Ypnos: Declarative, Parallel Structured Grid Programming

Dominic A. Orchard
Computer Laboratory, University of Cambridge, Cambridge, UK
dominic.orchard@cl.cam.ac.uk

Max Bolingbroke
Computer Laboratory, University of Cambridge, Cambridge, UK
maximilian.bolingbroke@cl.cam.ac.uk

Alan Mycroft
Computer Laboratory, University of Cambridge, Cambridge, UK
am@cl.cam.ac.uk

Abstract

A fully automatic, compiler-driven approach to parallelisation can result in unpredictable time and space costs for compiled code. On the other hand, a fully manual approach to parallelisation can be long, tedious, prone to errors, hard to debug, and often architecture-specific. We present a declarative domain-specific language, Ypnos, for expressing structured grid computations which encourages manual specification of causally sequential operations but then allows a simple, predictable, static analysis to generate optimised, parallel implementations. We introduce the language and provide some discussion on the theoretical aspects of the language semantics, particularly the structuring of computations around the category theoretic notion of a *comonad*.

Categories and Subject Descriptors D [*3*]: 2—Applicative (functional) languages, Concurrent, distributed, and parallel languages, Specialised application languages; D [*3*]: 3—Concurrent programming structures

General Terms Design, Languages, Theory

1. Introduction

Structured grids, or *meshes*, are a key computational pattern in parallel programming [2]. In structured grid computations, a *stencil function*, or *kernel*, is applied to each element in an array-like structure (which we call a *grid*) representing a discretised real-world space. A stencil function computes a new element value from the current element value and neighbouring cells. Typically many iterations of a stencil function are performed, producing a time series of data, until some convergence condition is reached. Structured grid programs are highly data-parallel due to limited dependencies between applications of a stencil function.

Many applications employ a structured grid model of computation, particularly highly graphical programs and applications in scientific computing. Typical applications compute approximations to systems of differential equations using finite difference methods to simulate the behaviour of natural phenomena such as fluid motion, stress, heat, or other dynamic systems.

Given the complexity of modern software and ever changing (parallel) hardware, *domain-specific languages* (DSLs) offer problem or implementation specific expressivity and optimisation which cannot be achieved with more general-purpose languages. Embedding DSLs within general-purpose languages provides an inexpensive technique for implementing new languages, as elements of of the host language, such as syntax, semantics, and implementation, can be reused [27].

This paper introduces a domain-specific declarative, functional language, Ypnos, for expressing structured grid computations, and compiling such programs to parallel implementations. Ypnos is currently implemented as an *embedded domain-specific language* (EDSL) in the Haskell programming language and consists of a novel syntactic extension for expressing data access patterns, a central grid data structure, and a library of primitive operations.

Underlying Ypnos is the category theoretic notion of a *comonad* (the formal dual of a monad), which characterises structured grid computations and gives a framework for organising such computations. In its implementation, Ypnos is parameterised by a comonadic data structure, from which its primitive operations are derived. Different instances of the comonadic structure provide different back-ends to the language.

Compared to current approaches to parallel structured grid programming, Ypnos has the following advantages:

- Ypnos is a pure, functional, declarative language, thus the absence of side effects prevents the programmer writing programs which are incorrect when parallelised.
- Ypnos does not require manual expression of parallelisation, distribution, or communication like some techniques such as using C with MPI for parallelisation [26]. Parallelisation in Ypnos is handled by its primitive operations, thus the description of a problem is not obscured by implementation details.
- Ypnos does not require complex, and in general undecidable, dependency analyses, such as the polyhedral analysis [1], to facilitate automatic parallelisation due to its novel approach to data access pattern expression and its strongly-typed primitives.
- Ypnos has a predictable cost model as its compilation does not require aggressive analyses and transformations. Rather, parallelisation and optimisation are available through guaranteed program properties and primitive operations.
- Ypnos is not tied to a particular hardware implementation, unlike more low-level GPU frameworks such as CUDA [11], OpenCL [22], and Cg [19].
- Due to its restrictions, Ypnos is more simple to program with for non-programming experts than many existing approaches.

In this introduction we characterise the computational pattern of structured grids, introduce its parallelisation, and discuss issues with current solutions for parallelising structured grid problems. Section 2 introduces the core elements of the Ypnos language, including the *single, independent writes* property of Ypnos programs. Section 3 gives further Ypnos primitives, followed by a discussion of the optimisation and parallelism provided by Ypnos

DAMP'10, January 19, 2010, Madrid, Spain.

primitives. Small examples are given throughout, with some further examples in Section 5. Section 6 gives some information on the proof-of-concept implementation, followed by a discussion of the comonadic structure of Ypnos and its back-end in Section 7.

An in-depth knowledge of Haskell is not required.

1.1 Structured Grid Computations

The crux of the structured grid model of computation is the application of a stencil function to all elements of an array-like data structure. A stencil function computes a new value for a grid element based on the current value and the values of a fixed set of neighbouring elements. **Figure 1** gives an example of a C-style program with a 5-point stencil access pattern (as illustrated in **Figure 2**), computing the mean of an element's neighbours.

```
while (condition) {
  for (int i=0;i<N;i++){
    for (int j=0;j<M;j++){
      Atemp[i][j] = (A[i][j]+A[i+1][j]+A[i-1][j]+
                     A[i][j-1]+A[i][j+1])/5.0; } }
  swap(Atemp, A); }
```

Figure 1. An example stencil computation in an imperative C-like language, computing the mean of surrounding elements in array `A`.

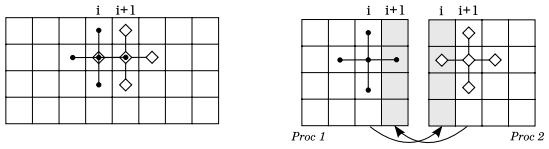
Note that results are written to a temporary array, `Atemp`, such that computation proceeds without interference between newly computed values and the values of the previous iteration. After each iteration `A` and `Atemp` are swapped (perhaps by exchanging pointers), so that the next iteration reads from the array written to in the previous iteration, and vice-versa.

1.1.1 Parallelisation

Structured grid programs have been parallelised for decades on symmetric-multiprocessor systems (SMPs), clusters, and now multi-core systems by the domain decomposition technique of partitioning grids into subgrids, which are distributed to processing elements for independent parallel computation. The size of a stencil function's data access pattern is usually small relative to the overall size of a grid thus dependencies between subgrids are minimal.

On a distributed memory architecture, each subgrid resides in the local memory of a processor. Where data dependencies lie outside a subgrid, data from neighbouring subgrids is replicated at the boundaries and updated after each iteration by inter-process communication (illustrated in **Figure 2**). For multiple iterations, data persists in local memory until the full data set is required.

With a shared memory model, a grid is partitioned by defining subsets of the iteration space for each processor. Values can be accessed from the shared memory in other iteration spaces where dependencies exist between subcomputations.



(a) Before decomposition (b) After decomposition

Figure 2. 2D domain decomposition with a 5-point stencil

1.2 Problems with Current Approaches

We motivate our language design by looking at a number of issues with current methods of parallel structured grid programming.

(Issue 1) Manual parallelisation is difficult to express, error prone, and hard to debug.

A common manual parallelisation approach to structured grid programming uses C or FORTRAN with the Message Passing Interface (MPI) [26]. Partitioning, distribution, and communication must be programmed by hand, often resulting in mixing of algorithmic parts of a program with parallelisation code. The manual approach becomes increasingly difficult with higher-dimensional grids, increasing numbers of grids, and more algorithmic stages, sometimes taking many days or weeks of programming [21].

(Issue 2) Imperative programming languages have a high potential for producing incorrect parallel programs.

Imperative programming languages are popular and attractive as they typically offer very good sequential performance and a predictable cost model for execution. However unrestricted side effects can result in programs which, when parallelised, are incorrect.

(Issue 3) Some approaches are too hardware specific.

Frameworks such as CUDA [11], OpenCL [22], and Cg [19] express stencil computations as *shaders* for execution on GPUs. Manual partitioning and distribution is generally not necessary as shaders are automatically scheduled on GPU cores with chunks of an input data stream. These frameworks are specific to GPU implementation thus cannot be executed on non-GPU hardware. Another example, OpenMP, is specific to shared-memory systems [7]. A complete rewrite is required for execution on other hardware.

(Issue 4) Automatic parallelisation in general-purpose languages is usually undecidable & can lead to unpredictable compiled code.

Automatic parallelisation requires sufficient information to be communicated to the compiler about the structure and dependencies of a program. The compiler must ascertain whether a program fits a specified model of computation before it can parallelise the program. Automatic parallelisation of a structured grid program requires sufficient information about which arrays have stencil-like functions applied, data access patterns, how much boundary communication is required between processors, and how different stages of a computation interact.

A fully automatic approach to parallelisation, even if successful, can lead to discontinuous compiler behaviour i.e. a small change in a program can result in a significant change in the topology of the computation and its efficiency because a small change can render an analysis or transformation intractable. This unpredictability can lead to much frustration for the programmer who must fathom how to appease the compiler.

(Issue 5) Random-access operations on arrays and arbitrary indexing renders analysis and automatic compilation difficult.

There is much work on loop dependency analysis, encompassing array access, for loop restructuring optimisations and parallel compilation. Given unrestricted pointer operations, aliasing, and arbitrary array indexing, it is in general undecidable to infer exact dependency information for all programs. Therefore, many analyses require that programs adhere to a number of constraints, such as *affine* indices (indices computed from just scalar multiplication of indices and addition of constants).

For example, a polyhedral analysis models the iteration spaces of nested loop structures as geometric objects. A polyhedral model is built from *static control parts* of a program which adhere to a

number of restrictions: constant loop step size, affine loop bounds, if-statement predicates based on affine expressions, affine indices, and use of functions that do not communicate using side effects. [1]. Many scientific computing applications do however conform to these analysis requirements [5], although the problems of *(Issue 4)* apply, and can be especially frustrating for novice programmers.

2. Ypnos

Ypnos is a declarative, functional, *domain-specific language* for structured grid programming. Currently, Ypnos is implemented as an *embedded* domain-specific language (EDSL) in the pure, functional programming language Haskell, benefiting from existing language syntax, semantics, abstractions, implementation, and libraries. Ypnos extends Haskell with some its own syntax, and benefits Haskell by providing an EDSL for parallel structured grid programming. Other host-language embeddings may be given, but we find Haskell the most convenient for our purposes.

The restricted, domain-specific nature of Ypnos means its programs fit the structured grid model of computation. Additionally, syntactic extensions provide decidable compile-time information about a program's data access and dependencies. Thus, Ypnos does not require complex analyses and transformations to identify the structured grid pattern in a program and to parallelise its execution (addressing *Issue 4* and *5*). Ypnos does not require manual partitioning and communication code because its restricted operations and constructions handle parallelism (addressing *Issue 1*). Furthermore, the Ypnos EDSL is parameterisable by different back-end implementations, permitting execution on different architectures and platforms (addressing *Issue 3)*. We use Haskell's strong typing to reject programs not matching the correct computational pattern, and to enforce absence of side effects beyond those permitted in the back-end of Ypnos, which can hinder parallelisation (addressing *Issue 2*).

This introduction to Ypnos commences with the central Grid data structure.

2.1 Grids

The Grid data structure represents a finite n-dimensional discrete space of values, and is parameterised by a *dimension* and *element type*. The dimension defines the number of *axes* belonging to a grid, giving distinct identifiers to each. For example, a two-dimensional grid of floating point values of dimension $X \times Y$ has type:

$$\mathsf{Grid}\ (X \times Y)\ \mathit{Float}$$

akin to a C array type `float[][]`. We use identifiers X, Y, and Z to denote particular dimensions, dim to range over all dimension identifiers, and D to range over all dimension terms. A dimension term is a type-level construction that is formed from a number of dimension identifiers and a tensor product operation $\times$ i.e:

$$D := \mathit{dim} \mid D\ \times D$$

Unlike C, Ypnos has an infinite number of two-dimensional grid types for some element type a because dimension identifiers are not equal e.g. $X \times D \neq Y \times D$. The tensor product operation is however associative and commutative, hence: $D_1 \times D_2 \equiv D_2 \times D_1$ and $D_1 \times (D_2 \times D_3) \equiv (D_1 \times D_2) \times D_3$.

The primitive function grid, constructs grids from a vector of finite dimension sizes and a list of elements, e.g.

$$\mathsf{grid}\ \langle X = 2, Y = 2\rangle\ [1, 2, 3, 4]$$

In the type of a function, the dimension parameter of a grid type may be universally quantified. In the primitive operations, D denotes a universally quantified dimension term.

2.2 Stencil Functions

Programs in Ypnos are written mostly in terms of user-defined stencil functions. Consider the example C program in **Figure 1**. This program can be abstracted on its stencil function, parameterising the computation by a stencil function `f`:

```
...
  Atemp[i][j] = f(A,(i,j));
...
```

`f` has the following type, where Array a is an array of element type a and (*Int*, *Int*) is a two-dimensional index:

$$(\mathsf{Array}\ a \times (\mathit{Int} \times \mathit{Int})) \rightarrow a$$

In Ypnos, stencil functions have type:

$$\mathsf{Grid}\ D\ a \rightarrow b$$

In the type of `f`, (*Int*, *Int*), is the index for the current position at which the stencil function is being applied. In Ypnos, the current index of application is hidden inside the Grid structure, which we call the *cursor* or *focal point* of the stencil. Instead of using array indexing operations, values are accessed from the grid using a novel pattern matching construction called a *grid pattern*.

2.3 Grid Patterns

A grid pattern consists of a number of sub-patterns which are matched to the elements of a grid based on their lexical order in relation to a central point. The following is an example of a one-dimensional grid pattern:

$$X : |\ \ l\ \ @c\ \ r\ \ |$$

This pattern binds the variables l, c, and r to consecutive elements in a grid along the dimension X. The variable which is bound to the *focal point* of the grid is delineated by the @ symbol. This grid pattern is analogous to following bindings in a C-like language where `A` is an array and `i` is the current index:

```
l = A[i-1];    c = A[i];    r = A[i+1];
```

The cursor (equivalent to the above index, `i`) is used by the implementation to give correct bindings to grid patterns, as the cursor denotes the position of the focal element in a grid during a stencil computation.

One-dimensional grid patterns can be nested inside one another to give n-dimensional patterns. Alternatively, variables can be bound in one dimension and passed to another stencil function. For example, if the above one-dimensional stencil was applied to a grid of dimensions $X \times Y$ each bound variable would correspond to an array slice of dimension Y which could be further matched upon by a separate stencil function.

As a syntactic convenience we provide an additional two-dimensional grid pattern whose concrete syntax spans multiple source lines; a change in line corresponds to an increment in the second dimension's index. For example, the following pattern is the standard 5-point stencil in dimensions $X \times Y$:

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

As an example of concrete syntax, the following is a complete stencil function with a grid pattern that matches elements in dimensions $X \times Y$ and computes the mean:

```
ave2D :: Grid (X * Y) Double -> Double
ave2D (X * Y): | _ t  _ | = (t+l+c+r+b)/5.0
               | l @c r |
               | _ b  _ |
```

Grid patterns directly express data access as part of a computation, they are not simply annotations given to purportedly describe the access pattern of a separate piece of low-level code. Because the grid patterns are static they provide decidable compile-time information. If conditional expressions are used to choose between stencils then a decidable over-approximation calculates the union of all possible data access patterns for the outer stencil function.

At compile time a bounding box of a grid pattern is constructed, capturing the amount of boundary communication and overlap required between subgrids when parallelising via domain decomposition. The maximum distance of subpatterns from the focal point, in each dimension, defines a bounding matrix. In the case of the `ave2D` example, a single row and column of data is required to be replicated at subgrid boundaries in each direction, thus the following bounding matrix is inferred:

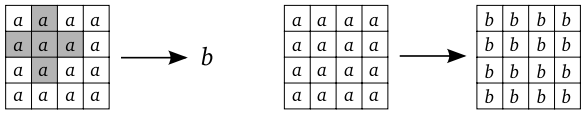
$$[\![\texttt{ave2D}]\!]_{access} = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$$

2.4 Applying Stencil Functions

The run primitive applies a stencil function to a grid, and has type:

$$\mathsf{run} :: (\mathsf{Grid}\ D\ a \to b) \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ b$$

The run primitive takes a stencil function as its first parameter and a grid as its second, applying the stencil function to the grid, once for every possible grid position, instantiating the grid's cursor to the index of each element. A value of type b is returned for every position in the grid which run returns in a new grid of element type b. **Figure 3(a)** illustrates a stencil, and **Figure 3(b)** illustrates run once it has been partially applied to a stencil function.



(a) Illustration of a stencil function $f :: \mathsf{Grid}\ a \to b$

(b) Illustration of run $f :: \mathsf{Grid}\ a \to \mathsf{Grid}\ b$

Figure 3. Illustrations stencil function application to a grid.

Note that the stencil function is applied to every element in the grid, therefore applications of a stencil applied at, or near, the edge of the grid may attempt to bind values outside of the grid. Default values outside of a grid can be specified, as well as more complex behaviours (see Section 3.4).

2.5 Comparing run to the *map* function on lists

Applying a stencil function with a grid pattern matching just the focal point is equivalent to a *map* function over grids. i.e.

$$\mathsf{run}\ (\lambda|@c|\,.\,f\ c)\ g \equiv \mathit{mapG}\ f\ g$$

Recall the standard definition of *map* over lists:

$$\mathit{map}\ f\ [x_1, \ldots, x_n] \equiv [f\ x_1, \ldots, f\ x_n]$$

Consider a function *map2*, which applies a binary function to consecutive pairs of elements from a list, of type

$$\mathit{map2}\ ::\ ((a \times a) \to b) \to [a] \to [b]$$

The operation of *map2* may be something like:

$$\mathit{map2}\ f\ [x_1, \ldots, x_n] \equiv [f(x_1, x_2), \ldots, f(x_{n-1}, x_n)]$$

The number of elements in the results list is one fewer than the parameter list. A default *boundary* value d remedies this discrepancy:

$$\mathit{map2}\ f\ [x_1, \ldots, x_n] \equiv [f(x_1, x_2), \ldots, f(x_{n-1}, x_n), f(x_n, d)]$$

An alternate version of *map2* might use the default value at the beginning of the list instead of at the end:

$$\mathit{map2'}\ f\ [x_1, \ldots, x_n] \equiv [f(d, x_1), f(x_1, x_2), \ldots, f(x_{n-1}, x_n)]$$

The Ypnos equivalent stencil computation of *map2* on grids is:

$$\mathsf{run}\ (\lambda|@x\ y|\,.\,f(x, y))\ g$$

The equivalent stencil computation of *map2'* is:

$$\mathsf{run}\ (\lambda|x\ @y|\,.\,f(x, y))\ g$$

map2 and *map2'* show the effect of two subtly different stencils, one binding the current element and the element one position to the right, the other binding the current element and element to the left.

2.6 Summary

We have introduced the central Grid data structure and the type-level concept of dimensions to differentiate between different dimensional grids. Stencil functions in Ypnos have type: $\mathsf{Grid}\ D\ a \to b$. The run combinator applies stencil functions to grids:

$$\mathsf{run} :: (\mathsf{Grid}\ D\ a \to b) \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ b$$

The Grid data structure contains a *cursor* which is instantiated by the run combinator to each position in the grid. The hidden cursor is required by the implementation to give the correct bindings of *grid patterns*. Grid patterns bind values from the grid to variables where the *focal point*, or current element, is delineated with an @ symbol. Grid access is defined statically with no dynamic index expressions and thus is known statically at compile time.

Grid patterns restrict stencil functions to locally dense, globally sparse, access patterns. That is, the neighbourhood of elements accessed from the grid around the focal point is relatively small in comparison with the problem size. Therefore dependencies between subgrids are minimal, reducing expensive inter-process communication under parallelisation via domain decomposition. It is syntactically inconvenient to write overly large stencils (greater than about 5 elements in each direction), thus communication requirements are kept relatively small.

Ypnos' programs have what we call the *single, independent writes* (SIW) property: that application of a stencil function makes a single write to the focal position, thus writes never overlap. This property is enforced by the types of stencil functions (producing a single value), the run combinator (applying the stencil function once per element), and the absence of random-access write operations. The guaranteed SIW property is leveraged for optimisation and parallelisation in Section 4.

3. Further Ypnos

Ypnos' primitive operations, of which run and grid have already been seen, are listed along with their types in **Figure 4**.

3.1 Tuples of values

The zip, and complementary unzip, operations respectively pair the elements of two grids and split a grid of pairs into a pair of grids. These operations are especially useful when performing stencil operations on several parameter grids. There are corresponding operations for 3-tuples, 4-tuples etc.

Extra syntactic sugar allows grid patterns on grids of tuples to be written as a tuple of grid patterns e.g.

```
f | lA @cA rA |,| lB @cB rB |
```

instead of

```
f | (lA, lB) @(cA, cB) (rA, rB) |
```

Note each grid pattern must be equal in size and have the focal element in the same position.

3.2 Reductions

A common part of structured grid computations is the *reduction* of grids to a single result such as the mean, maximum value, or sum. Ypnos provides a simple reduction primitive: reduce, which takes an associative reduction operator of type $(a \to a \to a)$ and applies the operation in parallel to all elements and partial results.

Some reductions generate values of a different type to the element type of a grid. A structure called a Reducer packs together a number of functions for parallel reduction under reduction operators of this type. The mkReducer constructor builds a Reducer, taking four parameters:

- A function reducing an element and partially-reduced value to another partially-reduced value: $(a \to b \to b)$
- A function combining two partially-reduced values, possibly from two reduction processes on subgrids: $(b \to b \to b)$
- An initial partial result: b
- A final conversion function that converts the partial-result to a final value: $(b \to c)$.

There are a number of built-in reducers: max, min, sum, and, or, and mean. A Reducer structure can be applied to a grid using the reduceR primitive. The iterate and iterateT operations also take a Reducer as a parameter.

3.3 Iterative stencil application

Similar to run, the iterate primitive iteratively applies the stencil function over a grid until a stop condition is reached. The iterate operation takes a stencil function, a Reducer of boolean result for the stop condition, and a parameter grid. The iterate operation can be derived from reduce and run, although the primitive provided uses local mutable state for optimisation (see Section 4.1):

$$\begin{aligned}\mathsf{iterate}\ f\ r\ g = {} & \text{if } (\mathsf{reduceR}\ r\ g) \\ & \text{then } \mathsf{iterate}\ f\ r\ (\mathsf{run}\ f\ g) \\ & \text{else } g\end{aligned}$$

i.e. if the stop condition is not reached then apply run and recurse on the result, if the stop condition is reached then return the current result. Note that the parameter grid and return grid must have the same element type for iterative stencil function application, as the result of one application is passed to the next.

3.4 Boundaries

Stencil functions applied near or at the edge of a grid may case out-of-bounds data access. One approach to handling boundaries in Ypnos is to manually account for boundary conditions via the use a # pattern in a grid pattern which binds the indices of the focal point (cursor) to a variable. For example, the following pattern binds i to the index of the current focal point:

$$X : |\ l\ \ @c\#i\ \ r\ |$$

An out-of-bounds check can be performed on the index, providing values to otherwise undefined variables. This feature can be used to implement a type of random access on grids, but the function must still be applied to a grid using run, thus would be vastly inefficient. The index checking technique has the added caveat that each application of a stencil requires extra index-testing control flow. Extra control flow can be especially undesirable when compiling to a GPU, which often has more costly control flow operations.

An alternative, more efficient, solution is provided by lift which lifts a finite grids to an infinite grid, allowing out-of-bounds access. The lift primitive takes a grid and a *facets* structure (essentially a record) which describes boundary behaviour for each facet (edge, face, etc.) of a grid. Possible boundary behaviours include: default values, *wrapping* (reading a value from the opposite side of a grid), or *reflecting*. A complementary unlift primitive returns a finite grid from an infinite grid. For space reasons, we omit a discussion of the exact nature of the facets structure.

$$\begin{aligned}
\mathsf{grid} &:: \langle D\ \mathit{Int}\rangle \to [a] \to \mathsf{Grid}\ D\ a \\
\mathsf{zip} &:: \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ b \to \mathsf{Grid}\ D\ (a,b) \\
\mathsf{unzip} &:: \mathsf{Grid}\ D\ (a,b) \to (\mathsf{Grid}\ D\ a, \mathsf{Grid}\ D\ b) \\
\mathsf{reduce} &:: (a \to a \to a) \to \mathsf{Grid}\ D\ a \to a \\
\mathsf{reduceR} &:: \mathsf{Reducer}\ a\ b \to \mathsf{Grid}\ D\ a \to b \\
\mathsf{mkReducer} &:: \exists b\ (a \to b \to b) \to (b \to b \to b) \to b \\
&\qquad \to (b \to c) \to \mathsf{Reducer}\ a\ c \\
\mathsf{run} &:: (\mathsf{Grid}\ D\ a \to b) \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ b \\
\mathsf{iterate} &:: (\mathsf{Grid}\ D\ a \to a) \to \mathsf{Reducer}\ a\ \mathit{Bool} \\
&\qquad \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ a \\
\mathsf{iterateT} &:: (\mathsf{Grid}\ (T \times D)\ a \to a) \to \mathsf{Reducer}\ a\ \mathit{Bool} \\
&\qquad \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ a \\
\mathsf{lift} &:: \mathsf{Grid}\ D\ a \to \mathsf{Facet}\ D\ a \to \mathsf{Grid}^{\infty}\ D\ a \\
\mathsf{unlift} &:: \mathsf{Grid}^{\infty}\ D\ a \to \mathsf{Grid}\ D\ a \\
\mathsf{defaults} &:: \mathsf{Grid}\ D\ a \to a \to \mathsf{Grid}^{\infty}\ D\ a \\
\mathsf{run}^{\infty} &:: (\mathsf{Grid}^{\infty}\ D\ a \to b) \to \mathsf{Grid}^{\infty}\ D\ a \to \mathsf{Grid}\ D\ b
\end{aligned}$$

Figure 4: Key primitive operations on structured grids

All but one of the primitive operations can be called with lifted grids, returning a lifted grid. The exception is run, which is unable to return a lifted grid when applied to a lifted grid because run allows the element type of the returned grid to be different to that of the parameter grid, thus a facet structure of the correct element type is unavailable. The type of run^{∞} reflects this behaviour.

A simple lifting operation, defaults, provides support for adding default values infinitely outside the boundaries of a grid.

3.5 Other considerations

In numerical analysis there are a number of techniques that speed up convergence, such as *Gauss-Seidel iterations* and *relaxation* methods such as *Successive Over Relaxation* (SOR). These techniques can be applied in practice to speed up computations.

Gauss-Seidel iterations compute results based on already computed results in the current iteration [9]. For example:

```
for(i=0;i<N;i++){
  A[i] = (A[i-1] + A[i+1])*0.5;
}
```

Ypnos provides Gauss-Seidel support with iterateInplace which causes a stencil function to read and write the same grid in memory.

In the SOR technique, grids are often processed in a checkerboard fashion, first computing values for odd elements, and then for even elements. There is currently no support for this in Ypnos, although a system of execution *masks* is under consideration.

4. Optimisation and Parallelisation

The guaranteed *single, independent writes* (SIW) property, and the static data access information provided by grid patterns, means optimised and parallel implementations can be given to Ypnos primitives, which are safe, i.e. do not alter program correctness, and are guaranteed to be applicable. Parallel and/or optimised implementations of run, iterate, and iterateT are provided by the back-end grid structure parameterising the Ypnos EDSL (see Section 6).

4.1 Optimisation by Destructive Update & Allocation Reuse

It is well known that the absence of side effects and the immutability of data structures is advantageous for parallelism but disadvantageous for updating large data structures, such as arrays, as time is spent copying, allocating, and deallocating data [13]. Ypnos provides optimised primitives, iterate and iterateT, which use local destructive array updates to reduce execution time spent in allocation, deallocation, and garbage collection.

The iterate operation (introduced in Section 3.3) can be given an optimised implementation where a pair of auxiliary mutable grids store intermediate results. One mutable grid stores the values of the previous iteration, the other is destructively updated as the current iteration is computed. At the end of an iteration the role of each allocation is swapped (illustrated in **Figure 5(a)**), similar to the swap operation of **Figure 1**. Due to the SIW property, write operations never interfere, thus destructively writing to a temporary grid structure is safe. When the stop condition of iterate is satisfied, all local effects are applied and a pure grid is returned, thus the effects of the destructive update are hidden inside iterate.

The iterateT primitive extends the (internally) destructive behaviour of iterate to computations involving past versions of a grid, without the programmer having to alias grids between calls to run. The iterateT primitive embeds a grid into a reserved temporal dimension T over which grid patterns are reused to define the exact number of previous grid versions required in a computation. Given this information, iterateT keeps in memory enough copies of the grid to satisfy this *historic* grid pattern. When previous grid allocations can no longer be accessed by the stencil function, iterateT safely reuses these allocations cyclically. For example, the following temporal stencil matches the two preceding iterations[1]:

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

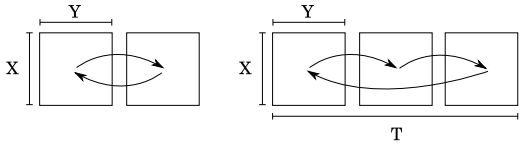
Subsequently, iterateT creates three mutable grid allocations which are cyclically written to in-place (illustrated in **Figure 5(b)**). Because of the static guarantees of grid patterns, iterateT obviates the need for an alias analysis, or a system such as *linear types* [31], to ensure that intermediate grids are not aliased and then used later once destructively updated, reallocated, or deallocated.

Neither iterate nor iterateT extend the expressive power of Ypnos, they are merely optimised forms of operations which could be derived using run and reduces. The use of either primitive communicates to the compiler that the programmer wishes to use the optimisation of mutable state. For iterateT, the programmer also communicates to the compiler the exact number of previous grid allocations required. Thus, these optimisations are not fully-automatic (i.e. they are not result of a compiler analysis and transformation) and are not manual either, but *human-driven*, with predictable gains thanks to the static guarantees of Ypnos.

4.2 Parallelisation

Due to the *single, independent writes* property, individual applications of a stencil function to a grid may be safely performed in any order or in parallel, as read and write operations do not interfere.

[1] Note, the focal point matching the current version must be a wildcard pattern, as iterateT marks the current grid version as undefined



(a) iterate (b) iterateT $(\lambda T : |g''g'@\_| \ldots)$

Figure 5. Allocation reuse patterns of optimised primitives

Applying each stencil function in parallel over a grid would however incur a large overhead from each thread creation. Instead, we can perform domain decomposition (described in Section 1.1.1), factoring a grid into subgrids for independent parallel computation.

Because of the static data access information afforded by grid patterns, the exact amount of boundary communication required between subgrids is known. Thus run, iterate, and iterateT primitives can be given parallel implementations which are guaranteed to be safe (by SIW) with the correct, and minimal, amount of communication between processes as specified by the programmer. We informally describe the parallel implementation of run for a stencil function f, applied to a grid g, with access pattern $acc = [\![\mathtt{f}]\!]_{access}$ (see bounding matrix in Section 2.3), where $acc[D]$ means the vector at the D-th column of the bounding matrix, representing the data access pattern in dimension D.

1. Split g into subarrays gs, adding boundaries of size $acc[D]$ for dimensions D from g or from the associated facets structure if at the grid edge. Prefer splitting along a dimension where $\sum acc[D] = 0$ i.e. no boundary communication required.
2. Spawn threads of runSeq f g", for each subgrid g" $\in$ gs, where runSeq is a sequential run operation.
3. Join the computed subgrids into a single grid g', removing boundaries, and return.

For iterate and iterateT we apply a similar scheme, where sequential, optimised versions of iterate and iterateT are applied to each subgrid in independent threads. After step 2, boundary communication between subgrids takes place, where boundary data is swapped with neighbouring processes, before testing the stop condition, moving to step 3 if true or iterating if false.

5. Further Examples

Solving The Laplace Equation The following example solves the Laplace equation using a Gauss-Seidel method over a grid.

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

g' = iterateInplace laplace (ntimes 1000) (defaults g 0.0)
```

A special impure reduction operator, ntimes, counts the number of iterations, reducing to *true* when a specified limit is reached.

Conway's Game of Life It has been noted online by Piponi that cellular automata are comonadic [24]. In a similar vein, the following example computes Conway's Game of Life in Ypnos.

```
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>2)
                              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'
```

The above code makes use of a special reduction combinator, orReducer, which creates a disjunction of two reducers with boolean results. There is a similar combinator for conjunction.

Writing Non-Structured Grid Applications The following, slightly contrived, example does not fit the structured grid programming pattern because it uses random-access writes to an array. We show the implementation of the program first in C and then show an implementation in Ypnos.

```
int A[5] = {1,4,2,3,0}; int Atemp[5];

for (int i=0;i<5;i++){
  int x = A[i];
  Atemp[x] = i;
}
```

A five element one-dimensional array is initialised with integers from 0 to 4. In the loop, each array value is used to index `Atemp`, where the current index `i` is written. The computation is $\in \mathcal{O}(n)$.

The equivalent Ypnos code uses a stencil function to emulate random writes, comparing the focal point index to the write index; if equal, the write element is returned, else the current grid value.

```
g = grid <X=5> [1,4,2,3,0]
g' = grid <X=5> [0,0,0,0,0]

randomWrite grid x i = run randomWrite' grid
   where
     randomWrite' | @c#j | = if (i==j) then x else c

reArrange | @c#i | = randomWrite g' i c

run reArrange g
```

The program is certainly inelegant to write in Ypnos, is slow ($\in \mathcal{O}(n^2)$), and returns a grid of grids, using more memory and presenting difficult-to-compile code for a GPU target. The infelicity of the programming task reflects its non-structured grid pattern.

6. Implementation

The current Haskell EDSL implementation of Ypnos is parameterised by a number of data structures which provide a semantics to Ypnos, with Haskell as the meta language (see Section 7). In this way, different back-end implementations can be given, such as a parallelising back-end, or one that generates C with MPI or CUDA. Section 7.2.1 gives a simple, pure, sequential semantics.

Syntactic extensions for grid patterns are implemented using the quasiquoting extension to Template Haskell [18] which are parsed into an AST using the Parsec parser combinator library [17]. The use of the quasiquoting technique adds some extra syntactic burden which we omitted in our examples for clarity. The following is an example stencil function in the current implementation:

```
ave2D = [$fun| X*Y:| _  t  _ |
                   | l @c  r |
                   | _  b  _ | -> (t+l+c+r+b)/5.0 |]
```

where `fun` preprocesses the grid pattern and generates grid access code, taking the expression after `->` as Haskell code which is translated verbatim as the stencil function body. The macro generates a tuple of the stencil function with a vector of grid's data access pattern and dimension information.

7. Mathematical Structure

The core of Ypnos is structured around the Grid type which models a *comonad* structure from category theory. The operations of the comonad correspond to, or are used to derive, the core primitive features of our language. The optimised primitives using mutability (iterate and iterateT) combine a comonad with a monad, describing their effects, via a *distributive law* (as in [25]).

The category theory structures used to organise the language give a clear separation of the concepts underlying Ypnos and facilitates modular back-end implementations. Abstractly, the associated coherence laws of comonads, monads, and distribution give properties that the back-end implementer should be mindful of if correct execution is expected. Although Haskell does not support the encoding of these properties, they can be checked by hand.

We briefly introduce the abstract structures used by Ypnos, and describe a sequential, pure instantiation of the Grid comonad that provides a simple semantics to the language.

7.1 Comonad

Until recently comonads have received less attention in programming than their dual: *monads*, traditionally used to describe computational effects [20]. Uustalu and Vene showed that the stream-based dataflow computations of the Lucid programming language can be described with a comonad [28]. Their thesis: comonads capture the *essence* of dataflow. The Lucid programming language, originally purposed for declarative iteration, can be understood as a language of context-dependent computations, where computations are modelled as streams mapping discrete time contexts to values [30]. Lucid was later extended to multi-dimensional streams, where contexts are Cartesian coordinates [3]. Multi-dimensional streams are akin to array structures. Our hypothesis is that comonads also capture the essence of structured grid computations.

Definition For a category $\mathcal{C}$ and an endofunctor $D : \mathcal{C} \to \mathcal{C}$, a *comonad* is a triple (D, ε, δ) of D and two natural transformations, where:

[C1] $\varepsilon : D \to 1_{\mathcal{C}}$
[C2] $\delta : D \to D^2$
[C3] associativity and identity laws hold for objects in $\mathcal{C}$

If we interpret the comonad's endofunctor D as a data structure in a functional language, the ε operation corresponds to a polymorphic function that extracts a value from the comonad (*counit*), and δ corresponds to a polymorphic function that expands a comonad into a nested comonad inside a comonad (*cojoin*):

$$\mathit{counit} :: \forall a\ .\ D\ a \to a$$
$$\mathit{cojoin} :: \forall a\ .\ D\ a \to D\ (Da)$$

The functional programming interpretation of functor application to morphisms is a higher-order function called *fmap*, which lifts a function to operate over a data structure:

$$\mathit{fmap} :: \forall a\ .\ (a \to b) \to (D\ a \to D\ b)$$

Comonads are defined above in the *comonoid* form, from which the *coextension* form can be derived. In the coextension form, a *coextension* natural transformation, often called *cobind* in functional programming[2], takes a function from a comonad $(D\ a)$ to an object (b) and lifts it to a function from $(D\ a)$ to a comonad $(D\ b)$. It can be derived by composition of functor application with *cojoin*:

$$\mathit{cobind} :: (D\ a \to b) \to (D\ a \to D\ b)$$
$$\mathit{cobind}\ f = (\mathit{fmap}\ f) \circ \mathit{cojoin} \quad (1)$$

[2] The extension form of a monad is often called *bind* in functional programming, hence the naming *cobind*.

The first parameter of *cobind* is known as a *coKleisli* morphism, or *arrow*. Given a comonad D in category $\mathcal{C}$, a *coKleisli* category $\mathcal{C}_D$ has morphisms $f, g : D\ A \rightarrow B$ from $\mathcal{C}$. The *cobind* operation allows such coKleisli arrows to be composed:

$$(g \circ_D f) = g \circ (\textit{cobind}\ f)$$

7.2 Grids as Comonads

We briefly describe the derivation of Ypnos primitives from the comonadic, parameter Grid structure:

- Grid comonads are parameterised by a dimension D, i.e. $(\mathsf{Grid}\ D)$ is the comonad.
- The run operation of Ypnos is exactly the *cobind* operation.
- Stencil functions are the coKleisli arrows of the comonad.
- Grid comonads are *copointed*, having a *cursor* (focal point); *counit* returns the grid value pointed to by its cursor.
- Grid comonads are *symmetric semi-monoidal comonads* [29] thus are equipped with the natural transformation:

  $$\textit{zip}_{A,B} : D\ A \times D\ B \rightarrow D\ (A \times B)$$

  providing zip; unzip is provided by a pair of the left and right tuple projections lifted to grids by *fmap*:

  $$\mathsf{unzip}\ x = (\textit{fmap}\ \pi_1\ x,\ \textit{fmap}\ \pi_2\ x)$$

- Grid comonads require additional *shift* operations (like the navigation principles of a *zipper* [14]) to modify the cursor, allowing grid patterns to access elements relative to the focal point of a grid. A grid comonad must provide *shiftLeft* and *shiftRight*, parameterised by the dimension D' in which to navigate:

  $$\textit{shiftLeft}\ ::\ D'\ \rightarrow \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ a$$
  $$\textit{shiftRight}\ ::\ D'\ \rightarrow \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ a$$

- Reduction (reduce, etc.) and lifting operations (lift, etc.) must be supplied as additional operations specific to the Grid comonad.

7.2.1 A Pure, Sequential Grid Comonad

The following comonadic Grid data structure provides a pure, sequential semantics to Ypnos. The grid structure is a pair of a cursor and a function from indices to values, where indices are Cartesian products of integers:

$$\mathsf{Grid}\ D\ A = \mathbb{N}^{|D|} \times (\mathbb{N}^{|D|} \rightarrow A)$$

Such grids are infinite. The Grid functor's operation on morphisms applies the parameter function to each element of the grid:

$$\textit{fmap}\ ::\ (a \rightarrow b) \rightarrow (\mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ b)$$
$$\textit{fmap}\ f\ (c, g) = (c, (\lambda n\ .\ f(g\ n)))$$

The *counit* operation returns the value at the location pointed to by the cursor. The *cobind* operation can be derived from *cojoin* and *fmap* on Grid as in (1), but we give a specialised version here.

$$\textit{counit}\ ::\ \mathsf{Grid}\ D\ a \rightarrow a$$
$$\textit{counit}\ (c, g) = g\ c$$
$$\textit{cobind}\ ::\ (\mathsf{Grid}\ D\ a \rightarrow b) \rightarrow \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ b$$
$$\textit{cobind}\ f\ (c, g)\ =\ (c, (\lambda n\ .\ f\ (n,\ g)))$$

The *zip* operation is defined:

$$\textit{zip}\ ::\ \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ b \rightarrow \mathsf{Grid}\ D\ (a, b)$$
$$\textit{zip}\ (c, g)\ (c', g')\ =\ (c, \lambda n\ .\ (g\ n, g'\ n)))\ \ \textit{where}\ c = c'$$

Shifting operations alter the focal point, where $[d \mapsto 1]$ is an index where the d-th element is 1 and all other elements are 0:

$$\textit{shiftLeft}\ ::\ D'\ \rightarrow \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ a$$
$$\textit{shiftLeft}\ d\ (c, g) = (c - [d \mapsto 1], g)$$
$$\textit{shiftRight}\ ::\ D'\ \rightarrow \mathsf{Grid}\ D\ a \rightarrow \mathsf{Grid}\ D\ a$$
$$\textit{shiftRight}\ d\ (c, g) = (c + [d \mapsto 1], g)$$

Lifting and reduction operations are omitted for spaces reasons.

7.3 Effectful Primitives

The optimised primitives, iterate and iterateT, generate and apply side effects internally thus the effects cannot "escape" and produce global effects. Conceptually, and ideally, the effectful primitives would be implemented using a state *monad* [20], and a distributive law to combine the state monad and Grid comonad [25].

The distribution operation, corresponding to the distributive law, has the following type (D is a comonad, T is a monad):

$$\textit{dist}\ ::\ D\ (T\ a) \rightarrow T\ (D\ a)$$

Thus, local effects in the elements of the comonad are distributed outside of the comonad. Using *dist* we derive an operation that we called *bibind*, allowing morphisms in a *biKleisli* category, of type $(D\ a \rightarrow T\ a)$, to be composed:

$$\textit{bibind}\ :(D\ a \rightarrow T\ a)\ \rightarrow T(D\ a) \rightarrow T(D\ a)$$
$$\textit{bibind}\ f = \textit{bind}\ (\textit{dist} \circ \textit{cobind}\ f)$$

Stencil functions of coKleisli arrow type are wrapped in an effect producing write operation to destructively update the grid structure, thus giving a biKleisli arrow which is applied using *bibind*.

The current implementation of Ypnos uses a slightly different structure to the monad and distributive law approach because the mutable array structures in Haskell cannot be given pure comonadic operations. Instead we use an *effectful comonad*-style structure. Unfortunately there is not time to discuss the issues here, but there is certainly future work in looking at how mutable data structures in functional programming, such as mutable arrays, can be described purely as a comonad, and then impurely by generating local monadic effects which are made global by a distributive law.

8. Related Work

There is a multitude of work on parallelisation of languages, both automatic and manual. We mention just a couple here.

The Chapel language, developed at Cray, has the similar aim to Ypnos, of separating problem and parallelisation code. Chapel can be used for stencil computations and distribution of arrays [4]. Arrays in Chapel are typed by a *domain*, a finite space of indices, and are accessed by a `forall` iterator which visits each element, similar to the run operation of Ypnos. Stencils are defined by offset tuples (akin to basis vectors) added to a current index tuple. Stencils can also be defined as an array of tuples which is applied with a `reduce` function to an array. The reduction-based approach conveys the intent of the stencil and the domains to the compiler for possible efficient computation, although there are no static guarantees of efficient execution.

The CAPTools toolkit for FORTRAN provides an *almost* automatic transformation tool for parallelising structured grids, featuring a powerful symbolic dependence analysis augmented by user interaction to attain a more accurate dependence graph [8]. Our own approach mitigates the need for interaction by communicating data dependencies statically via grid patterns.

Piponi noted the comonadic structure of structured grid-like computations in his superlative blog [24], giving a small example in Haskell of a one-dimensional cellular automata.

The Lucid dataflow language, already mentioned in Section 7.1, and the work of Uustalu and Vene, who described comonads as the "essence of dataflow" [28] inspired the comonadic structuring of Ypnos. Lucid is often described as an *intensional programming language*, or a *context-dependent language*. Lucid expressions can be understood as describing histories of computations, which can be modelled in the language semantics as streams. Special operations navigate forwards and backwards inside streams to access elements. The language looks and feels very different to Ypnos because in Lucid *everything* is implicitly a stream; in Ypnos, grids are explicit, and scalar non-grid expression coexist with grid expressions. Ypnos grids are however similar in flavour to the multidimensional streams of Multidimensional Lucid and its descendants [3], where operations that manipulate and navigate through streams are replaced with grid patterns. In Multidimensional Lucid, the current context of a stream can be accessed via the `#` operator, and the `@` operator provides random-access into a stream. These operators influenced our syntax in the index accessor pattern # in grid patterns, and `@` to denote which variable is bound to the current context.

Recent work by Lee, Chakravarty, Grover, and Keller targets GPU programming with an EDSL in Haskell that constrains the programmer to code suitable for GPU execution [16]. Their approach has some similarities to our own, using the strong static typing of Haskell to guide the programmer. In their EDSL, the types of operations are sufficiently restricted such that it is not possible to write code that would not be executable on a GPU. Their approach uses array types similar to our grids types, but stencil computation is not addressed in the same way. Our work is more general in terms of implementation, but is more specific in the computational pattern supported. There are a number of important issues addressed by their language which Ypnos does not address because it aims to be implementation agnostic. For example, array element types are constrained to those which can be efficiently handled by a GPU, and nested parallelism and recursion are disallowed. There is much that is similar in motivations of both our designs, each with slightly different focus. Such work could provide an implementational back-end for execution of Ypnos programs on GPUs.

9. Further Work

There are certainly many possible extensions to Ypnos. In the current implementation we plan to implement efficient back-ends for parallel GPU execution and parallelisation via domain-decomposition, and hope to have performance measures soon. Here we describe possible further extensions to the Ypnos language.

9.1 Mixing Sequential and Parallel Code

The current Ypnos EDSL implementation permits different back-ends, thus allowing sequential or parallel implementations of primitive operations. However, a compiler that always produces parallel implementations may produce inefficient solutions as some minor computations may be more efficient when implemented sequentially [21]. Thus, we hope to extend Ypnos to include both sequential and parallel primitives: `run` and `runSeq`, `iterate` and `iterateSeq`, etc. Thus the back-end requires modification, perhaps to be parameterisable by sequential and parallel *cobind* operations on the same grid structure, or parameterisable by some other structure representing distributed grids which interacts with the `Grid` comonad.

Furthermore, there are some useful programs that are currently difficult to get efficient executions of using Ypnos. One example: using parallel reductions during application of `iterateT`, which are not used as stop conditions. Currently such reductions cannot be made. Instead, such code must be written using `run`, thus the `iterateT` optimisations of mutable state cannot be used, and the persistence of subgrids on process-local memory during domain decomposition is lost. We hope to rectify this inflexibility soon.

9.2 Cost Model and Predictability

A cost model for execution is provided to the user by the static guarantee of optimisation by `iterate` and `iterateT`, and no optimisation by `run`; there are no unpredictable automatic analyses and transformations. However, there is no control over implementation properties such as the size of subgrid tiles, affecting execution time and memory requirements. We would like to extend Ypnos with *configuration variables*, a form of compiler directive, for expressing numerical parameters for decomposition and parallelisation.

9.3 Multi-scale Grids

Multigrid methods use different levels of discretisation (essentially different resolutions or scales), to speed up convergence by using coarse grained approximations of a grid to guide computation on fine grained versions of the grid [10]. The type of interaction between levels of discretised grids is often problem dependent.

Adaptive mesh refinements are a more general technique for speeding up convergence by approximating regions of a grid which have reached a fixed point, hence for which computation would be unnecessary [6]. Regions in equilibrium are *refined* by representation as a unit at a different scale to the rest of the grid, resulting in a grid with tessellating rectangles of different sizes (or resolutions).

Both techniques are valuable when performing intensive computations with large data sets, but it is not clear to us how either technique could be achieved in a language similar to Ypnos.

9.4 Vertex Shaders: the Formal Dual of Fragment Shaders

Earlier GPUs had a fixed execution pipeline, with computations largely split between two types of processors: *vertex* and *fragment* processors, handling respective *vertex* and *fragment* (or *pixel*) shaders. Vertex shaders are capable of random-access writes (*scatters*) but only perform single reads from the current stream position. Conversely, fragment shaders are capable of random-access reads (*gathers*) but can only perform single writes to the current stream position [12]. Fragment shaders correspond to the structured grid computations seen in this paper. Newer architectures have no differentiation between types of processor, offering many general processors that can handle either type of computation. [23]

It is conceivable that the scatter operations of vertex processors could be parallelised in the same way as structured grid operations assuming the domain of writes was sufficiently local so as to permit efficient execution under domain decomposition.

We have already described fragment shader computations, corresponding to stencil functions, as coKleisli arrows: $\mathsf{Grid}\ D\ a \to b$. However, vertex shader computations, which read a single value and produce several write operations, are more suitably typed: $a \to \mathsf{Grid}\ D\ b$, corresponding to *Kleisli* morphisms: the morphisms of a Kleisli category formed from a *monad*. Gather operations are therefore the dual of scatter operations, thus vertex computations are the dual of fragment computations, which can be structured by the dual of comonads: monads!

A larger language, perhaps *Ypnos++*, might have two classes of application primitives: the standard comonadic coextension operations (`run`, `iterate`, etc.) for stencil computations and monadic extension operations (maybe `runVertex`) for ***mould*** computations:

$$\mathsf{runVertex} :: (a \to \mathsf{Grid}\ D\ b) \to \mathsf{Grid}\ D\ a \to \mathsf{Grid}\ D\ b$$

Such a language could encompass a larger class of algorithms, allowing both stencil and mould functions to be expressed, with their interaction mediated by a comonadic/monadic grid data structure.

10. Conclusions

There are many different parallel programming patterns [2]. No single programming language can be suitable for programming all

such patterns. We have introduced the domain-specific language Ypnos, targeted at parallel structured grid programming.

Ypnos is restricted to the single application domain of structured grid programming by its primitive operations, its novel syntactic forms, and its embedding into a pure, functional language. All programs satisfy the *single, independent writes* property, thus correct optimised and parallel executions can be provided by its primitives. Furthermore, aggressive and unpredictable analyses and transformations are not required as data access patterns are encoded in *grid patterns*, which have a simple translation to compile-time data access information. Ypnos' restricted forms provide an easier approach to structured grid programming for those unfamiliar with (parallel) programming.

Ypnos employs category theoretic abstractions to organise computations. The Ypnos EDSL implementation can be instantiated with different instances of the underlying abstract structures. In this way, Ypnos is hardware- and implementation-agnostic, allowing different back-ends for different implementations, even code generation for further compilation.

Just as assembly programming can result in faster code than that produced by compilers, we expect that Ypnos code will often be slower than that programmed manually by an expert. The gain, however, is in ease of programming, low development times, and high portability.

We would be interested to see if other restricted syntactic forms, like grid patterns, could be applied in further domain-specific languages to facilitate parallelisation of other computational patterns.

In the late 1980s and early 90s there was much interest in applying dataflow languages to parallel architectures (such as Connections Machines). Besides the GLU (Granular Lucid) parallel programming system [15] the application of dataflow programming to parallel architectures never gained momentum. Now, 20 years later, there is a veritable renaissance in parallel architectures with multi-core general purpose CPUs abounding, and many-core machines now a commodity through GPUs. Ypnos is child of modern functional programming and multi-dimensional Lucid, targeting, as its dataflow ancestors were primed to target, parallel computation.

Acknowledgments

This work has been generously supported by an EPSRC DTA. Many thanks to Tom Schrijvers for various insights and help with paper, and to Marcelo Fiore for many interesting discussions.

References

[1] ACE Associated Compiler Experts bv. Parallelization using Polyhedral Analysis, 2008, last accessed September 2009. `https://www.opencosy.org/node/37`.

[2] K. Asanovic, R. Bodik, Demmel, et al. The Parallel Computing Laboratory at U.C. Berkeley: A research agenda based on the Berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

[3] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional programming*. Oxford University Press, Oxford, UK, 1995. ISBN 0-19-507597-8.

[4] R. F. Barret, P. C. Roth, and S. W. Poole. Finite Difference Stencils Implemented Using Chapel. Technical Report TM-2007/119, 2007.

[5] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting Polyhedral Loop Transformations to Work. Research Report RR-4902, INRIA, 2003.

[6] M. J. Berger. *Adaptive mesh refinement for hyperbolic partial differential equations*. PhD thesis, Stanford, CA, USA, 1982.

[7] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998. ISSN 1070-9924.

[8] E. W. Evans, S. P. Johnson, P. F. Leggett, and M. Cross. Automatic and effective multi-dimensional parallelisation of structured mesh based codes. *Parallel Comput.*, 26(6):677–703, 2000. ISSN 0167-8191.

[9] C. Gerald and P. Wheatley. *Applied Numerical Analysis*. Pearson Education, Addison Wesley, San Francisco, 2004.

[10] W. Hackbusch. *Multi-grid methods and applications*. Volume 4 of Springer series in computational mathematics. Springer, 1985.

[11] T. Halfhill. Parallel Processing with CUDA. *Microprocessor Report*, Januray 2008.

[12] M. Harris. Mapping computational concepts to GPUs. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM.

[13] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314, New York, NY, USA, 1985. ACM.

[14] G. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997. ISSN 0956-7968.

[15] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency: Practice and Experience*, 9(1):63–83, 1997.

[16] S. Lee, M. M. Chakravarty, V. Grover, and G. Keller. GPU Kernels as Data-Parallel Array Computations in Haskell. 2009.

[17] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report, 2001.

[18] G. Mainland. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5.

[19] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5.

[20] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1): 55–92, 1991. ISSN 0890-5401.

[21] N. Mukherjee and J. R. Gurd. A comparative analysis of four parallelisation schemes. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 278–285, New York, NY, USA, 1999. ACM. ISBN 1-58113-164-X.

[22] A. Munshi. OpenCL: Parallel computing on the GPU and CPU. presentation at SIGGRAPH 2008 `http://s08.idav.ucdavis.edu/munshi-opencl.pdf`.

[23] S. Patidar, S. Bhattacharjee, J. M. Singh, and P. J. Narayanan. Exploiting the Shader Model 4.0 Architecture, March 2007. Technical Report IIIT/TR/2007/145, 2007

[24] D. Piponi. Evaluating cellular automata is comonadic, December 2006, Last retrieved September 2009. `http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html`.

[25] J. Power and H. Watanabe. Combining a monad and a comonad. *Theor. Comput. Sci.*, 280(1-2):137–162, 2002. ISSN 0304-3975.

[26] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.

[27] D. Stewart. Domain Specific Languages for Domain Specific Problems. In *Workshop on Non-Traditional Programming Models for High-Performance Computing, LACSS*, 2009.

[28] T. Uustalu and V. Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, November 2006.

[29] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008. ISSN 1571-0661.

[30] W. Wadge and E. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-729650-6.

[31] P. Wadler. Linear Types Can Change the World! In *Programming Concepts and Methods*. North, 1990.