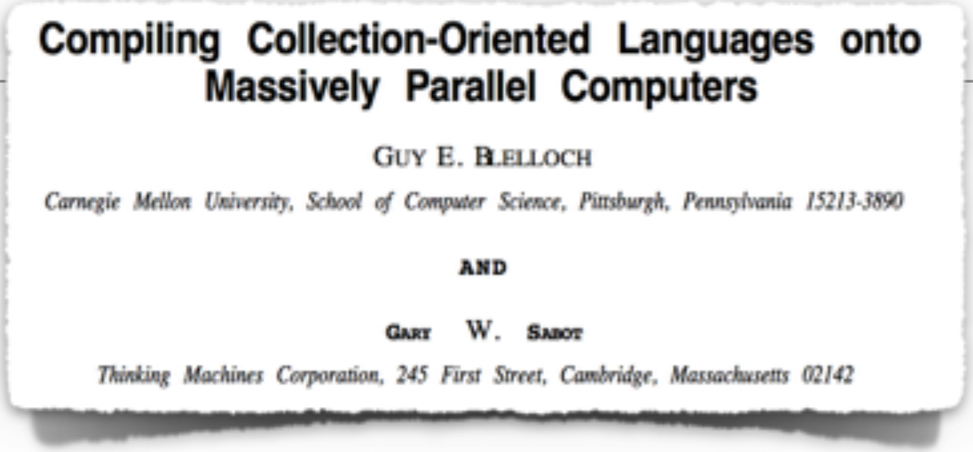
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

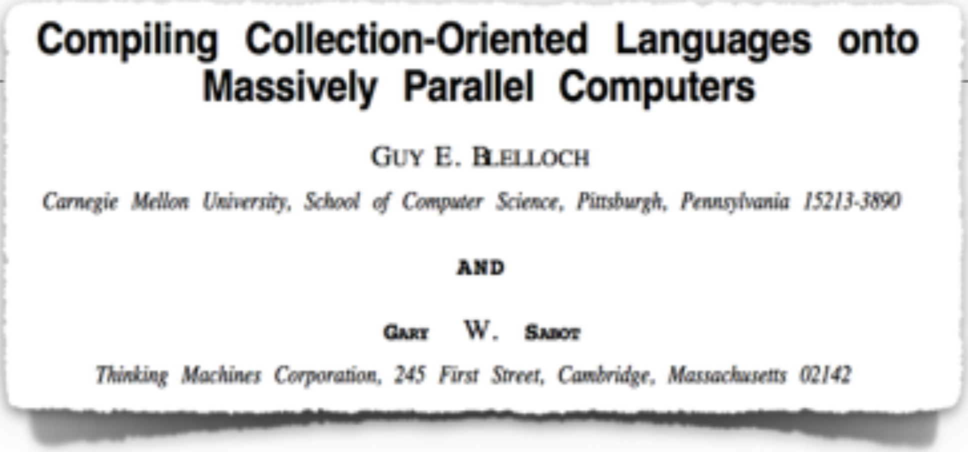
GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Complexity problems
  - Focus of more recent work

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Complexity problems
  - Focus of more recent work



Compiling Collection-Oriented Languages onto Massively Parallel Computers

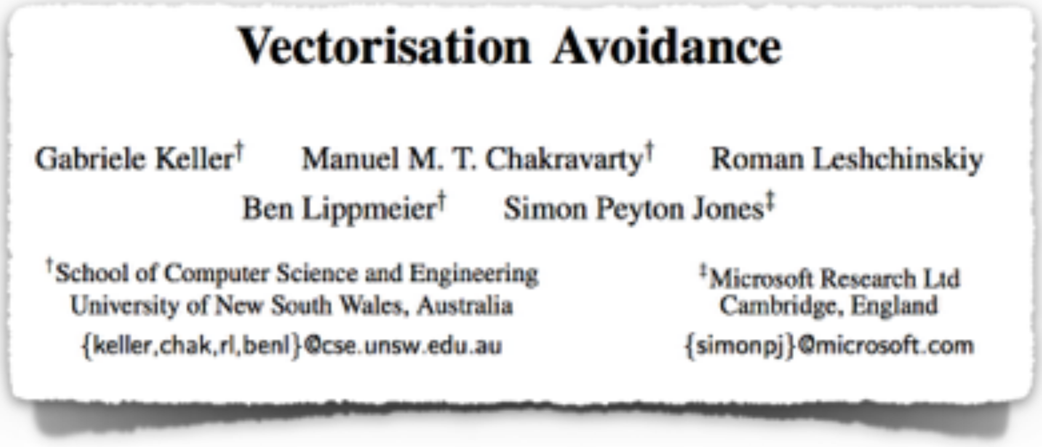
GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142



Vectorisation Avoidance

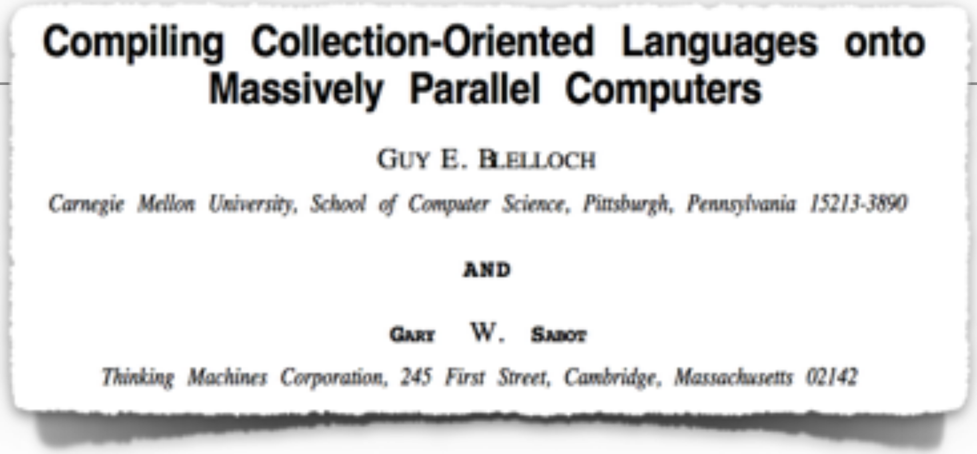
Gabriele Keller† Manuel M. T. Chakravarty† Roman Leshchinskiy
Ben Lippmeier† Simon Peyton Jones‡

†School of Computer Science and Engineering
University of New South Wales, Australia
{keller,chak,rl,benl}@cse.unsw.edu.au

‡Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Complexity problems
  - Focus of more recent work



Compiling Collection-Oriented Languages onto Massively Parallel Computers

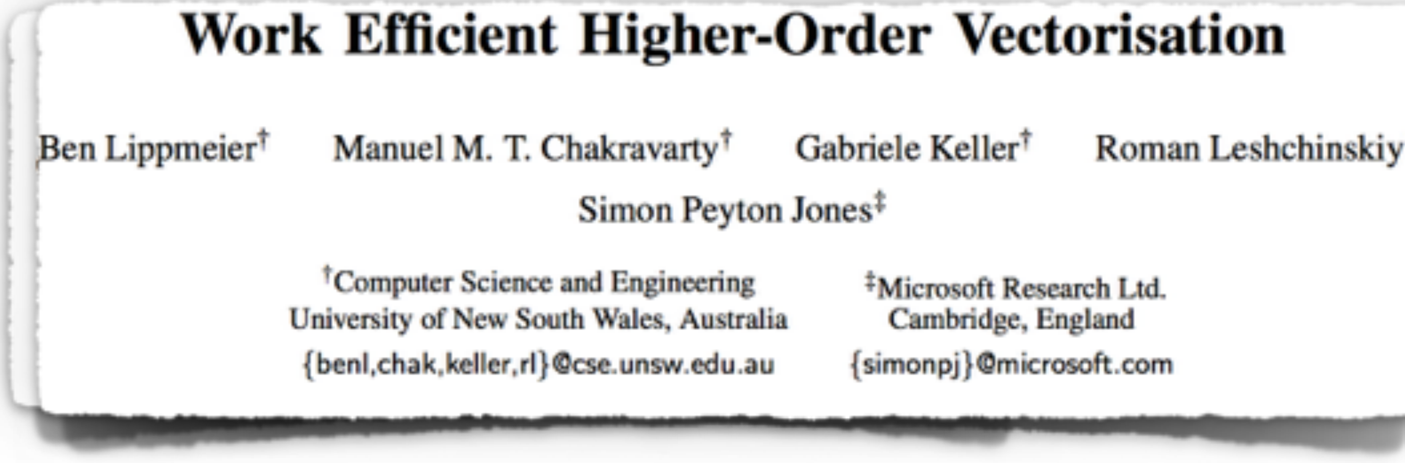
GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142



Work Efficient Higher-Order Vectorisation

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Roman Leshchinskiy

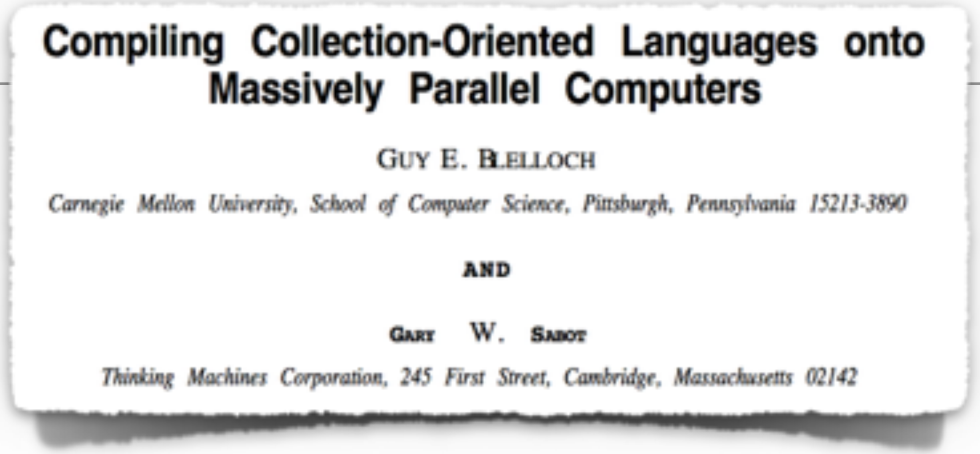
Simon Peyton Jones[‡]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,rl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd.
Cambridge, England
{simonpj}@microsoft.com

Enabling nested parallelism

- Vectorisation (flattening)
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Complexity problems
  - Focus of more recent work



Compiling Collection-Oriented Languages onto Massively Parallel Computers

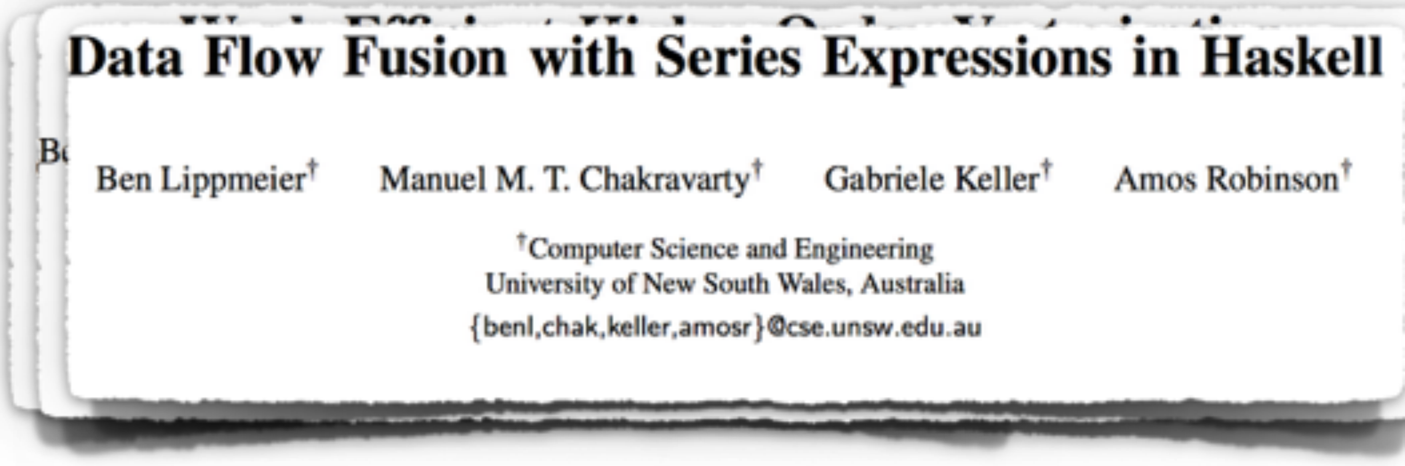
Guy E. Blelloch

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

Gary W. Sabot

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142



Data Flow Fusion with Series Expressions in Haskell

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Amos Robinson[†]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,amosr}@cse.unsw.edu.au

The lifting transformation

`foo     :: Int -> Float -> Float`

$\mathcal{L}_n[\![\mathtt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The lifting transformation

```
foo     :: Int -> Float -> Float
```

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The expression being transformed

The lifting transformation

`foo     :: Int -> Float -> Float`

$\mathcal{L}_{n}[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size

The expression being transformed

The lifting transformation

foo :: Int -> Float -> Float

$\mathcal{L}_n[\![\mathtt{foo}]\!]$ :: Vector Int -> Vector Float -> Vector Float

The size

The expression being transformed

$\mathcal{L}_n[\![\mathtt{c}]\!]$ = replicate n c (Where c is a constant)

The lifting transformation

`foo     :: Int -> Float -> Float`

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size

The expression being transformed

$\mathcal{L}_n[\![\texttt{c}]\!]$ `= replicate n c` (Where c is a constant)

$\mathcal{L}_n[\![\texttt{x}]\!]$ `= replicate n x` (Where x is not a lifted variable)

The lifting transformation

`foo     :: Int -> Float -> Float`

$\mathcal{L}_{n}[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size

The expression being transformed

$\mathcal{L}_{n}[\![c]\!]$	`= replicate n c`	(Where c is a constant)
$\mathcal{L}_{n}[\![x]\!]$	`= replicate n x`	(Where x is not a lifted variable)
$\mathcal{L}_{n}[\![x]\!]$	`= x`	(Where x is a lifted variable)

The lifting transformation

$\texttt{foo} \quad :: \texttt{Int -> Float -> Float}$

$\mathcal{L}_n[\![\texttt{foo}]\!] :: \texttt{Vector Int -> Vector Float -> Vector Float}$

The size

The expression being transformed

$\mathcal{L}_n[\![\texttt{c}]\!] = \texttt{replicate n c}$ (Where c is a constant)

$\mathcal{L}_n[\![\texttt{x}]\!] = \texttt{replicate n x}$ (Where x is not a lifted variable)

$\mathcal{L}_n[\![\texttt{x}]\!] = \texttt{x}$ (Where x is a lifted variable)

$\mathcal{L}_n[\![\texttt{e1 e2}]\!] = \mathcal{L}_n[\![\texttt{e1}]\!]\ \mathcal{L}_n[\![\texttt{e2}]\!]$

The lifting transformation

$\texttt{foo} \quad :: \texttt{Int -> Float -> Float}$

$\mathcal{L}_{n}[\![\texttt{foo}]\!] :: \texttt{Vector Int -> Vector Float -> Vector Float}$

The size

The expression being transformed

$\mathcal{L}_{n}[\![\texttt{c}]\!] = \texttt{replicate n c}$ (Where c is a constant)

$\mathcal{L}_{n}[\![\texttt{x}]\!] = \texttt{replicate n x}$ (Where x is not a lifted variable)

$\mathcal{L}_{n}[\![\texttt{x}]\!] = \texttt{x}$ (Where x is a lifted variable)

$\mathcal{L}_{n}[\![\texttt{e1 e2}]\!] = \mathcal{L}_{n}[\![\texttt{e1}]\!] \; \mathcal{L}_{n}[\![\texttt{e2}]\!]$

$\mathcal{L}_{n}[\![\lambda\texttt{x.e}]\!] = \lambda\texttt{x.} \; \mathcal{L}_{\texttt{(length x)}}[\![\texttt{e}]\!]$

The lifting transformation

`foo     :: Int -> Float -> Float`

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size

The expression being transformed

$\mathcal{L}_n[\![\texttt{c}]\!] = \texttt{replicate n c}$ (Where c is a constant)

$\mathcal{L}_n[\![\texttt{x}]\!] = \texttt{replicate n x}$ (Where x is not a lifted variable)

$\mathcal{L}_n[\![\texttt{x}]\!] = \texttt{x}$ (Where x is a lifted variable)

$\mathcal{L}_n[\![\texttt{e1 e2}]\!] = \mathcal{L}_n[\![\texttt{e1}]\!]\ \mathcal{L}_n[\![\texttt{e2}]\!]$

$\mathcal{L}_n[\![\lambda\texttt{x.e}]\!] = \lambda\texttt{x.}\ \mathcal{L}_{(\texttt{length x})}[\![\texttt{e}]\!]$

$\mathcal{L}_n[\![\texttt{p}]\!] = \texttt{p}^{\uparrow}$ (Where p is a built-in operation and $p^{\uparrow}$ is the lifted equivalent)

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

```
$\mathcal{L}_n[\![\text{bar}]\!]$ :: Vector Int -> Vector Int

$\mathcal{L}_n[\![\text{bar}]\!]$ = λx. (replicate (length x) 2) $*^{\uparrow}$ x $+^{\uparrow}$ (replicate (length x) 1)
```

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![\mathtt{bar}]\!]$ `:: Vector Int -> Vector Int`

$\mathcal{L}_n[\![\mathtt{bar}]\!]$ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

What about vector functions?

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

```
ℒn⟦bar⟧ :: Vector Int -> Vector Int

ℒn⟦bar⟧ = λx. (replicate (length x) 2) *↑ x +↑ (replicate (length x) 1)
```

What about vector functions?

```
sum :: Vector Int -> Int
```

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![\texttt{bar}]\!]\ \texttt{:: Vector Int -> Vector Int}$

$\mathcal{L}_n[\![\texttt{bar}]\!]\ \texttt{= λx. (replicate (length x) 2)}\ \ast^{\uparrow}\ \texttt{x}\ +^{\uparrow}\ \texttt{(replicate (length x) 1)}$

What about vector functions?

```
sum :: Vector Int -> Int
```

$\mathcal{L}_n[\![\texttt{sum}]\!]\ \texttt{:: Vector (Vector Int) -> Vector Int}$

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

```
ℒn⟦bar⟧ :: Vector Int -> Vector Int

ℒn⟦bar⟧ = λx. (replicate (length x) 2) *↑ x +↑ (replicate (length x) 1)
```

What about vector functions?

```
sum :: Vector Int -> Int

ℒn⟦sum⟧ :: Vector (Vector Int) -> Vector Int
```

Nested vectors

Nested vectors

Nested vectors

- Vectors of pointers? Grossly inefficient.

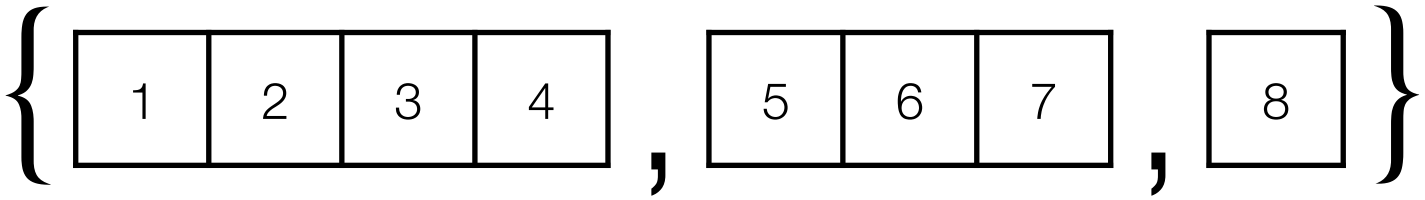
Nested vectors

- Vectors of pointers? Grossly inefficient.

- Blelloch’s solution

Nested vectors

- Vectors of pointers? Grossly inefficient.
- Blelloch's solution

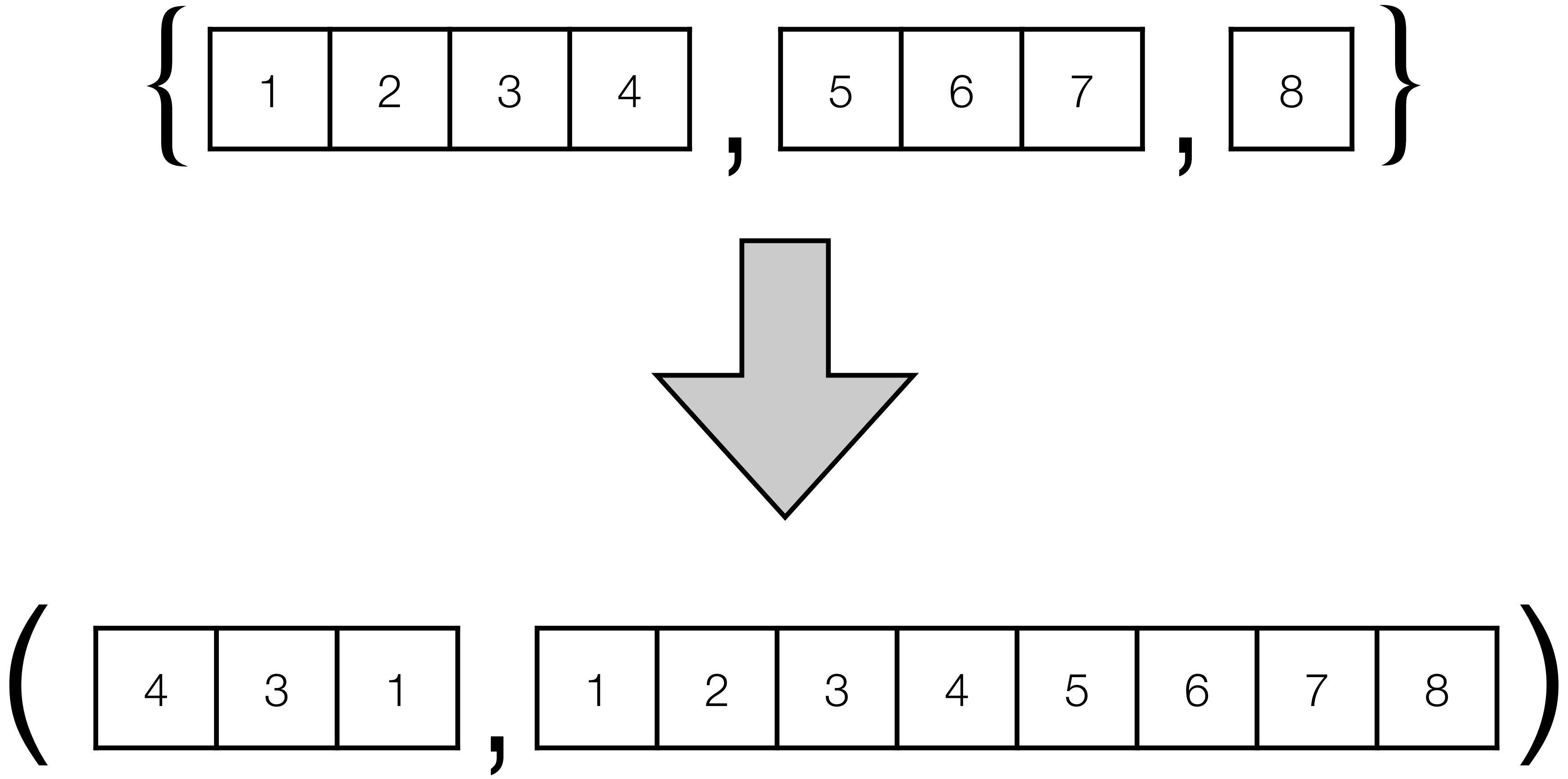


{ 1 2 3 4 , 5 6 7 , 8 }

Nested vectors

- Vectors of pointers? Grossly inefficient.

- Blelloch’s solution



{ 1 2 3 4 , 5 6 7 , 8 }

(4 3 1 , 1 2 3 4 5 6 7 8)

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?
- Does it now require this?

```
foo     :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo      :: Int -> Float -> Float
ℒn⟦foo⟧  :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

`foo     :: Int -> Float -> Float`

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Array sh Int -> Array sh Float -> Array sh Float`

- No, lifting to vectors is sufficient
  - At the machine level it’s all vectors anyway

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo     :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
    - At the machine level it’s all vectors anyway

- What about nested arrays?

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo     :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
  - At the machine level it’s all vectors anyway

- What about nested arrays?
  - But, we only need vectors of arrays

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo    :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
  - At the machine level it’s all vectors anyway

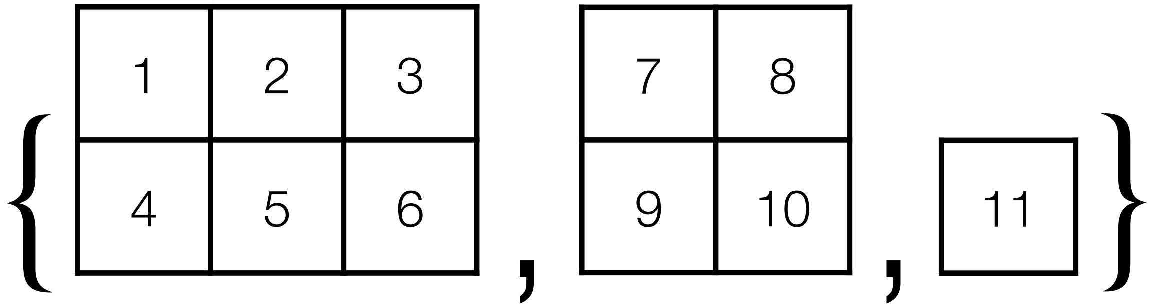
- What about nested arrays?
  - But, we only need vectors of arrays

Vectors of arrays

```
type Vector' = …a vector of arrays…
```

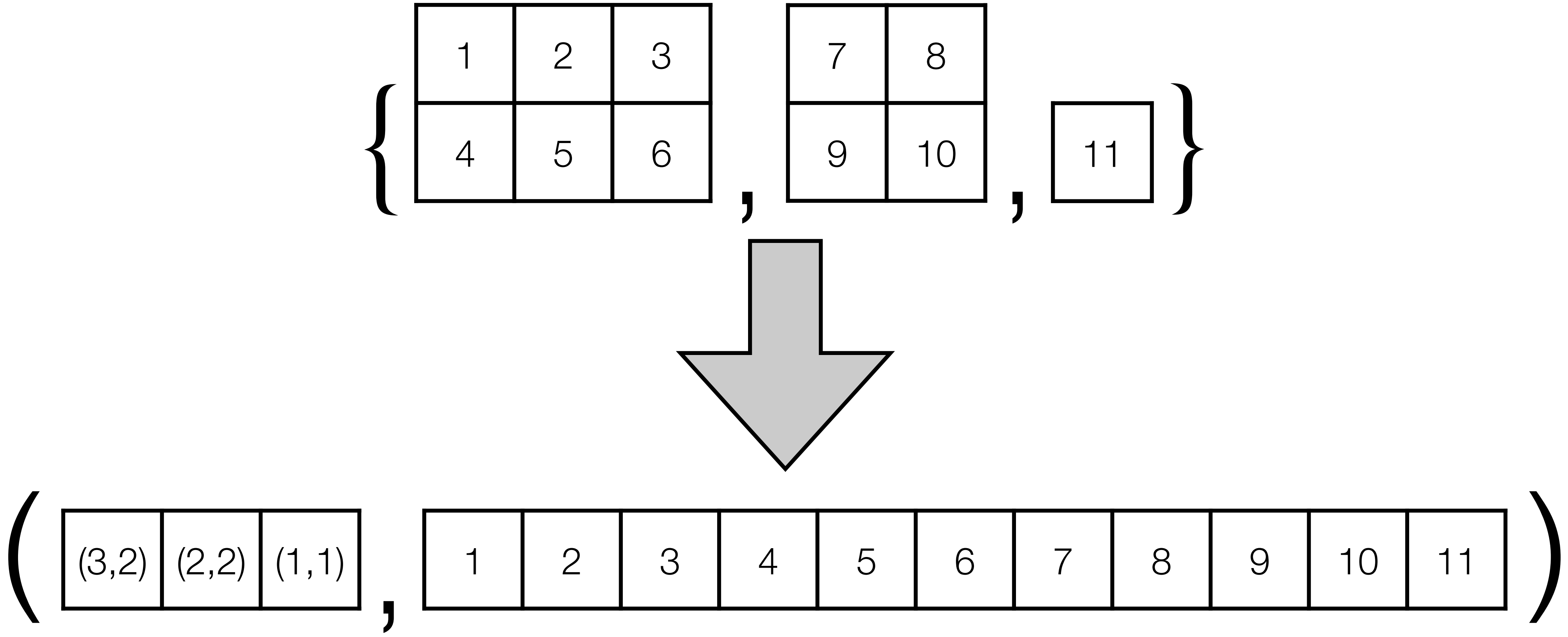
Vectors of arrays

```
type Vector' = …a vector of arrays…
```



Vectors of arrays

```
type Vector' = …a vector of arrays…
```



{ 1 2 3 / 4 5 6 , 7 8 / 9 10 , 11 }

((3,2) (2,2) (1,1) , 1 2 3 4 5 6 7 8 9 10 11)

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)

Array DIM2 (Vector e)
```

Trees

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)

Array DIM2 (Vector e)
```

Trees

Nested Parallelism

Nested Operations

MVM

```
fact n =
  map (\m -> product [1..m]) [1..n]
```

Nested Structures

```
Vector (Vector e)
```

```
Array DIM2 (Vector e)
```

Trees

Sequences

Sequences

Sequences

- Sequences of arrays (or tuples of arrays)

Sequences

- Sequences of arrays (or tuples of arrays)

- Can only be accessed linearly

Sequences

- Sequences of arrays (or tuples of arrays)
- Can only be accessed linearly
- So map, fold and scan, but no permuting, indexing or constant time length

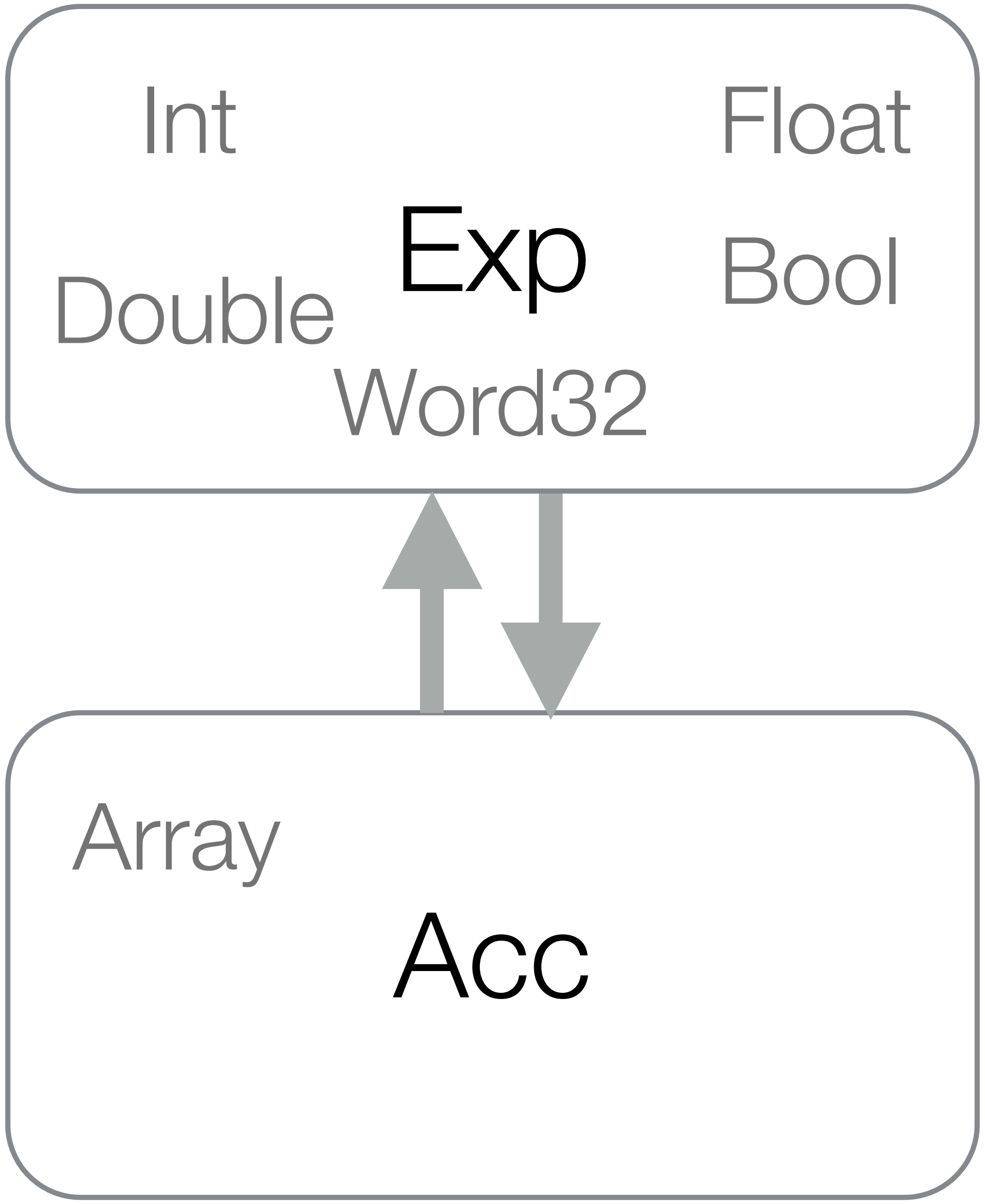
Sequences

- Sequences of arrays (or tuples of arrays)
- Can only be accessed linearly
- So map, fold and scan, but no permuting, indexing or constant time length
- Like Haskell lists

Sequences

- Sequences of arrays (or tuples of arrays)
- Can only be accessed linearly
- So map, fold and scan, but no permuting, indexing or constant time length
- Like Haskell lists
- Does that mean all the operations have to be made polymorphic over sequences and arrays?

Sequences


Int

Float

Exp

Double

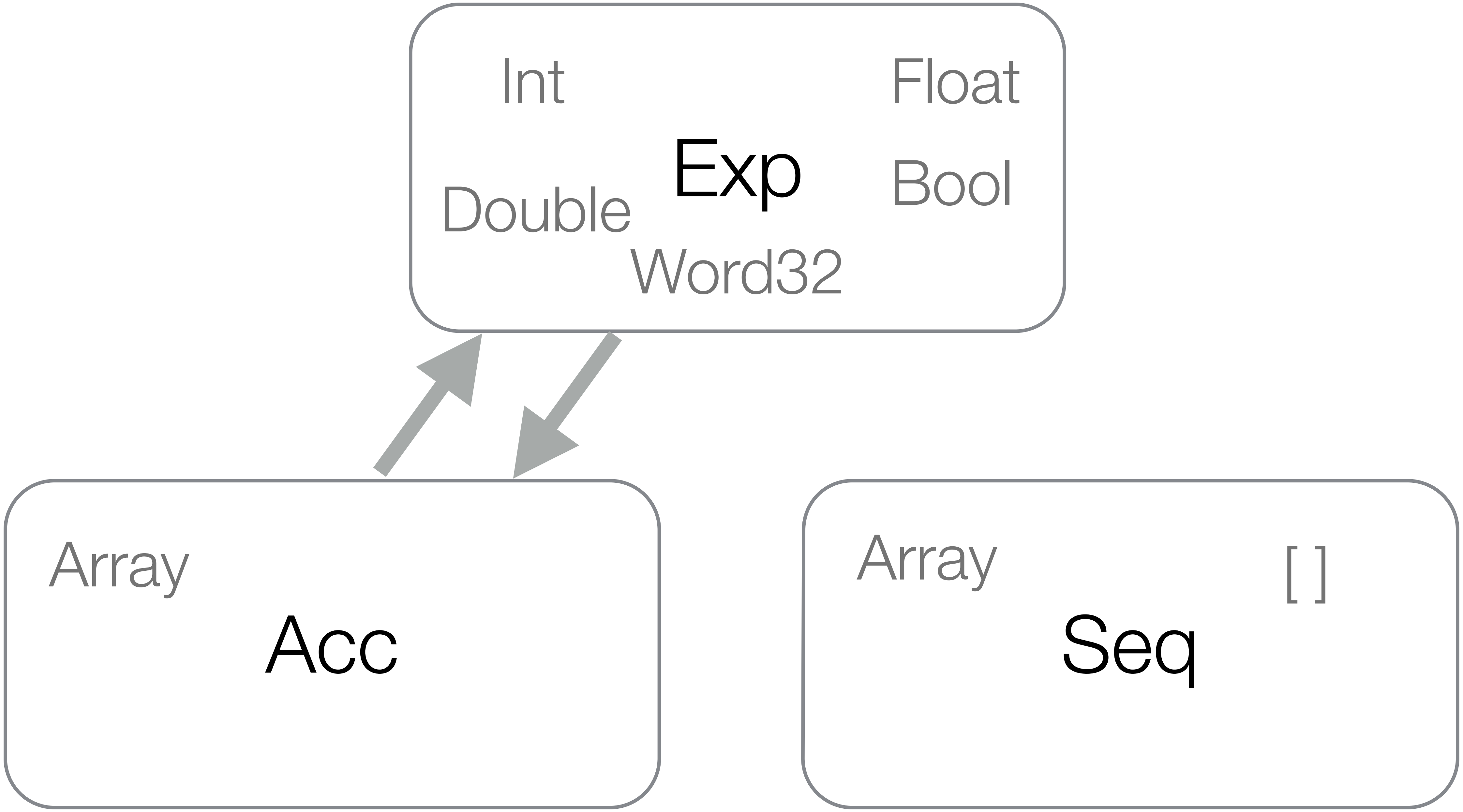
Bool

Word32

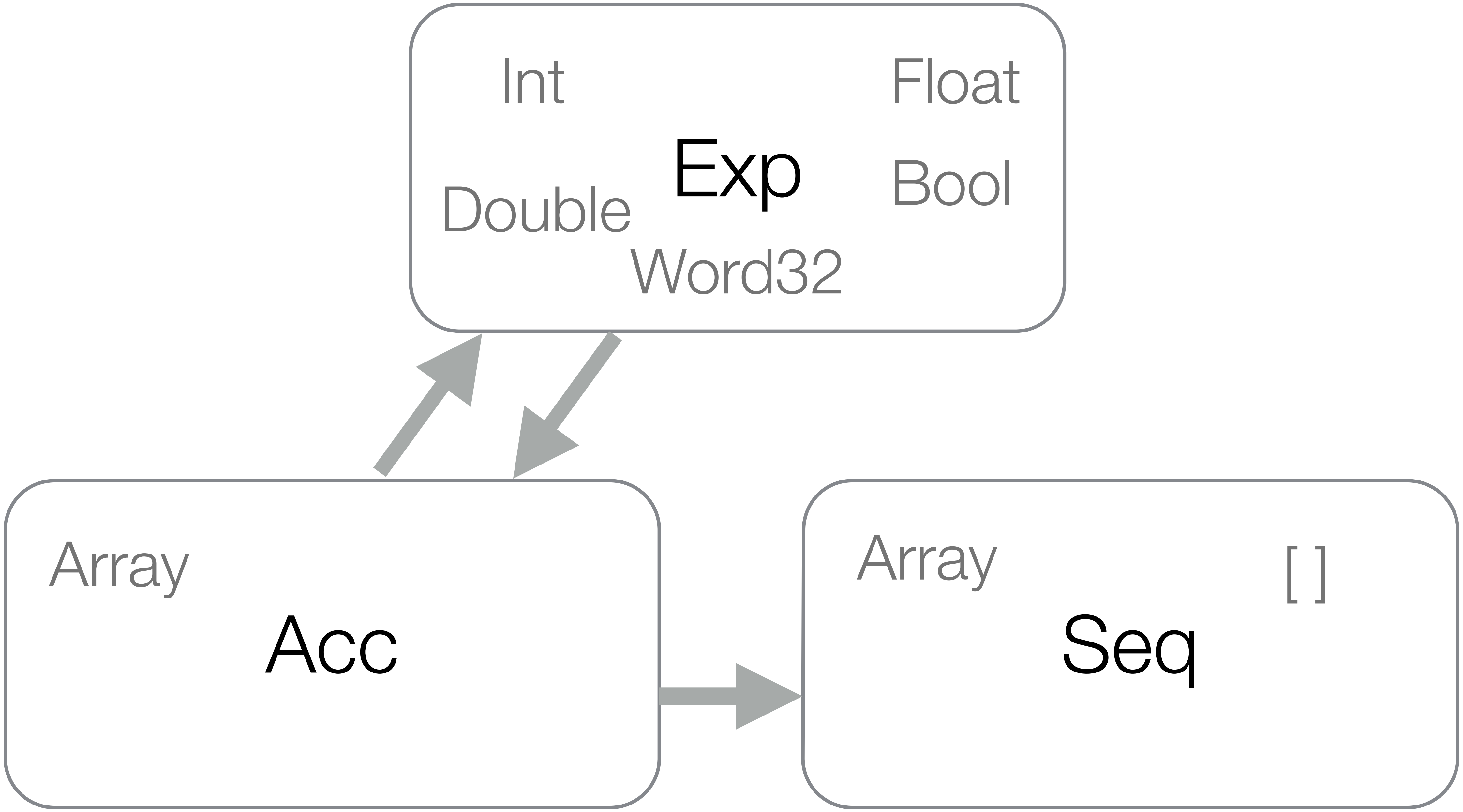
Array

Acc

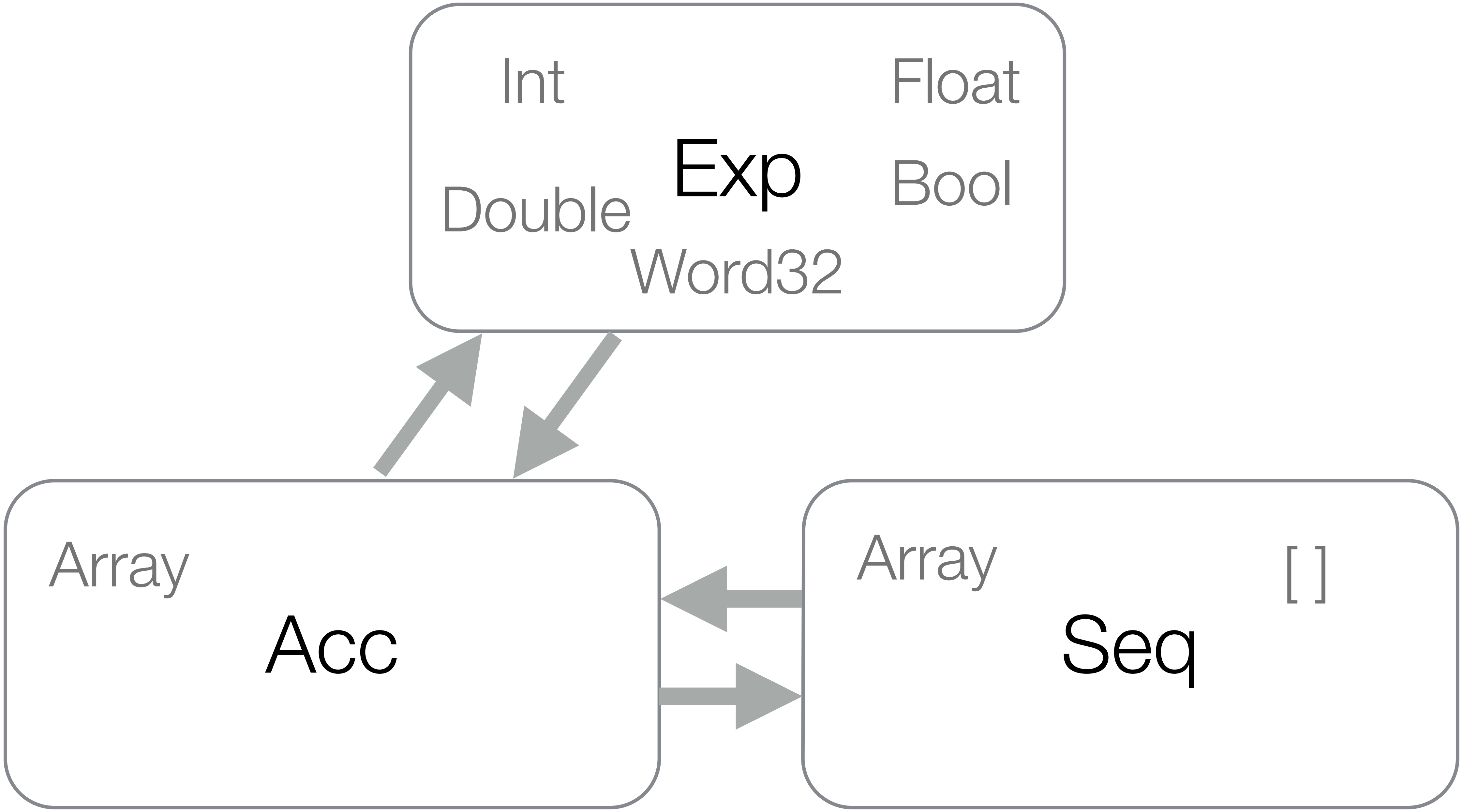
Sequences



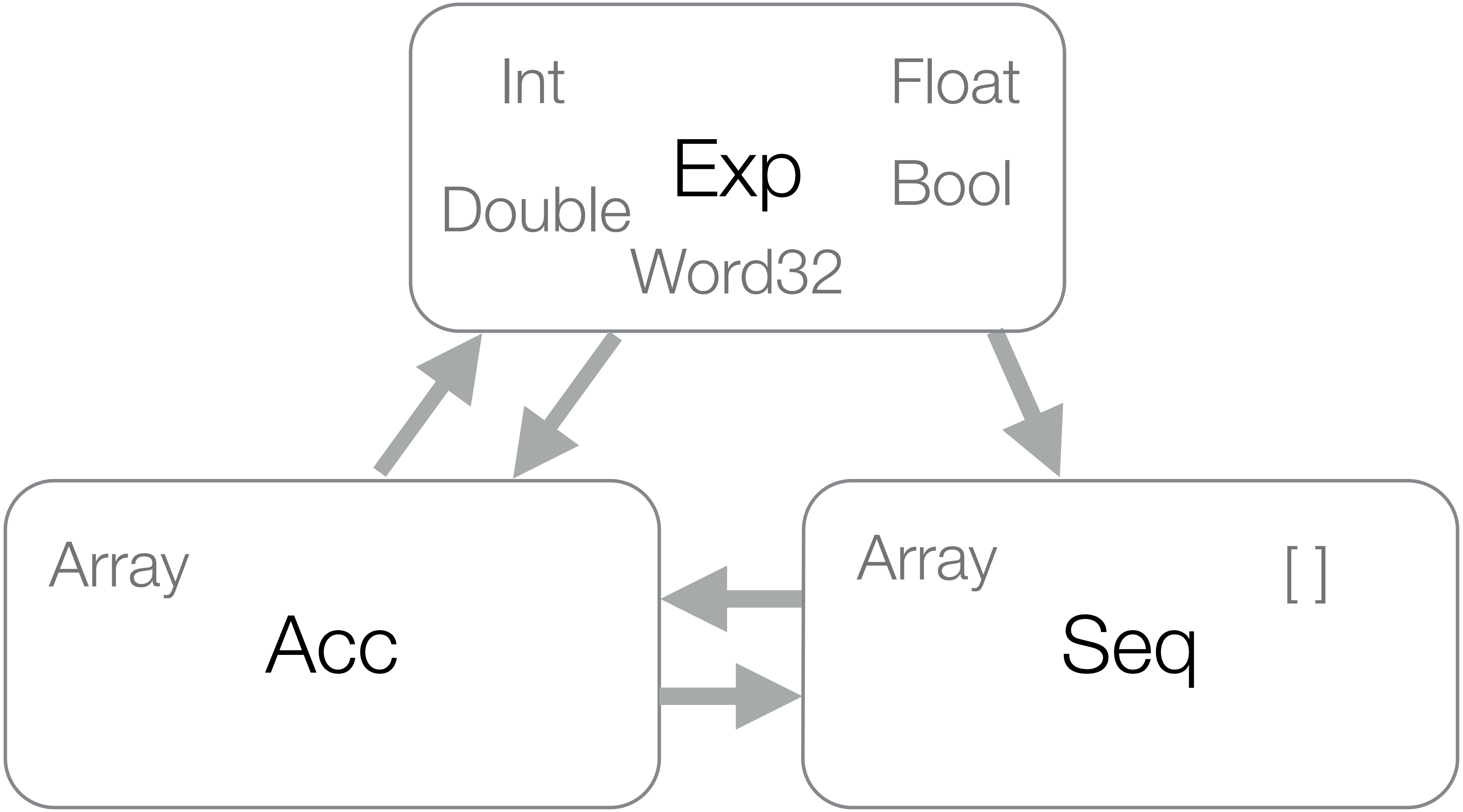
Sequences



Sequences



Sequences



Sequence operations

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Elements not always the same size

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Elements not always the same size

```
collect :: Arrays a => Seq a -> Acc a
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Elements not always the same size

```
collect :: Arrays a => Seq a -> Acc a
```

Sequence operations

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]

toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]

fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Elements not always the same size

```
collect :: Arrays a => Seq a -> Acc a
```

Array or tuple of arrays (not a sequence)

Sequence operations

```
mvm mat vec =
```

Sequence operations

```
mvm mat vec =


            $ toSeq mat
```

Sequence operations

```
mvm mat vec =


            $ mapSeq (dotp vec)
            $ toSeq mat
```

Sequence operations

```
mvm mat vec =

            $ fromSeq
            $ mapSeq (dotp vec)
            $ toSeq mat
```

Sequence operations

```
mvm mat vec = snd
            $ collect
            $ fromSeq
            $ mapSeq (dotp vec)
            $ toSeq mat
```

Execution and representation

Execution and representation

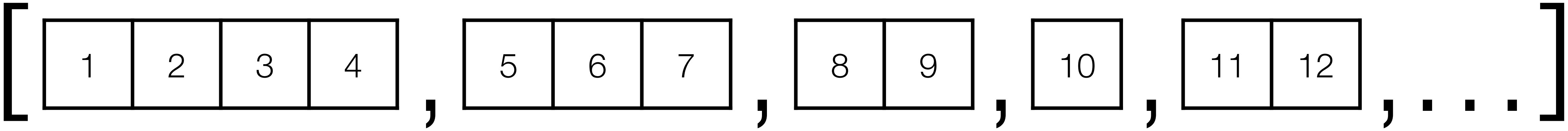
- Sequentially
  - The processing of each element has to expose enough parallelism

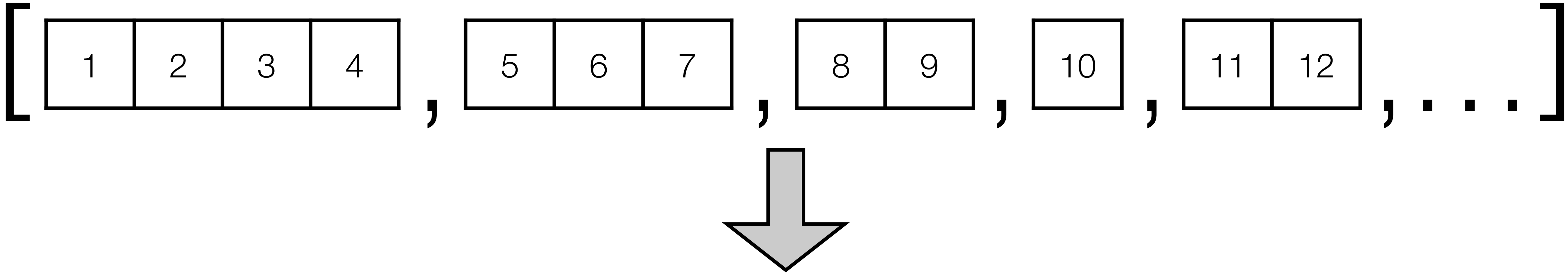
Execution and representation

- Sequentially
  - The processing of each element has to expose enough parallelism
- As one large vector
  - Use the lifting transform
  - Space problems

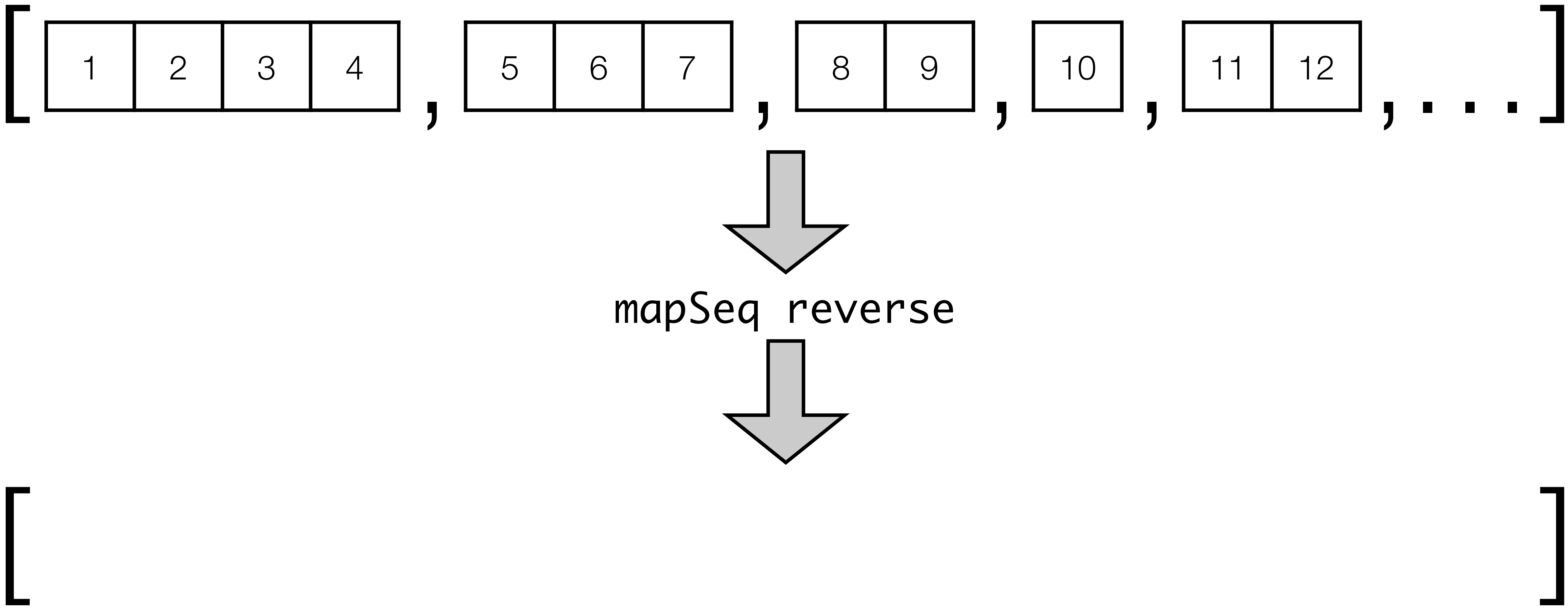
Execution and representation

- Sequentially
  - The processing of each element has to expose enough parallelism
- As one large vector
  - Use the lifting transform
  - Space problems
- Chunk-wise
  - Work on many elements in parallel

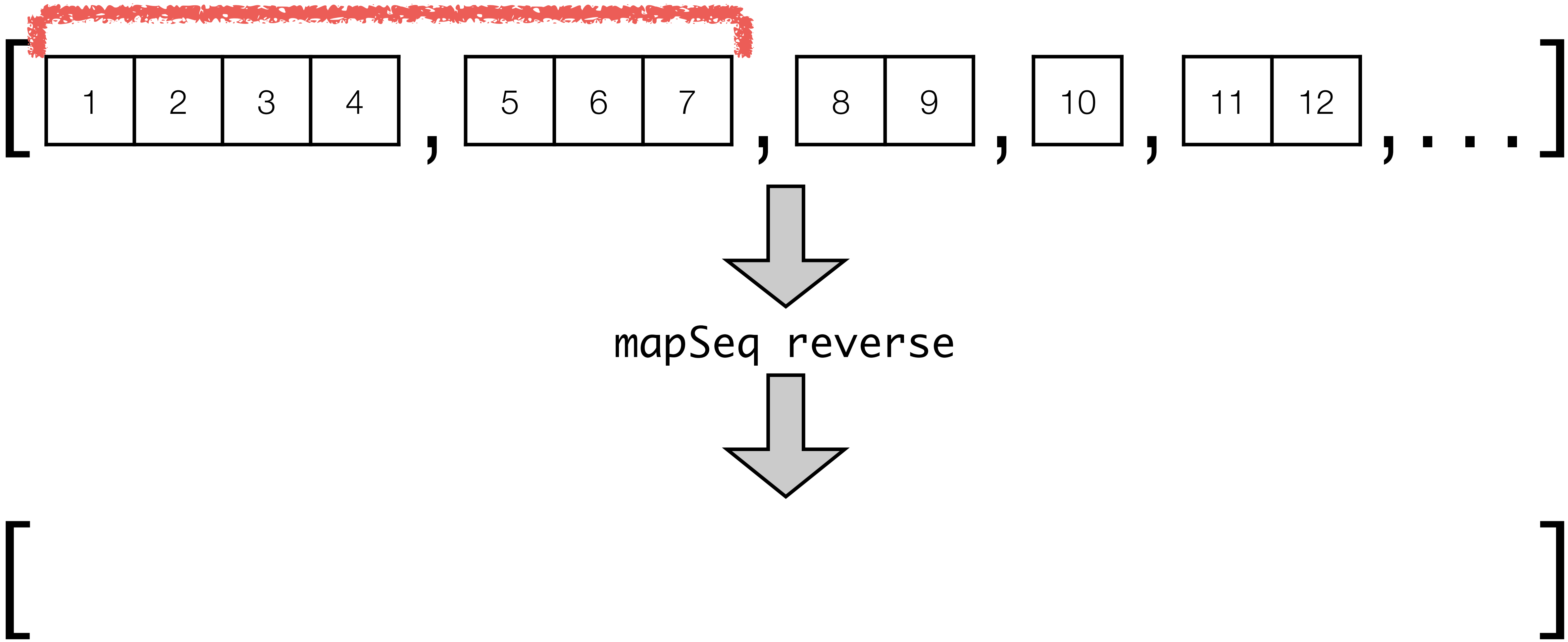

[
1
2
3
4
,
5
6
7
,
8
9
,
10
,
11
12
, . . .]



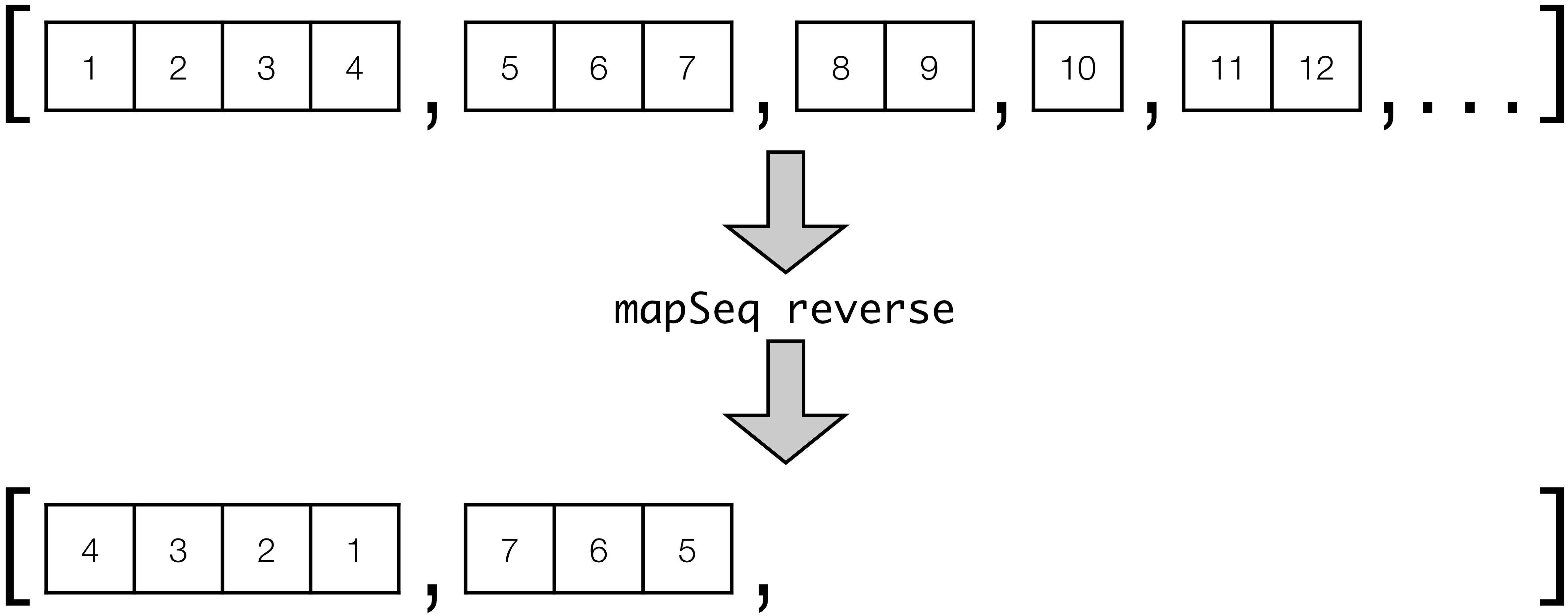
`mapSeq reverse`



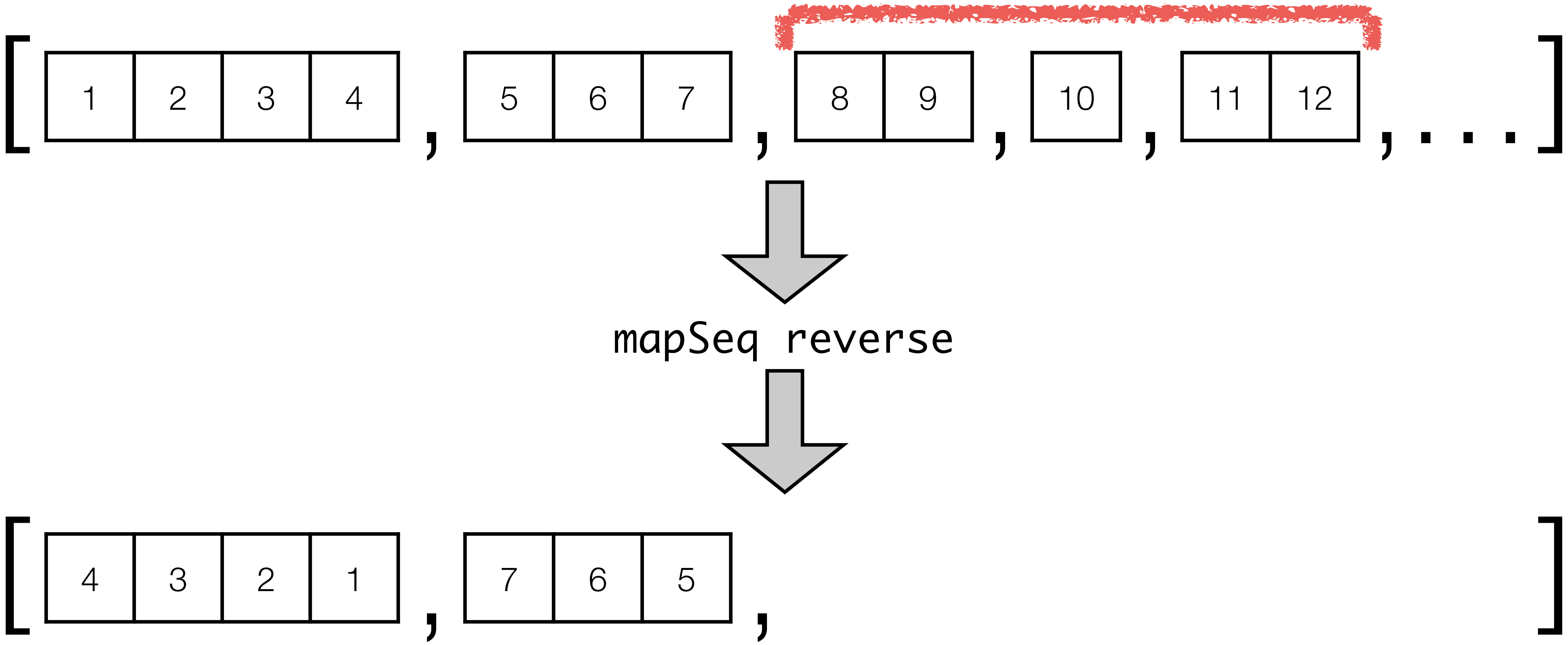
mapSeq reverse



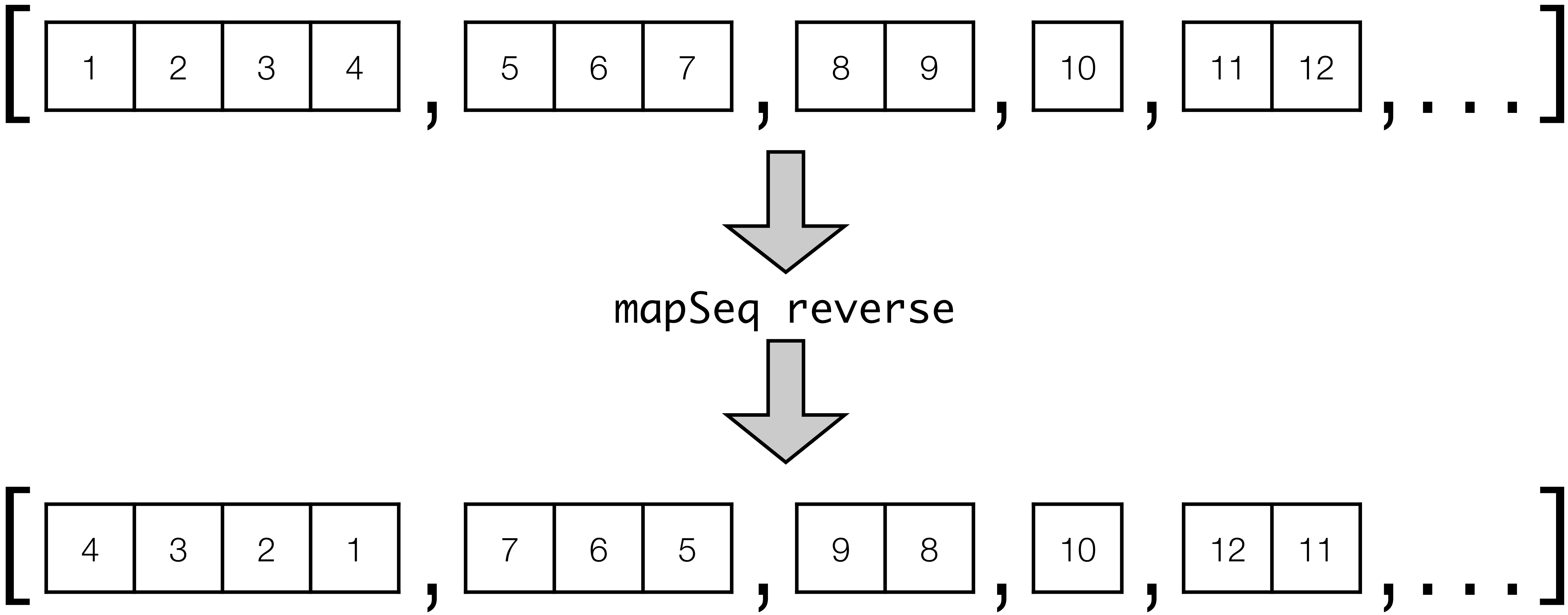
`mapSeq reverse`



mapSeq reverse



`mapSeq reverse`



`mapSeq reverse`

Reductions

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Has to be scalar!

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Has to be scalar!

- Better

```
foldSeqFlatten :: (Acc a -> Acc (Vector sh) -> Acc (Vector b) -> Acc a)
               -> Acc a
               -> Seq [Array sh b]
               -> Seq a
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Has to be scalar!

The accumulated value

- Better

```
foldSeqFlatten :: (Acc a -> Acc (Vector sh) -> Acc (Vector b) -> Acc a)
               -> Acc a
               -> Seq [Array sh b]
               -> Seq a
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Has to be scalar!

The accumulated value

- Better

```
foldSeqFlatten :: (Acc a -> Acc (Vector sh) -> Acc (Vector b) -> Acc a)
               -> Acc a
               -> Seq [Array sh b]
               -> Seq a
```

Reductions

- Ideal

```
foldSeq :: (Acc a -> Acc a -> Acc a)
        -> Acc a
        -> Seq [a]
        -> Seq (a)
```

- Basic

```
foldSeq :: (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Seq [Scalar a]
        -> Seq (Scalar a)
```

Has to be scalar!

The accumulated value

- Better

```
foldSeqFlatten :: (Acc a -> Acc (Vector sh) -> Acc (Vector b) -> Acc a)
               -> Acc a
               -> Seq [Array sh b]
               -> Seq a
```

A flattened chunk

Chunk size

Chunk size

- What’s the best size?

Chunk size

- What’s the best size?

- A lot of factors involved

Chunk size

- What’s the best size?

- A lot of factors involved
    - Number of GPU cores

Chunk size

- What's the best size?

- A lot of factors involved
  - Number of GPU cores
  - Available device Memory

Chunk size

- What's the best size?

- A lot of factors involved
  - Number of GPU cores
  - Available device Memory
  - The computation itself

Chunk size

- What’s the best size?

- A lot of factors involved
  - Number of GPU cores
  - Available device Memory
  - The computation itself
  - Space and time analysis of array computations

Chunk size

- What's the best size?

- A lot of factors involved
    - Number of GPU cores
    - Available device Memory
    - The computation itself
    - Space and time analysis of array computations

- Still ongoing work

Streaming

- Sequences allow for working with data sets larger than available GPU memory
  - A painful experience before
- Streaming operations

```
streamIn :: Arrays a => [a] -> Seq [a]

streamOut :: Arrays a => Seq [a] -> [a]
```

Lots more to do

- Regularity
  - Sequences where all elements are the same size
- Streaming from different sources
- Stateful operations
  - Scans
- Nested sequences

Questions?