

# *Ypnos*: Declarative, Parallel Structured Grid Programming

Dominic Orchard, Max Bolingbroke, Alan Mycroft  
Computer Laboratory, University of Cambridge, UK

DAMP'10

January 19, Madrid, Spain

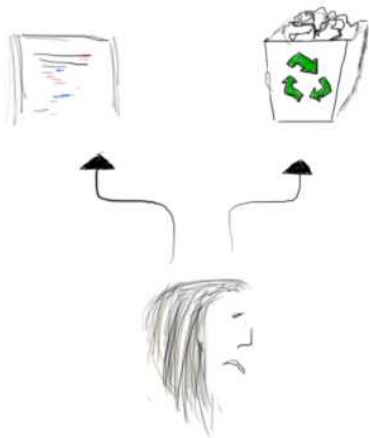
# Ypnos

- Functional, declarative DSL (embedded in Haskell)
- Domain: Structured grids (stencils) in N-dimensions
- Restricted operations with novel syntactic form
- Guarantees optimisation and parallelisation
- Prevents some broken programs
- Permits different backends

## We may have lots of ideas about our programs...

- *“This function has no side effects.”*
- *“This variable is never zero.”*
- $\mathbf{A} \times \mathbf{A}^{-1} = \mathbf{I}$
- *“These variables are not aliases for the same value.”*
- *“These two operations can be run in parallel.”*
- ...

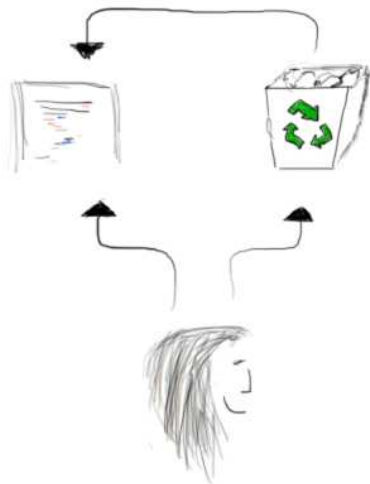
... but ideas and information are lost.



## So... how can we exploit higher-level program properties?

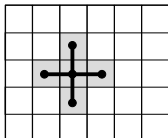
- Manually
  - Hand parallelisation or optimisation
  - Libraries and frameworks
- Automatically
  - Analysis: Get lost information back
  - Transformation

## DSLs to the rescue

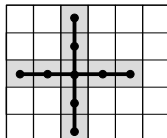


# A Specific Application Domain: Structured Grids

- Arrays representing discretised environments
- Stencil function: compute a new value for each array element from neighbours



(a) 5-point stencil



(b) 9-point stencil

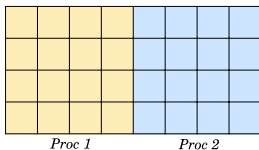
- Common in scientific computing, graphics, games, etc.

## C Example

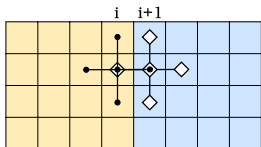
```
while(condition) {  
    for (int i=0; i<N; i++) {  
        for (int j=0; j<M; j++) {  
            Atemp[i][j] = (A[i+1][j]+A[i-1][j]+  
                A[i][j-1]+A[i][j+1])/4.0;  
        }  
    }  
    swap(Atemp, A);  
}
```

# Highly data parallel

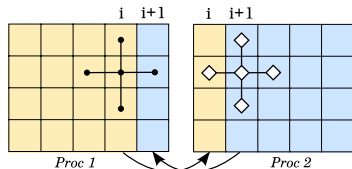
- Shared memory:



- Distributed memory:



(c) Before decomposition



(d) After decomposition

## Current Approaches

- Manual: Maybe C/FORTRAN parallelised manually with MPI (Message Passing Interface)
- Automatic: HPF, Chapel, SISAL, Polyhedral analyses
- Frameworks: OpenCL, CUDA, Cg, OpenMP

Ypnos

## Grid data access

$gridMap :: (a \rightarrow b) \rightarrow Grid\ a \rightarrow Grid\ b$

$f :: Double \rightarrow Double$

$f\ x = \dots$

$grid' = arrayMap\ f\ grid$

## Grid data access (2)

`run :: (Grid a → b) → Grid a → Grid b`

*f :: Grid Double → Double*

## Grid data access (2)

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

$f :: \text{Grid Double} \rightarrow \text{Double}$

$f \mid l @c r \mid = \dots$

## Grid data access (2)

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

$f :: \text{Grid Double} \rightarrow \text{Double}$

$f \mid l @c r \mid = \dots$

$\text{grid}' = \text{run } f \text{ grid}$

Equivalent to:

$l = A[i-1];$

$c = A[i];$

$r = A[i+1];$

## Grid data access (3)

$\mathbf{run} :: (\text{Grid } D \ a \rightarrow b) \rightarrow \text{Grid } D \ a \rightarrow \text{Grid } D \ b$

$f :: \text{Grid } X \ Double \rightarrow Double$

$f \ X : | l @c r | = \dots$

$grid' = \mathbf{run} \ f \ grid$

## Grid data access (4)

$\text{run} :: (\text{Grid } D \ a \rightarrow b) \rightarrow \text{Grid } D \ a \rightarrow \text{Grid } D \ b$

$f :: \text{Grid } (X \times Y) \ \text{Double} \rightarrow \text{Double}$

$f \ X : | l @c r | = \dots$

$\text{grid}' = \text{run } f \ \text{grid}$

$l, c, r : \text{Grid } Y \ \text{Double}$

## Grid data access (4)

$\text{run} :: (\text{Grid } D a \rightarrow b) \rightarrow \text{Grid } D a \rightarrow \text{Grid } D b$

$f :: \text{Grid } (X \times Y) \text{ Double} \rightarrow \text{Double}$

$f X : | l @c r | = \dots g l \dots g c \dots g r \dots$

$g Y : | l @c r | = \dots$

$grid' = \text{run } f \text{ grid}$

## Grid data access (5)

`run` :: (Grid  $D$   $a \rightarrow b$ )  $\rightarrow$  Grid  $D$   $a \rightarrow$  Grid  $D$   $b$

$f$  :: Grid  $(X \times Y)$  Double  $\rightarrow$  Double

$f$   $(X \times Y)$  :  $\left| \begin{array}{ccc} - & t & - \\ l & @c & r \\ - & b & - \end{array} \right| = \dots$

$grid' = \text{run } f \text{ grid}$

## Applying Stencil Functions

$\text{run} :: (\text{Grid } D \ a \rightarrow b) \rightarrow \text{Grid } D \ a \rightarrow \text{Grid } D \ b$

- Applies a stencil function at every point in a grid.
- Collects results into a new grid.

<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>

$\longrightarrow$  *b*

$f :: \text{Grid } a \rightarrow b$

<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>

$\longrightarrow$

<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>

$\text{run } f :: \text{Grid } a \rightarrow \text{Grid } b$

## Boundaries

		?		
?	a	a	a	a
	a	a	a	a
	a	a	a	a
	a	a	a	a

- Via grid patterns we know maximum boundary needed – statically check this is provided‘

## Example

```
laplace (X*Y): | _ a _ | = (a+b+d+e)*0.25  
               | b @_ d |  
               | _ e _ |
```

```
g = grid <X = 10, Y = 10> data
```

```
g' = run laplace (defaults 0.0 g)
```

## Using program information

### Info:

- Single, non-overlapping updates
- Know exact data access
- No side effects

### Use:

- Guaranteed optimisation: `iterate`, `iterateT`
- Guaranteed parallelisation: `runPar`, `iteratePar`,  
`iterateTPar`

## Guaranteed optimisation: `iterate` (1)

- Usually want to iteratively apply `run` until some stop condition:

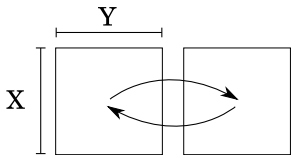
```
iterate f r g = if condition
                then iterate f r (run f g)
                else g
```

## Guaranteed optimisation: `iterate` (2)

`run` :: (Grid  $D$   $a \rightarrow b$ )  $\rightarrow$  Grid  $D$   $a \rightarrow$  Grid  $D$   $b$

`iterate` :: (Grid  $D$   $a \rightarrow a$ )  $\rightarrow$  Reducer  $a$   $Bool$   $\rightarrow$  Grid  $D$   $a \rightarrow$  Grid  $D$   $a$

- Allocations are used cyclically (cf. `swap` in the C-example)



## Guaranteed optimisation: `iterateT`

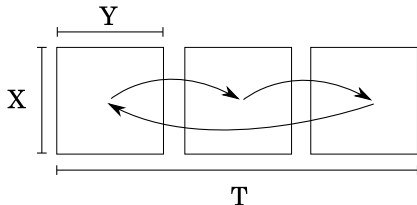
Time is just another dimension!

`iterate` :: (Grid  $D$   $a \rightarrow a$ )  $\rightarrow$  Reducer  $a$   $Bool \rightarrow$  Grid  $D$   $a \rightarrow$  Grid  $D$   $a$

`iterateT` :: (Grid ( $T \times D$ )  $a \rightarrow a$ )  $\rightarrow$  Reducer  $a$   $Bool \rightarrow$  Grid  $D$   $a \rightarrow$  Grid  $D$   $a$

Example temporal stencil (two previous grid versions):

$T : | g'' \ g' \ @\_ |$



## Guaranteed parallelisation

- `run`, `iterate`, `iterateT`  $\rightarrow$  `runPar`, `iteratePar`, `iterateTPar`
- Guaranteed: No side effects
- Guaranteed: No overlapping writes
- Guaranteed: Information on data access pattern
- Guaranteed: Local optimised behaviour of destructive update

## Guaranteed parallelisation (2)

- Grid patterns provide size of overlap regions.
- Even know if communication across some dimensions is unnecessary, e.g.

$$f(X * Y): \begin{array}{|c|c|c|} \hline & & \\ \hline 1 & @c & r \\ \hline & & \\ \hline \end{array} = g(l, c, r)$$

## Backend

- Shallow/deep embedded DSL - Template Haskell macros expand into vanilla Haskell
- Write once  $\rightarrow$  execute anywhere
- Enforce properties of language with types

## Math funsies

- Monads  $\Rightarrow$  structure (amongst others) computational effects

$$\text{bind} :: (a \rightarrow m\ b) \rightarrow m\ a \rightarrow m\ b$$

- $\text{run} :: (\text{Grid } D\ a \rightarrow b) \rightarrow \text{Grid } D\ a \rightarrow \text{Grid } D\ b$
- $\text{run} \equiv \text{comonadic extension}$
- Stencil functions,  $\text{Grid } D\ a \rightarrow b \equiv \text{coKleisli arrows}$
- Comonads  $\Rightarrow$  structure aggregate, contextual computations

## Conclusions

- Ypnos DSL: Information rich structured grid programs
- Grid patterns  $\Rightarrow$  data access information
- Purity  $\Rightarrow$  lack of side effects
- Simple primitives  $\Rightarrow$  no arbitrary read/write
- Optimisation and parallelisation is guaranteed –  
*programmer directed*
- Easier to use for non-programming experts
- Can prevent lots of wrong programs
- Port to many hardware platforms

## Further Work

- Complete backends (benchmarks imminent)
- Write a CUDA backend, C+MPI backend, PASTHA backend? etc.
- Configuration variables: tile size, memory layout, which dimensions to split etc.
- Look at multi-scale methods

## 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
initialState = grid <X=10, Y=10> randomConfiguration

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

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