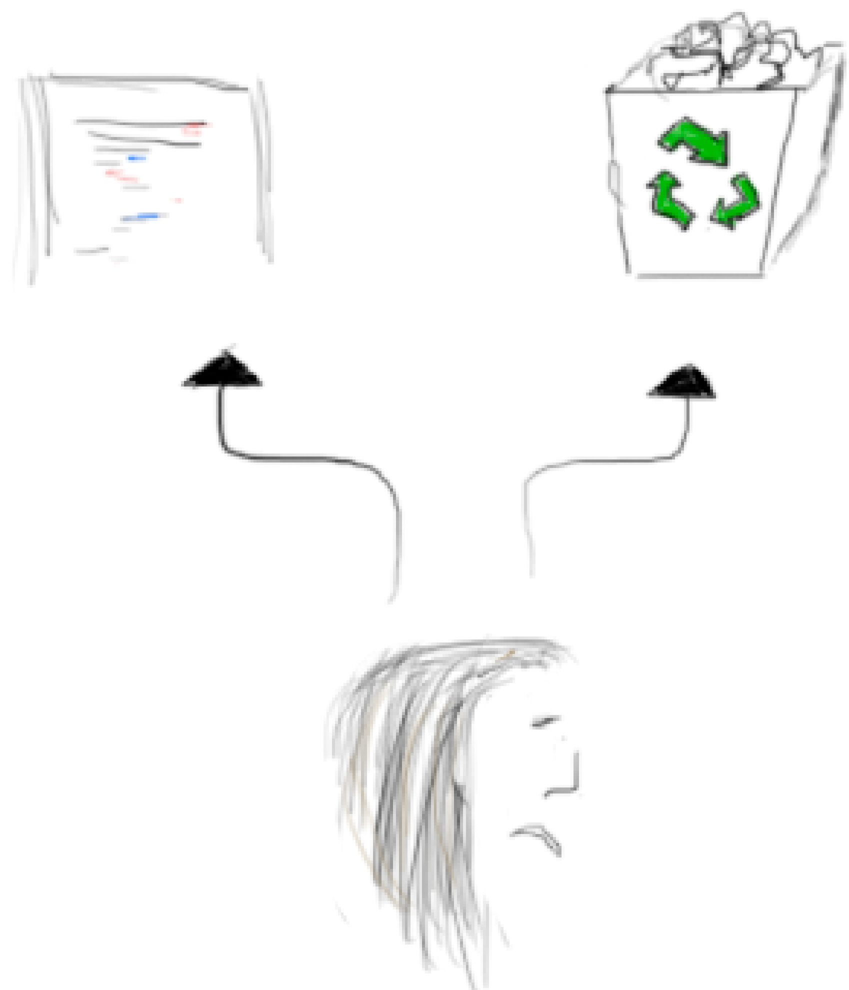
Rethinking Language Design for Parallelization

Dominic Orchard
Friday 20th August, 2010
Intel, Santa Clara, CA

Work from the Cambridge Programming Research Group with Max Bolingbroke and Alan Mycroft

Properties: lost



What to do if we want to utilise these (lost) properties

What to do if we want to utilise these (lost) properties

- Analysis + automatic transformation

What to do if we want to utilise these (lost) properties

- Analysis + automatic transformation
- Manual

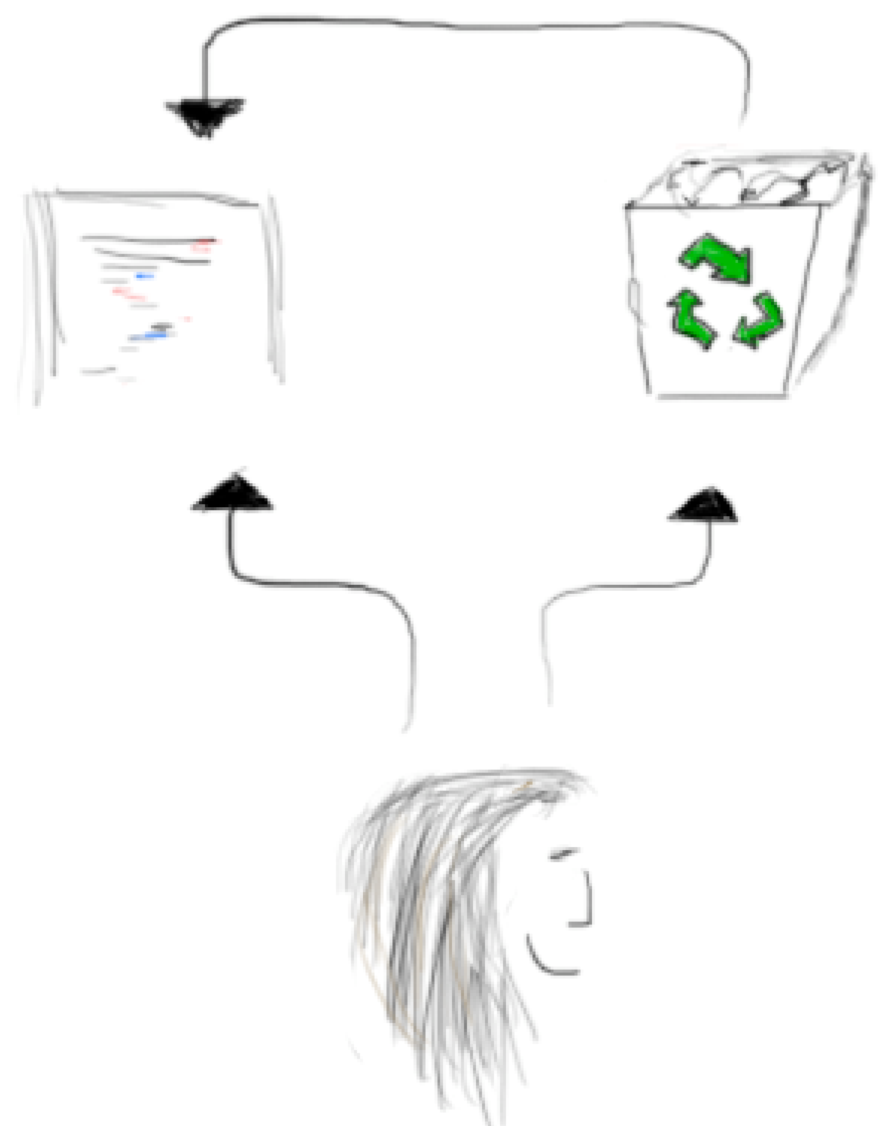
“Everything becomes clearer once you express it in the proper language.”

Greg Egan, *Schild’s Ladder*

Approach

Design a language to encode high-level program properties simply and directly

Properties: regained



A design approach

A design approach

- Pure

A design approach

- Pure
- Static typing

A design approach

- Pure
- Static typing
- Declarative, abstract (not ***how*** but ***what***)

A design approach

- Pure
- Static typing
- Declarative, abstract (not ***how*** but ***what***)
- Restricted => richer information encoding

Ypnos

- Ask about the name later!
- Haskell EDSL
- Data parallel programming with arrays
- User-instructed optimisation & parallelisation

Ypnos

- Ask about the name later!
- Haskell EDSL
- Data parallel programming with arrays
- User-instructed optimisation & parallelisation

Array type:

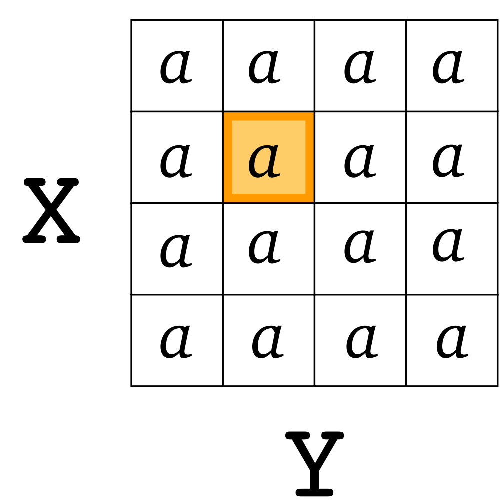
```
Grid d a
```

Ypnos

- Ask about the name later!
- Haskell EDSL
- Data parallel programming with arrays
- User-instructed optimisation & parallelisation

Array type:

`Grid d a`



`:: Grid (X*Y) a`

Computational pattern

example: Laplace

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] =
                    (A[i+1][j]+A[i-1][j]+
                     A[i][j-1]+A[i][j+1])/4.0;
     }
   }
   swap(Atemp, A);
}
```

Computational pattern

example: Laplace

```
while(condition) {
   for (int i=0; i<N; i++) {
      for (int j=0; j<M; j++) {
         Atemp[i][j] =
                    (A[i+1][j]+A[i-1][j]+
                     A[i][j-1]+A[i][j+1])/4.0;
      }
   }
   swap(Atemp, A);
}
```

- Called *mesh codes*, *stencil codes*, *kernels*, *structured grids*, *convolutions*, *gather operations*, *pixel shaders*

Computational pattern

example: Laplace

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] = f(A, i, j);
     }
   }
   swap(Atemp, A);
}

f(A, i, j) {
      ...
}
```

Computational pattern (2)

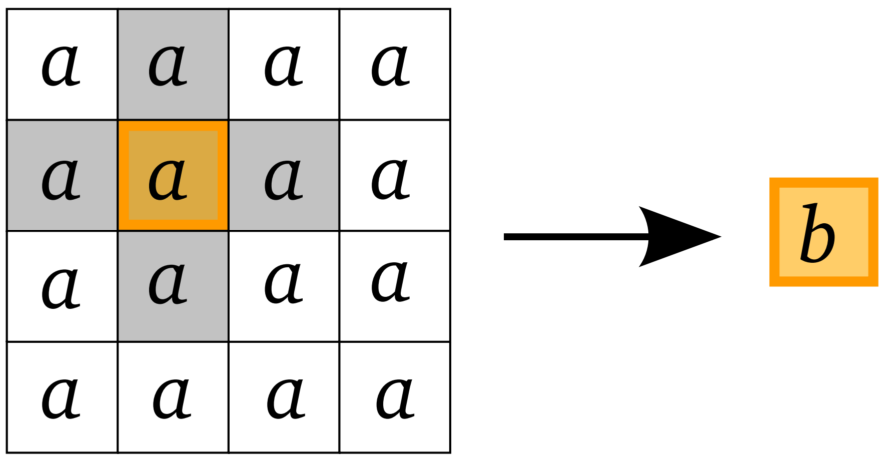
Kernel or *stencil* function:

$$\texttt{f ::}\quad \texttt{Grid D A} \longrightarrow \texttt{B}$$

Computational pattern (2)

Kernel or *stencil* function:

$$\texttt{f :: Grid D A} \rightarrow \texttt{B}$$



Computational pattern (3)

Apply the *stencil* function:

$$\texttt{run} :: (\texttt{Grid d a} \rightarrow \texttt{b}) \rightarrow (\texttt{Grid d a} \rightarrow \texttt{Grid d b})$$

Computational pattern (3)

Apply the *stencil* function:

$$\texttt{run} :: (\texttt{Grid d a} \rightarrow \texttt{b}) \rightarrow (\texttt{Grid d a} \rightarrow \texttt{Grid d b})$$

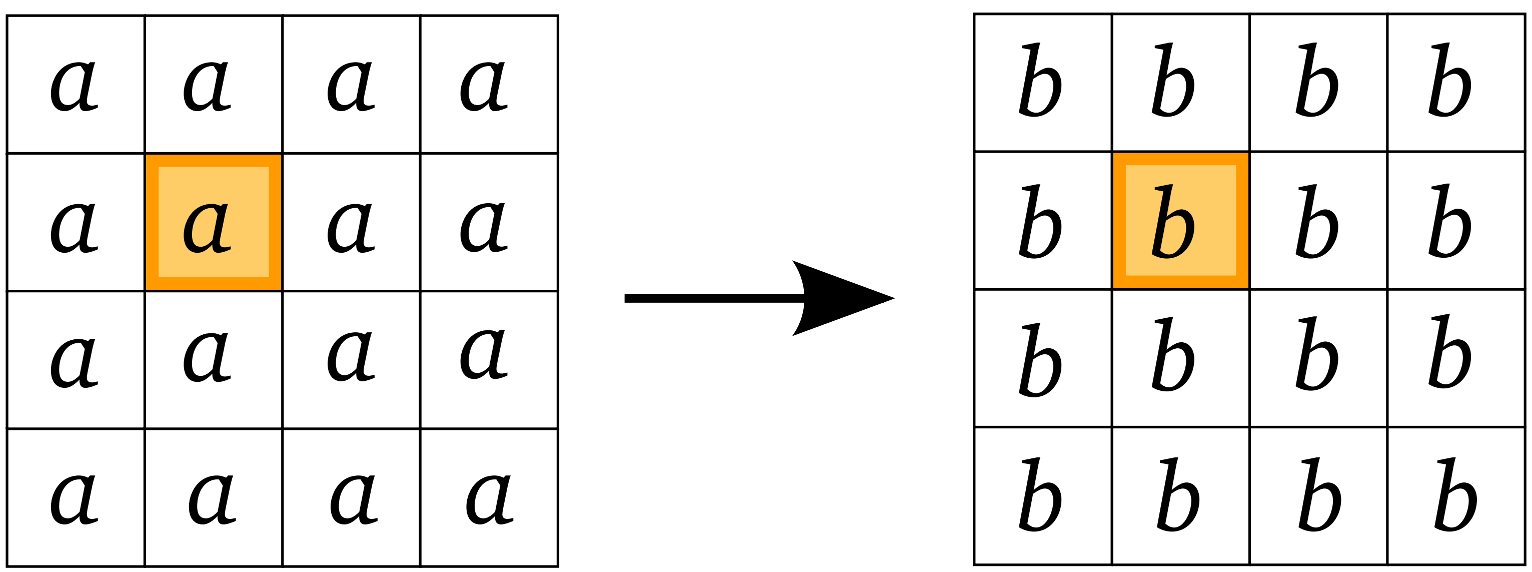
parameter stencil function

Computational pattern (3)

Apply the *stencil* function:

$$\texttt{run} :: (\texttt{Grid d a} \rightarrow \texttt{b}) \rightarrow (\texttt{Grid d a} \rightarrow \texttt{Grid d b})$$

parameter stencil function



Array access

Array access

- Usually general indexing operation:

Array access

- Usually general indexing operation:
  - `a[i][j]`, `a!!(i, j)`, `get(a, i, j)` etc.

Array access

- Usually general indexing operation:
  - `a[i][j]`, `a!!(i, j)`, `get(a, i, j)` etc.
- Allows random read/write

Array access

- Usually general indexing operation:
    - `a[i][j]`, `a!!(i, j)`, `get(a, i, j)` etc.
- Allows random read/write
- Ypnos, *"Throw indexing to the dogs, I'll none of it!"*

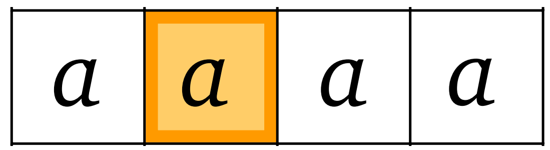
Grid patterns

$$\texttt{f :: Grid X a} \rightarrow \texttt{b}$$

$$\texttt{f} \,|\texttt{l @c r}|\, = \texttt{...}$$

Grid patterns

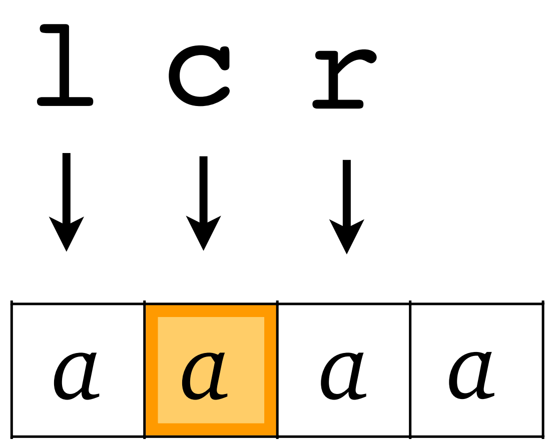
```
f ::  Grid X a → b
f |l @c r| = ...
```



Grid patterns

$$\texttt{f :: Grid X a} \rightarrow \texttt{b}$$

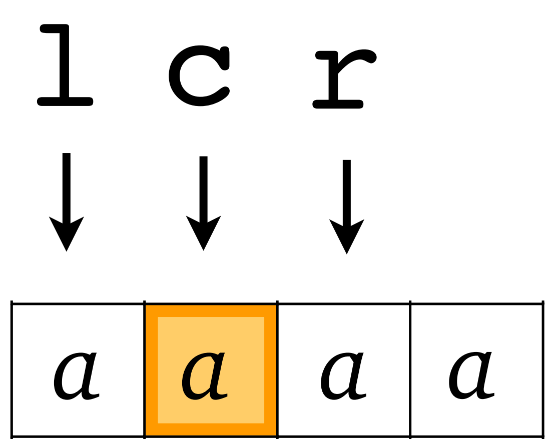
$$\texttt{f}\ |\texttt{l @c r}| = \ldots$$



Grid patterns

$$f :: \ \texttt{Grid X a} \rightarrow \texttt{b}$$

$$f \, | \, l \ @c \ r \, | = \ldots$$




```
c = a[i]
l = a[i-1]
r = a[i+1]
```

Grid patterns (continued...)

```
f ::  Grid (X × Y) a → b
f X:| l @c r | = ...
```

Grid patterns (continued...)

```
f :: Grid (X × Y) a → b
f X:| l @c r | = ...
```

slices in X dimension `l,c,r :: Grid Y a`

Grid patterns (continued...)

```
f :: Grid (X × Y) a → b
f X:| l @c r | = ...
```

slices in X dimension `l,c,r :: Grid Y a`

```
f :: Grid (X × Y) a → b
f X:| l @c r | = ..g l..g c..g r
g :: Grid Y a → b
g Y:| t @c b | = ...
```

Grid patterns (continued...)

Can nest grid patterns:

Grid patterns (continued...)

Can nest grid patterns:

```
f ::  Grid (X × Y) a → b
f || lt lc lb || ct @cc cb || rt rc rb || = ...
```

Grid patterns (continued...)

Can nest grid patterns:

```
f ::  Grid (X × Y) a → b
f || lt lc lb || ct @cc cb || rt rc rb || = ...
```

or 2D pattern match sugar:

Grid patterns (continued...)

Can nest grid patterns:

$$f :: \texttt{Grid}\ (X \times Y)\ a \rightarrow b$$

```
f || lt lc lb || ct @cc cb || rt rc rb || = ...
```

or 2D pattern match sugar:

$$f :: \texttt{Grid}\ (X \times Y)\ a \rightarrow b$$

```
f (X × Y):| tl tc tr  |
          | cl @cc cr | = ...
          | bl bc br  |
```

Grid patterns (continued...)

Grid patterns (continued...)

- Static, constant

Grid patterns (continued...)

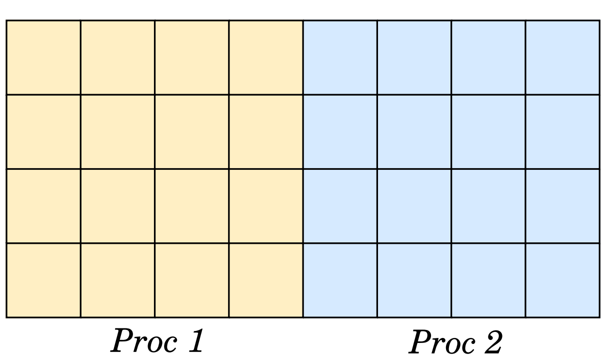
- Static, constant
- Guaranteed access information, no analysis

Grid patterns (continued...)

- Static, constant
- Guaranteed access information, no analysis
- Facilitates vectorisation, distributed-memory implementations

Grid patterns (continued...)

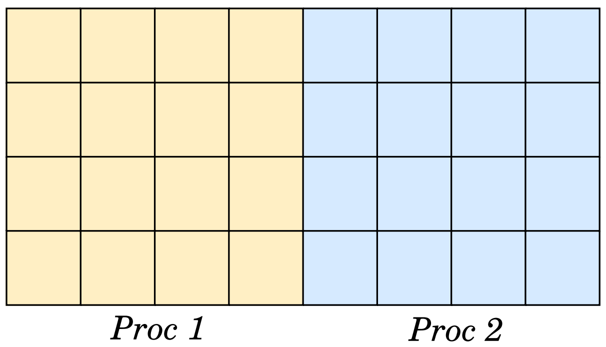
- Static, constant
- Guaranteed access information, no analysis
- Facilitates vectorisation, distributed-memory implementations



Shared

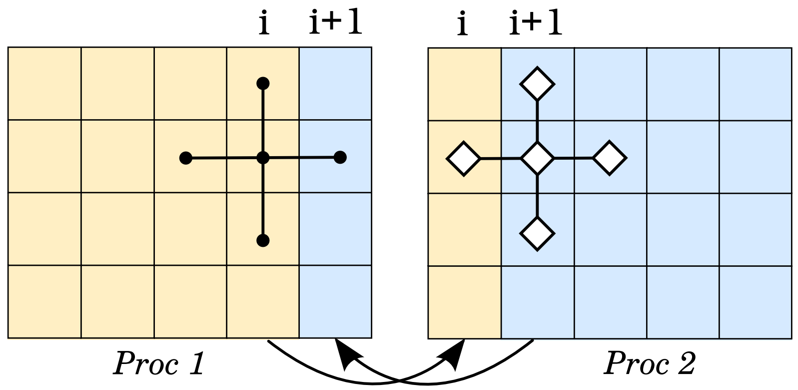
Grid patterns (continued...)

- Static, constant
- Guaranteed access information, no analysis
- Facilitates vectorisation, distributed-memory implementations



Shared

Distributed



Example: Laplace

```
laplace :: Grid (X*Y) Double -> Double
laplace (X*Y):| _ a  _ | = (a+b+c+d)*0.25
              | b @_ c |
              | _ d  _ |

g = grid <X = 10, Y = 10> data
g' = run laplace (defaults 0.0 g)
```

Parallelisation

Parallelisation

- No side effects

Parallelisation

- No side effects
- No loop carried dependencies

Parallelisation

- No side effects
- No loop carried dependencies
- Stencil function = single, independent write

Parallelisation

- No side effects
- No loop carried dependencies
- Stencil function = single, independent write
- Can only read/write with `run`

Parallelisation

- No side effects
- No loop carried dependencies
- Stencil function = single, independent write
- Can only read/write with `run`
- Parameterisable backend for `runPar`

Reductions

Reductions

- Parallel reduction structure (tree reduce)

Reductions

- Parallel reduction structure (tree reduce)

$$
\begin{array}{rlll}
\textsf{mkReducer} :: & (a \to b \to b) & \textit{partial reduce} \\
\to & (b \to b \to b) & \textit{combine partials} \\
\to & b & \textit{initial partial result} \\
\to & (b \to c) & \textit{final conversion} \\
\to & \textsf{Reducer}\ a\ c &
\end{array}
$$

Reductions

- Parallel reduction structure (tree reduce)

$$
\begin{array}{lll}
\text{mkReducer} :: & (a \to b \to b) & \textit{partial reduce} \\
\to & (b \to b \to b) & \textit{combine partials} \\
\to & b & \textit{initial partial result} \\
\to & (b \to c) & \textit{final conversion} \\
\to & \text{Reducer}\ a\ c &
\end{array}
$$

$$\texttt{reduce} :: \text{Reducer}\ a\ b \to \text{Grid}\ D\ a \to b$$

Computational pattern

example: Laplace

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] =
                      (A[i+1][j]+A[i-1][j]+
                       A[i][j-1]+A[i][j+1])/4.0;
     }
   }
   swap(Atemp, A);
}
```

Computational pattern

example: Laplace

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] =
                   (A[i+1][j]+A[i-1][j]+
                    A[i][j-1]+A[i][j+1])/4.0;
     }
   }
   swap(Atemp, A);
}
```

- Many iterations until convergence

Iterating computations

Iterating computations

```
iterate :: (Grid d a -> a) -> Grid d a ->
             Reducer a Bool -> Grid d a
iterate stencil g r =
```

Iterating computations

```
iterate :: (Grid d a -> a) -> Grid d a ->
              Reducer a Bool -> Grid d a
iterate stencil g r =

            if (reduce g r) then
               g
            else
               let g' = (run stencil g)
               in iterate stencil g' r
```

Iterating computations (2)

`iterate` :: $(\text{Grid}\ d\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$

Iterating computations (2)

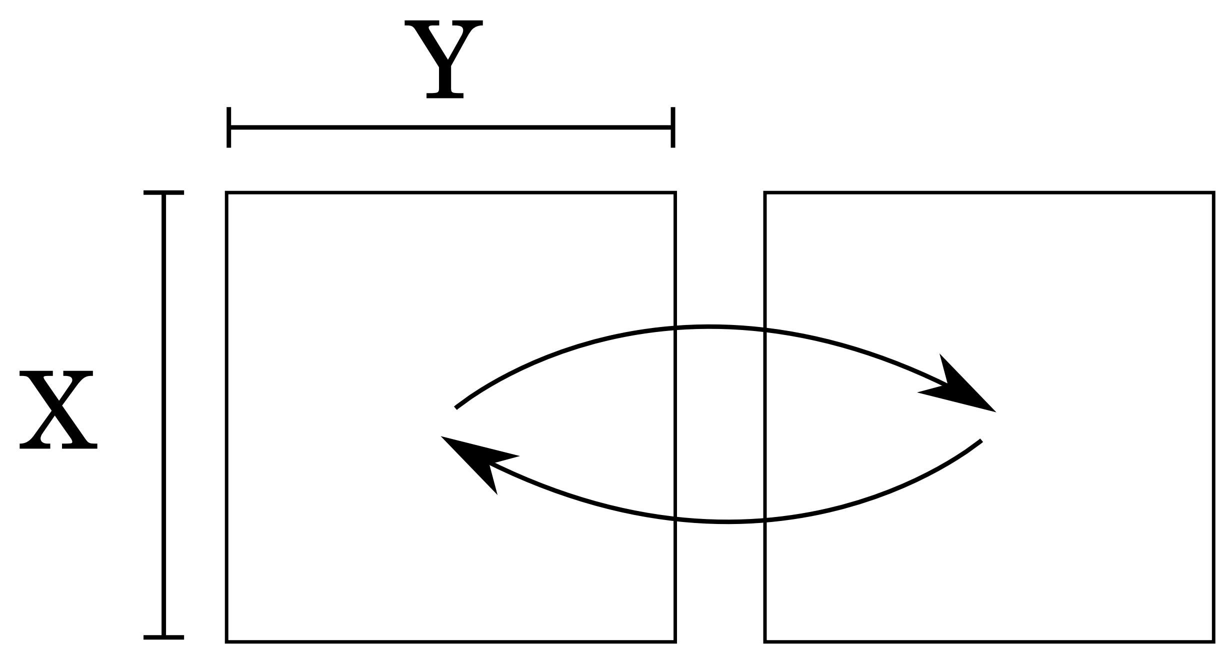
`iterate` :: $(\text{Grid}\ d\ a \to a) \to \text{Reducer}\ a\ Bool \to \text{Grid}\ d\ a \to \text{Grid}\ d\ a$

- Locally reuse allocations

Iterating computations (2)

`iterate` :: $(\text{Grid}\ d\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$

- Locally reuse allocations



When aliasing of old grids is needed...

When aliasing of old grids is needed...

- Use grid patterns in *time!*

When aliasing of old grids is needed...

- Use grid patterns in *time!*

$$\texttt{iterate} :: (\mathsf{Grid}\ d\ a \rightarrow a) \rightarrow \mathsf{Reducer}\ a\ Bool \rightarrow \mathsf{Grid}\ d\ a \rightarrow \mathsf{Grid}\ d\ a$$

When aliasing of old grids is needed...

- Use grid patterns in *time!*

$$\texttt{iterate} :: (\text{Grid}\ d\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$$

$$\texttt{iterateT} :: (\text{Grid}\ (T \times d)\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$$

When aliasing of old grids is needed...

- Use grid patterns in *time!*

$$\texttt{iterate} :: \ (\text{Grid}\ d\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$$

$$\texttt{iterateT} :: \ (\text{Grid}\ (T \times d)\ a \rightarrow a) \rightarrow \text{Reducer}\ a\ Bool \rightarrow \text{Grid}\ d\ a \rightarrow \text{Grid}\ d\ a$$

e.g. $T : |\ \ g''\ \ g'\ \ @_{\_}\ \ |$

When aliasing of old grids is needed...

- Use grid patterns in *time!*

$$\texttt{iterate} :: (\text{Grid } d\ a \rightarrow a) \rightarrow \text{Reducer } a\ Bool \rightarrow \text{Grid } d\ a \rightarrow \text{Grid } d\ a$$

$$\texttt{iterateT} :: (\text{Grid } (T \times d)\ a \rightarrow a) \rightarrow \text{Reducer } a\ Bool \rightarrow \text{Grid } d\ a \rightarrow \text{Grid } d\ a$$

e.g. $T : |\ g''\ \ g'\ \ @_{\_}\ |$

Creates three intermediate allocations:

When aliasing of old grids is needed...

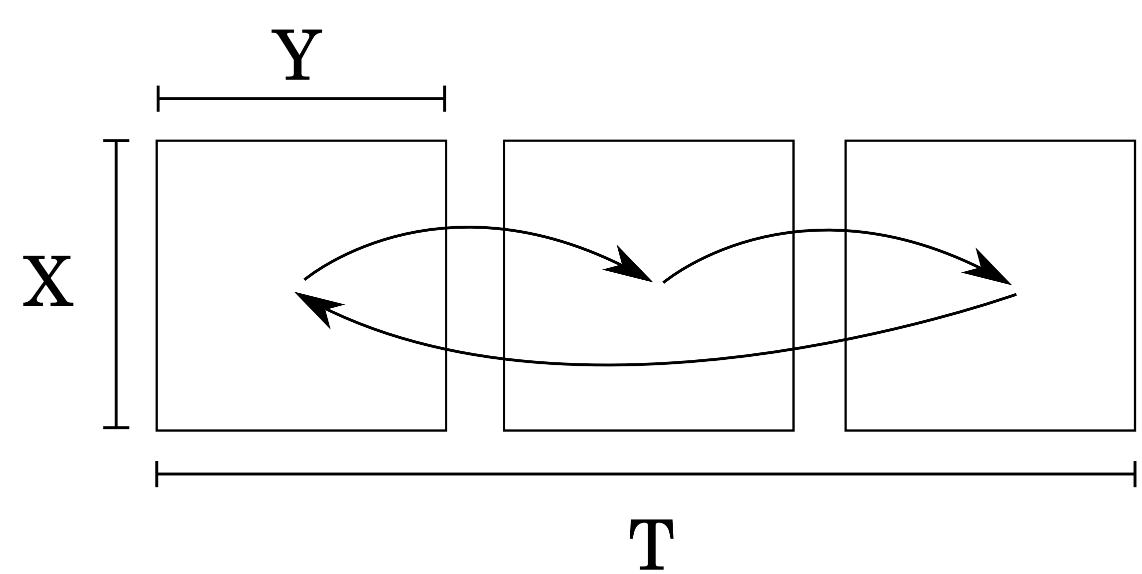
- Use grid patterns in *time!*

$\texttt{iterate} :: \; (\mathsf{Grid}\; d\; a \rightarrow a) \rightarrow \mathsf{Reducer}\; a\; Bool \rightarrow \mathsf{Grid}\; d\; a \rightarrow \mathsf{Grid}\; d\; a$

$\texttt{iterateT} :: \; (\mathsf{Grid}\; (T \times d)\; a \rightarrow a) \rightarrow \mathsf{Reducer}\; a\; Bool \rightarrow \mathsf{Grid}\; d\; a \rightarrow \mathsf{Grid}\; d\; a$

e.g. $T : | \;\; g'' \;\; g' \;\; @_{\_} \;\; |$

Creates three intermediate allocations:



Optimisation & Parallelisation

Optimisation & Parallelisation

`iteratePar` `iterateTPar`

Optimisation & Parallelisation

`iteratePar` `iterateTPar`

- Parallel versions of `iterate` & `iterateT`

Optimisation & Parallelisation

`iteratePar` `iterateTPar`

- Parallel versions of `iterate` & `iterateT`
- Locally use optimized destructive update

Optimisation & Parallelisation

`iteratePar`   `iterateTPar`

- Parallel versions of `iterate` & `iterateT`
- Locally use optimized destructive update
- Compiler configurations for parameters such as tile size

Summary

$$\texttt{run} :: (\text{Grid}\ d\ a \to b)\ \to\ \text{Grid}\ d\ a\ \to\ \text{Grid}\ d\ b$$

$$\texttt{iterate} :: (\text{Grid}\ d\ a \to a) \to \text{Reducer}\ a\ Bool \to \text{Grid}\ d\ a \to \text{Grid}\ d\ a$$

$$\texttt{iterateT} :: (\text{Grid}\ (T \times d)\ a \to a) \to \text{Reducer}\ a\ Bool \to \text{Grid}\ d\ a \to \text{Grid}\ d\ a$$

and

`runPar, iteratePar, iterateTPar`

Conclusions

Conclusions

- Restricted DSL provides strong information

Conclusions

- Restricted DSL provides strong information
- Tractable cost model for programmer

Conclusions

- Restricted DSL provides strong information
- Tractable cost model for programmer
- **Guaranteed** parallelisation and optimisations

Conclusions

- Restricted DSL provides strong information
- Tractable cost model for programmer
- **Guaranteed** parallelisation and optimisations
- Easy to write, rewrite, change strategy

Conclusions

- Restricted DSL provides strong information
- Tractable cost model for programmer
- **Guaranteed** parallelisation and optimisations
- Easy to write, rewrite, change strategy
- Hardware agnostic. Currently shared & distributed memory backends

Paper

- Orchard D, Bolingbroke M, Mycroft A "*Ypnos: Declarative Parallel Structured Grid Programming*" In proceedings of ACM SIGPLAN DAMP 2010, January, Madrid

Conway's Game of Life

```
life (X*Y):| a b  c | = let local = (a+b+c+d+e+f+g+h+i)
           | d @e f |   in  if (e==1) then
           | g h  i |         if (local<2 || local>3)
                               then 0 else 1
                             else
                               if (local==3)
                               then 1 else 0
-- Create environment
initalState = grid <X=10, Y=10> randomConfiguration

untilMostlyDead = Reducer (+) (+) 0.0 (\x -> (x<10))
stopCondition = (untilMostlyDead `orReducer` (ntimes 100))

intialState' = defaults 0.0 initialState
finalState = iterate life stopCondition initialState'
```