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.”
• \( A \times A^{-1} = I \)
• “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
  
  ![5-point stencil](image1.png)  ![9-point stencil](image2.png)

  (a) 5-point stencil  (b) 9-point stencil

- Common in scientific computing, graphics, games, etc.
while(condition) {
    for (int i=0; i<N; i++) {
        for (int j=0; j<M; j++) {
        }
    }
    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

\[ \text{gridMap} :: (a \rightarrow b) \rightarrow \text{Grid } a \rightarrow \text{Grid } b \]

\[ f :: \text{Double} \rightarrow \text{Double} \]

\[ f \ x = \ldots \]

\[ \text{grid}' = \text{arrayMap } f \ 	ext{grid} \]
Grid data access (2)

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

\[f :: \text{Grid } \text{Double} \rightarrow \text{Double}\]
Grid data access (2)

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

\[ f :: \text{Grid} \textit{Double} \rightarrow \text{Double} \]

\[ f \mid \text{l \ @c r} \mid = \ldots \]
Grid data access (2)

run :: (Grid a → b) → Grid a → Grid b

\[
\begin{aligned}
f &:: Grid \text{Double} \rightarrow \text{Double} \\
f \mid l \@ c \mid r &= \ldots \\
grid' &= \text{run } f \text{ grid}
\end{aligned}
\]

Equivalent to:

\[
\begin{aligned}
l &= A[i-1] \\
c &= A[i] \\
r &= A[i+1]
\end{aligned}
\]
Grid data access (3)

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

\[
f :: \text{Grid } X \ \text{Double} \rightarrow \text{Double}
\]

\[
f \ X : \mid l \ @c r \mid = \ldots
\]

\[
\text{grid}^\prime = \text{run} \ f \ \text{grid}
\]
Grid data access (4)

\[ \text{run} :: (\text{Grid } D \ a \to b) \to \text{Grid } D \ a \to \text{Grid } D \ b \]

\[ f :: \text{Grid} \ (X \times Y) \ 	ext{Double} \to \text{Double} \]

\[ f \ X : | \ l \@c \ r | = \ldots \]

\[ \text{grid}' = \text{run} \ f \ \text{grid} \]

\[ l, c, r : \text{Grid } Y \ 	ext{Double} \]
Grid data access (4)

\[
\begin{align*}
\text{run} & : (\text{Grid } D \ a \to b) \to \text{Grid } D \ a \to \text{Grid } D \ b \\
\text{run} & = \text{run } f \text{ grid}
\end{align*}
\]

\[
\begin{align*}
f & : \text{Grid } (X \times Y) \ Double \to \ Double \\
f \ X & : | l \ @c r | = \ldots g \ l \ldots g \ c \ldots g \ r \ldots \\
g \ Y & : | l \ @c r | = \ldots \\
\text{grid}^\prime & = \text{run } f \text{ grid}
\end{align*}
\]
Grid data access (5)

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

\[ f :: \text{Grid} \ (X \times Y) \, \text{Double} \rightarrow \text{Double} \]

\[
\begin{array}{ccc}
\_ & t & \_ \\
\_ & @c & r \\
\_ & b & \_ \\
\end{array}
\]

\[
grid' = \text{run} \ f \ \text{grid}
\]
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.

\[ f :: \text{Grid } a \rightarrow b \]

run \( f :: \text{Grid } a \rightarrow \text{Grid } b \)
• Via grid patterns we know maximum boundary needed – statically check this is provided.
Example

\[
\text{laplace (X*Y):} \begin{array}{c|cc}
_ & a & _ \\
\hline
b & @ & d \\
\hline
_ & e & _ \\
\end{array} = (a+b+d+e)*0.25
\]

\[g = \text{grid } \langle \text{X} = 10, \text{Y} = 10 \rangle \text{ data}
\]

\[g' = \text{run laplace (defaults 0.0 } g)\]
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:

\[
\text{iterate } f \ r \ g = \begin{cases} 
\text{if condition} & \text{then iterate } f \ r \ (\text{run } f \ g) \\
\text{else } g & 
\end{cases}
\]
Guaranteed optimisation: **iterate (2)**

\[
\text{run} :: (\text{Grid } D \ a \to \ b) \to \ \text{Grid } D \ a \to \ \text{Grid } D \ b
\]

\[
\text{iterate} :: (\text{Grid } D \ a \to \ a) \to \ \text{Reducer } a \ \text{Bool} \to \ \text{Grid } D \ a \to \ \text{Grid } D \ a
\]

- Allocations are used cyclically (cf. **swap** in the C-example)
Guaranteed optimisation: iterateT

Time is just another dimension!

\[
\text{iterate} :: (\text{Grid } D a \rightarrow a) \rightarrow \text{Reducer } a \text{ Bool} \rightarrow \text{Grid } D a \rightarrow \text{Grid } D a
\]

\[
\text{iterateT} :: (\text{Grid } (T \times D) a \rightarrow a) \rightarrow \text{Reducer } a \text{ Bool} \rightarrow \text{Grid } D a \rightarrow \text{Grid } D a
\]

Example temporal stencil (two previous grid versions):

\[
T : | \ g'' \ g' \ @_ \ |
\]
Guaranteed parallelisation

- run, iterate, iterateT → 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.

\[
f(X \ast Y) : \begin{array}{c}
| & _ & _ & _ & |
\end{array} = g(l, c, r)
\]

\[
\begin{array}{c}
| l \ @c & r |
\end{array}
\]

\[
\begin{array}{c}
| & _ & _ & _ & |
\end{array}
\]
Backend

- Shallow/deep embedded DSL - Template Haskell macros expand into vanilla Haskell
- Write once → execute anywhere
- Enforce properties of language with types
Math funsies

• Monads ⇒ structure (amongst others) computational effects
  \[ \text{bind} :: (a \to m b) \to m a \to m b \]
• run :: (Grid D a \to b) \to Grid D a \to Grid D b
• \text{run} \equiv \text{comonadic} \text{ extension}
• Stencil functions, Grid D a \to b \equiv \text{coKleisli} \text{ arrows}
• Comonads ⇒ 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 – 
  \emph{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

```plaintext
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'
```