Ypnos : A functional programming language for structured grid computations

Max Bolingbroke [1] Dominic Orchard [2]

Abstract

Many applications in scientific computing, particularly in computational fluid dynamics, employ structured grid models of computation. It is common for simulated spatial environments to be discretised into a structured grid and decomposed for parallel computation, a task normally left to the programmer. We present a domain-specific functional language named Ypnos for expressing structured grid computation for parallel execution. Ypnos provides abstractions over arrays and index operations via an intensional view of values in which structured grids are presented as mappings from a multi-dimensional *context* to a value. Pattern matching in Ypnos is used to express the stencil of a computation thus facilitating static analysis of grid point access. This static analysis, along with the type system for dimensions, provides significant information to guide automatic domain decomposition, distribution, data allocation, and parallel execution of Ypnos programs. Our aim is that the language will be accessible to the scientific computing community and will aid development of parallel structured grid applications via the separation of a core algorithm from the details of parallelisation. This research paper focusses on the static semantics of Ypnos and its primary concepts.

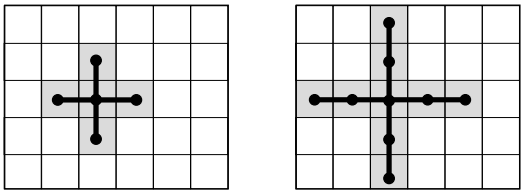
Key words: Structured grids, Domain-specific language, Stencil operations, Parallel programming, Computational Fluid Dynamics

1 Introduction

This paper presents Ypnos, a domain-specific language for expressing structured grid applications in scientific computing. Structured grid applications are typified by a discrete grid approximation of a real-world continuous space and the use of a *stencil* to describe grid access. Typical scientific computing domains using structured grid computations are computational fluid dynamics, weather modelling, and stress testing — although the technique is used in many areas.

[1] maximilian.bolingbroke@cl.cam.ac.uk
[2] dominic.orchard@cl.cam.ac.uk

The *stencil* of a computation defines a finite data access pattern and thus describes the data dependencies between *points* (elements) in the structured grid (see Fig. 1). The terms *stencil operation*, *stencil computation*, or *stencil code* are synonymous, referring to a grid operation over one or more grids based on a stencil whose result is written to a point in a grid.

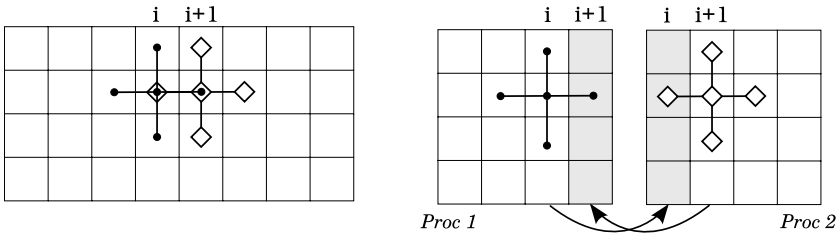


(a) (b)

Fig. 1. (a) 5-point stencil on a 2D grid. (b) 9-point stencil on a 2D grid.

Stencil operations are normally applied to each point in a grid; the result of a stencil operation at a grid point is used as that point's value in the next iteration of the stencil operation. A common stencil operation is the iterative approximation of partial differential equations.

Structured grid problems are traditionally parallelised by the programmer via manual *domain decomposition* in which the grid environment is partitioned into sub-grids and distributed to several processors for parallel execution. When data dependencies in a stencil operation cross boundaries of distributed sub-grids it is usual for data to be replicated at the boundaries of sub-grids and for updated points near the edges of the sub-grid to be communicated. Fig. 2 gives a diagrammatic example. Once an iteration of the computation has been performed on each sub-grid the updated values in column i are sent to *Proc. 2* and the updated values in column $i+1$ are sent to *Proc. 1*. The next iteration proceeds with the updated values.



(a) Before decomposition (b) After decomposition

Fig. 2. Domain decomposition example on a 2D-grid

Structured grid programming is a key computational pattern of high performance and desktop computing [1][2]. Faster, more efficient, structured grid applications allow more accurate simulation of the real world, thus testing can be improved and expensive real-world prototyping can be reduced. The domain decomposition technique for structured grid programming has been successfully employed for over a decade, with programmers writing decomposition and parallelisation code manually, often using libraries such as OpenMP.

However the manual parallelisation process is long, tedious, prone to errors (computationally and numerically), and hard to debug. The technique of domain decomposition is a prime candidate for languages and tools to target for the automatic parallelisation of structured grid applications. Recent advances in commodity parallel hardware (multi-core/many-core processors) have prompted further development of languages and compilers to facilitate parallel programming (see Section 5).

We present our own offering in the field of automatic parallelisation, Ypnos: a functional domain-specific programming language for expressing and automatically parallelising structured grid applications. Ypnos provides abstractions over classical array concepts found in imperative programming so that static analysis on grid operations is decidable, predictable, and reliable. This static analyses facilitates compile time decisions on memory layout, decomposition, and parallelisation. Thus greater time and memory efficiency can be achieved with program-specific memory layouts and distributions, which are automatically generated by the compiler. Furthermore it is our aim that Ypnos provides a means of structured grid programming for the scientific computing community whose semantics and operational behaviour is tractable both for the compiler and human readers. This motivates one of the prime design principles in the language – the performance of an Ypnos program should never depend on compiler transformations that are not explicitly represented in the source code in some way.

This paper primarily explores the abstractions Ypnos provides to aid humans, in expressing their ideas, and computers, in effective compilation for parallel architectures.

Section 2 discusses the abstract concepts in Ypnos. Section 3 describes the formal unsugared syntax and type system of Ypnos. Section 4 discusses the automatic decomposition and distribution of Ypnos grids with in terms of compile-time static analyses. Section 5 compares Ypnos with other work of similar motivation which is followed by the further work and conclusions in Section 6.

2 Ypnos Concepts

2.1 Grids and Dimensions

Structured grids are often programmed using mutable arrays in a language with random array access via data access constructs that are aware of memory layout e.g. C and Fortran arrays. Ypnos abstracts over arrays by providing a system for talking about values that reside within a *dimension* or multiple dimensions. Such a value represents a mapping from multi-dimensional contexts to values, not just a single value. Similarly a traditional array can be thought of as a mapping from indices to values.

The concept of mapping contexts to values has some correspondence to

intensional logic. Consider the natural language expression:

```
Today it is raining
```

If we evaluate this expression in London on a Monday morning it is probably true but if we evaluate this expression in California the result is probably false. Clearly the *context* of the expression's evaluation, in time and space, is important. An intensional expression maps from all possible contexts to values of the expression. Similarly in Ypnos by giving a value a dimensionally-indexed type a mapping from all possible contexts within those dimensions to values is created. In Ypnos we call these intensional entities, unsurprisingly, *grids*.

Dimensions are denoted in the language by single uppercase letters. A one-dimensional grid of integers, where the single dimension is the arbitrary X dimension would have type: $X\ int$. Multiple dimensions can be composed via a dimension product operation: $\times$. For example a two-dimensional grid of integers in X and Y dimensions would have type $X \times Y\ int$.

Ypnos dimensions are discrete, finite, uniform, and orthogonal to each other. Dimensions have an initial start context, the origin 0, and a last context, thus they are finite. The dimension product is a binary operation on the set of all dimensions with the following properties:

[D1] Commutativity: $X \times Y = Y \times X$

[D2] Associativity: $X \times (Y \times Z) = (X \times Y) \times Z$

[D3] Non-reflexivity: $X \times X$ is undefined, thus $\times$ is partial

[D4] Identity: $1 \times X = X$ where 1 is the zero-dimension [3]

A further construct relating to dimensions is a dimension vector which is a record mapping dimensions names to values with the syntax $< X = x_{val}, Y = y_{val}, \ldots >$ where X and Y are dimensions, and with type $< X \times Y \times \ldots > \tau$ for some type τ.

Grids are created via the `grid` built-in which is a function from a dimension vector specifying finite sizes of dimensions, and an initial grid point value, to a grid. For example

```
grid <X = 10, Y = 10> 0
```

creates a 10×10 two-dimensional array filled with 0s.

Returning to the notion of grids as a mapping from contexts to values, a context on a grid $A\ \alpha$ has type $@A\ \alpha$. A context can only be created by the built-in combinators `run`, `zipWith` and `foldT` - in particular it is not possible to return a context from a function.

[3] The 1 notation for a **zero**-dimension comes from the categorical notion of an initial object.

2.2 Stencils and Runs, Zips, and Reduces

Most structured grid computations in Ypnos can be specified with the `run` combinator which applies a stencil operation at every context within a grid. The `run` combinator has type:

$$\texttt{run} :: \forall A\ \alpha, \beta.(@A\ \alpha \to \beta) \to A\ \alpha \to A\ \beta$$

The `run` combinator is a higher-order function which takes as it's first argument a function of a context type on $A\,\alpha$ to a value of type β: the stencil operation. The stencil operation is applied at each context of a grid *A alpha* by `run` returning a new grid of type $A\ \beta$. Stencil operations pattern match on a context which binds values from the grid. Ypnos has two-dimensional and one-dimensional pattern match constructs, both of which can be nested within each other to form n-dimensional patterns. *These pattern matches define the stencil.* The following denotes a 5-point stencil operation that averages its surrounding points (note the use of the underscore wild-card character):

```
f :: @(X × Y int) -> double
f (X, Y): | _ t  _ | = l+t+r+b+c/5
          | l @c r |
          | _ b  _ |

somegrid' = run f somegrid
```

The stencil operation, `f`, has a context type on a grid of type `X × Y int`. The `@` syntax on the variable `c` in the pattern denotes that `c` is the variable which which to bind the value at the *current context*. The surrounding variable bindings in the pattern bind to values in the surrounding contexts of the grid. The C array equivalent would be that for indices `[i][j]` on an array `A`, variable `t` is `A[i-1][j]`, `l` is `A[i-1][j]`, `c` is `A[i][j]` etc. The syntax `(X, Y):` occuring before the pattern defines the matching order on the grid dimensions i.e. that the horizontal code direction in the pattern layout corresponds to the X dimension and vertical corresponds to Y. The result of the stencil operation is the value of the grid point in the new grid of type $X \times Y\ \beta$ at the current context of the stencil operation.

There is a similar one-dimensional pattern syntax:

```
f :: @(X int) -> int
f | @a b c d | = ....
```

Here the dimension ordering is unnecessary but can be optionally included. The current context annotation `@` can appear once at any position in the pattern, here appearing on the left of the pattern, thus `b`, `c`, and `d` are bound to the values three points "forward" (to the right) of the current context in the X dimension.

A `zipWith` combinator allows two grids of the same dimensionality to be "zipped" together via a binary operation at equal contexts in each grid. For example:

```
z :: @(X int) -> @(X int) -> int
z | l1 @c1 |  | l2 @c2| = (c1-l1)+(c2-l1)

g = zipWith (grid <X=3> 0) (grid <X=3> 1)
```

A `reduce` combinator is available that is similar to a fold over grids. For example the summation of every point in a grid `g` can be defined using a reduce:

```
plus ::(int -> int -> int)
plus x y = x+y

sum = reduce plus 0 g
```

Note how the function given to `reduce` is dimension invariant. However `reduce` can also be used to reduce over certain dimensions with a grid, thus it can perform *partial* grid reduction that reduces the dimensionality of a grid. E.g.

```
g = grid <X = 13, Y=37> 1

g' :: Y int
g' = reduce plus 0 g
```

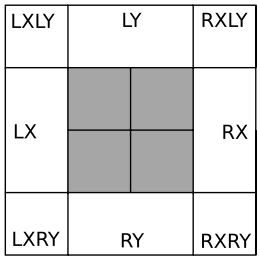
The type given to our result `g'` forces the reduce to operate on just the X dimension of the grid, thus the X row at each Y context is summed and returned in a Y dimensioned grid of 37 points.

2.3 Lifted Grids and Facets

Stencil operations performed at an edge of a grid may access points outside of the grid. Ypnos provides a "lifting" operation on finite grids to infinite grids to allow access outside of a grids finite bounds. Conceptually a lifted n-dimensional grid has contexts that range through $(-\infty, \infty)$ in each of the n dimensions. Lifting can be thought of as the embedding of a finite grid into an infinite grid. The lifted grid is associated with a *facets* structure which describes the boundary behaviour for each facet of the grid. An n-dimensional grid (an orthotope) has $3^n - 1$ facets (this excludes the middle cell). For example, a square has 4 edges and 4 vertices, and therefore 8 facets (Fig. 3(a)); for a cube there are 8 vertices, 12 edges, and 6 faces, therefore giving 26 facets; for tesseracts (4-cubes) there are 16 vertices, 32 edges, 24 faces, and 8 "cells" (3-faces), therefore 80 facets and so on. The `Facet D` structure has $3^n - 1$ boundary cases to which are attached values of type `Boundary D` τ for some τ ("boundary entities") which define the behaviour at these possible boundaries.

There are four possible boundary entities:

- `constantBoundary` - Contains a function from a vector of distances from the grid boundary in various dimensions to a value that should appear in that position.



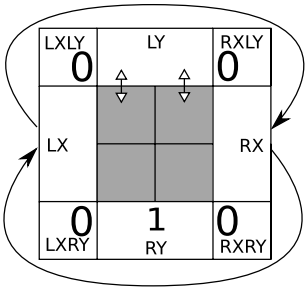
(a) Two-dimensional facets on an $X \times Y$ grid

(b) One-dimensional facets on an X grid

Fig. 3. Facets for a two-dimensional and one-dimensional grids

- `wrapBoundary` - Denotes a wrapping of this facet to the opposite facet on the other side of the orthotope, thus facilitating periodic grids and multi-toroidal shapes.
- `flippedWrapBoundary` - Denotes a flipped wrapping to the opposite facet i.e. Möbius strips, Klein bottles, Real projective planes, etc.
- `reflectBoundary` - Denotes a boundary where values are reflected at the edge which an additional adjustment function for defining energy loss.

Facets are constructed via the `facet` combinator which takes a record of facet labels to boundaries. Each dimension in a grid has a "left" boundary and a "right" boundary which are composed within the left boundaries and right boundaries of other dimensions – these can be enumerated to give the labels that should occur within the associated facets structure. One-dimensional grid facet labels as shown in Fig. 3(b) and two-dimensional grid facet labels are shown in Fig. 3(a). Fig. 4 gives an example of a possible facets structure for a two-dimensional grid.




```
facet {LXLY = constantBoundary (\_ -> 0)
       RXLY = constantBoundary (\_ -> 0)
       LXRY = constantBoundary (\_ -> 0)
       RXRY = constantBoundary (\_ -> 0)
       LX = wrapBoundary
       RX = wrapBoundary
       LY = reflectBoundary (\x -> x)
```

Fig. 4. Example two-dimensional facets on an $X \times Y$ grid.

2.4 Masks

It may be required that a computation be performed on subset of a grid. Alternatively it may be required that contexts are skipped with a set pattern e.g. skip every odd context, a pattern common in relaxation methods. This type of access is facilitated by *masks* and the `runWithMask` combinator. In keeping with the theme running throughout this work, Ypnos adopts a declarative

approach to specifying masks in order to increase the robustness of compiler analysis and transformation.

A mask entity is created given a record with fields of dimension vectors:

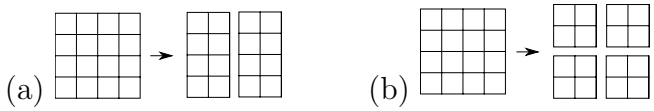
- *after* - Offsets after which the stencil operation should be applied
- *before* - Offsets before which the stencil operation should be applied
- *scale* - A number of points to skip between stencil applications

By supplying such a record to the built-in constructor function `mask` a value of type `Mask A` is obtained where the dimensional variable A statically ensures that the before, after, and scale vectors are consistent with each other. This mask is then used with the `runWithMask` combinator on grids of dimension A.

More details on the use and analysis of masks are given in Section 4.1.

2.5 Decomposition and Casts

A primary aim of Ypnos is to allow automatic parallelisation of structured grid applications via domain decomposition. An n-dimensional grid can be decomposed by partitioning some or all of its dimensional axes. For example a two-dimensional grid can be decomposed by one-dimensional decomposition (a) or by two-dimensional decomposition (b):



In manually parallelised structured grids the programmer must partition and distribute the data themselves. As the dimensionality of decomposition increases the difficulty of manual parallelisation grows sharply. Ypnos automates decomposition and distribution (see Section 4 for details) but still provides the programmer with some declarative control over the dimensionality of decomposition via *distributed dimension* types (inspired by, but somewhat different to, the distributed types of Keller [8]).

Distributed dimension types specify either strong hints to decompose on a particular dimension, keep a dimension localised on a single processor, or (in the absence of any annotation) to leave it up to the compiler. Distributed types are expressed via annotations on dimensions types where a dimension variable X is enclosed in brackets $[X]$ to denote decomposition and enclosed in banana brackets $(\!|X|\!)$ to denote localisation. The standard dimension types with annotation specify that the compiler can decompose the dimension as it deems efficient. Some examples are:

2D decomposition of 2D	Distribute X and Y	[X] × [Y]
1D decomposition of 2D	Distribute X and localise Y	[X] × $(\!\|Y\|\!)$
2D decomposition on X, Y of 3D	Distribute X, Y and localise Z	[X] × [Y] × $(\!\|Y\|\!)$
At least 1D decomposition of 2D	Distribute X, leave Y to compiler	[X] × Y

Redistribution is expressed by explicit type-casts to a possibly distributed dimension grid type, which gives operational information on the type of decomposition given appropriate parallel resources. Changing the distribution of a grid can be very costly thus type-casts are explicit in order to promote awareness of the operational cost of distribution and redistribution of a grid.

Casts are performed with the `distributionCast` combinator which the programmer explicitly types with an inline type signature. For example:

```
A' = (distributionCast :: [X] × Y int -> X × [Y] int) A
```

casts a grid of integers where X is distributed to one where Y is distributed.

We discuss decomposition further in Section 4.

2.6 Mutability

An important feature of imperative programming is mutability. The lack of mutability in functional languages means that arrays must be created and copied throughout execution as opposed to updating arrays in memory. This is known as the aggregate update problem [7]. Being able to update array elements in place not only reduces memory requirements but also reduces execution time spent in allocation and deallocation. While a static analysis of live variables and data dependency may be able to determine where and when array updates can be made this has the potential to change the efficiency of the program drastically "behind the back" of the programmer, which is contrary to our key philosophy of a human-tractable cost model for the language.

Ypnos solves this issue with a special combinator `foldT` that embeds a grid into a new dimension: time. The time dimension is created and handled by `foldT` and allows Ypnos grids to use standard context access methods on temporal views of itself. The type of `foldT` is:

$$\texttt{foldT} : \forall A\ \alpha.(@T\ (A\ a) \to \texttt{Continue}\ A\ a) \to A\ a \to A\ a$$

The `foldT` combinator creates a time dimension `T` in which it embeds a grid of type `A a` (note this is not a product of dimensions $T \times A$ but is a nested grid of grids). The first argument of `foldT` is a function from a `T (A` α`)` context to a `Continue A` α type. This type is analogous to a `Maybe` α type in Haskell, and is constructed by either the expression `proceed a` (where a has a type that instantiates α) or by `terminate` (which can instantiate α with any type necessary). This serves to indicate to `foldT` whether to continue or not at each recursive step.

The function given to `foldT` pattern matches on contexts in the time dimension allowing the user to specify temporal accesses on a grid i.e. the grid in the previous iteration, the grid in the previous previous iteration and so on. Thus, a pattern match over n consecutive locations defines a *history stencil* on a grid which looks n places into the past.

The combinator `foldT` holds in memory enough copies of the grid to satisfy

this history stencil. When grid allocations can no longer be accessed by the history stencil the memory becomes unused and thus the `foldT` combinator may overwrite its own allocations safely. This provides a controlled mutability that is performed for the programmer by `foldT` but which is made explicit in the program text by the use of `foldT`.

Furthermore, `foldT` allocates storage for all intermediate grid allocations caused by grid operations such as `run`, `zipWtih`, and `reduce` within the computation being folded. Each intermediate allocation in this computation allocates into memory space which is cyclically reused as `foldT` iterates forward through time. Fig. 5 demonstrates this model of `foldT` allocations in which the grey block denotes the main grid, g, and the white blocks represent intermediate grids caused by other grid operations. The arrows denote updates to memory. When the end of the space is reached at context t the updates wrap to the start of the memory block, overwriting the now unused memory.

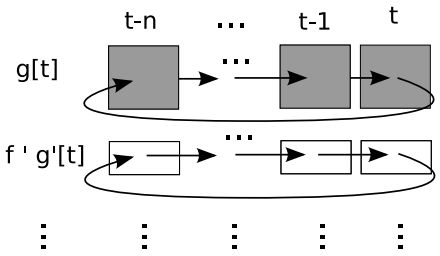


Fig. 5. `foldT` stores n allocations of all grids and intermediate grids, reusing memory once a temporal context can no longer be matched by the history stencil of the function supplied to `foldT`.

Here is an example using `foldT`:

```
n-order-derivative :: (@X × Y int) -> (@X × Y int) -> int
n-order-derivative | _ t1 _  | | _  t2 _ | = (t2-t1)+(l2-l1)+(b2-b1)+(r2-r1)
                   | l1 _ r1 | | l2 _ r2 |
                   | _ b1 _  | | _  b2 _ |


program :: (@T (X × Y int) -> Continue (X × Y int))
program T:| g2 g1 @_ | = let  converge = reduce g1 (λx -> x + x)
                         in   if converge==0
                                 terminate
                              else
                                 proceed (zipWith n-order-derivative g2 g1)

g' = foldT program g
```

in which we successively compute derivatives between grids of $X \times Y$ based on the previous two grids.

3 Syntax and Type System

Fig. 6 gives the unsugared syntax for Ypnos programs – explanations for the various constructs are given inline throughout this paper, particularly in Section 2.

The type system for the language (see Fig. 8 and Fig. 9) is mostly standard, and is based on that for Standard ML [10], albeit without mutable references or the complications that follow from them. However, there are some complications particular to Ypnos that it may be instructive to discuss.

Firstly, the rules for pattern matching (Fig. 10) must ensure that:

- All the subcomponents of the pattern match on the same set of dimensions
- There is precisely one *current context* point (e.g. `@x`) in a pattern match
- Patterns that need a context get a type that reflects that requirement (i.e. a grid type containing `@`)

Secondly, we introduce a notion of subtyping (Fig. 11) to allow conversions that should be costless at runtime to occur with an explicit (and inconvenient) cast. In particular, this allows treating grids with locality and distribution annotations as grids without any such annotations for some purposes.

Thirdly, we have to introduce built-in type rules for the `distributionCast` and `facets` primitives that do not have parametrically polymorphic types. In particular, `distributionCast` only allows changing distribution and locality annotations on some grid, and `facets` must ensure that the record it is given as an argument has the complex structure of facet labels to define a `Facets` record. We omit the complete rule system for `facets`, but Section 2.3 should provide an intuitive sense of what would be required here.

The astute reader will have noticed that although we predicate the effectiveness of our language on the ability of the compiler to carry out static analysis, we model masks and facet information in the language as arbitrary expression types which might contain non-terminating values, or undecidable expressions. In this paper we assume that the compiler rejects programs where facet or mask information arriving as an argument to one of the primitives `run`, `zipWith`, or `lift` is not resolvable as a compile-time constant. Note that we do *not* extend this restriction to size information such as that arriving as an argument to `grid`, although adding such a restriction is not out of the question and would help the precision of at least the analysis outlined in Section 4.2.

4 Decomposition and distribution

The use of multidimensional pattern matches encoding the stencil of a computation provides a principled source of information about the data dependencies of a computation. The data dependence information allows the compiler to implement a number of interesting optimisations, but more importantly it

In the following α denotes type variables, δ denotes dimension names, $x, y, z, \ldots$ denote variables, and l denotes record labels.

$$
\begin{array}{rll}
\mathit{prog} ::= & \overline{\texttt{dimension}\ \delta}\ \overline{x\ \texttt{=}\ e} & \text{Programs} \\
p ::= & x \mid @x \mid \_ \mid @\_ \mid (\delta)\texttt{:}\,|p \ldots p| \mid (\delta,\ \delta)\texttt{:} \begin{vmatrix} p & \cdots & p \\ \vdots & \ddots & \vdots \\ p & \cdots & p \end{vmatrix} & \text{Patterns} \\
e ::= & x \mid \mathbb{R} \mid \mathbb{Z} \mid \mathbb{B} \mid e \oplus e \mid \left\{\overline{l\ \texttt{=}\ e}\right\} \mid \left\langle\overline{\delta\ \texttt{=}\ e}\right\rangle \mid & \\
& \lambda p.e \mid e\ e \mid \mathit{built\text{-}ins} \mid \texttt{distributionCast} \mid & \\
& \texttt{facets} \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{Expressions} \\
\mathit{built\text{-}ins} ::= & \texttt{run} \mid \texttt{runWithMask} \mid \texttt{zipWith} \mid \texttt{foldT} \mid \texttt{grid} \mid \texttt{reduce} \mid & \\
& \texttt{lift} \mid \texttt{unlift} \mid \texttt{mask} \mid \texttt{constantBoundary} \mid & \\
& \texttt{wrapBoundary} \mid \texttt{flippedWrapBoundary} \mid & \\
& \texttt{reflectBoundary} \mid \texttt{proceed} \mid \texttt{terminate} & \text{Simply-typed built-ins} \\
\sigma ::= & \forall\overline{\delta}\ \overline{\alpha}.\tau^{+} & \text{Quantified types} \\
\tau^{-} ::= & \tau^{+} \mid @\, d^{top}\ \tau^{+} & \text{Negative types} \\
\tau^{+} ::= & \alpha \mid \tau^{prim} \mid d^{top}\ \tau^{+} \mid \left\{\overline{l\ \texttt{:}\ \tau}\right\} \mid \left\langle d^{top}\right\rangle \tau^{+} \mid \tau^{-} \to \tau^{+} & \text{Positive types} \\
\tau^{prim} ::= & \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid \texttt{Facet}\ d^{top}\ \tau^{+} \mid & \\
& \texttt{Boundary}\ d^{top}\ \tau^{+} \mid \texttt{Mask}\ d^{top} \mid \texttt{Continue}\ \tau^{+} & \text{Primitive types} \\
d^{top} ::= & d^{\infty} \mid d & \text{Dimension types} \\
d ::= & \delta \mid (\!|\delta|\!) \mid [\delta] \mid d \times d &
\end{array}
$$

Note that $\oplus$ denotes any standard built-in primitive binary operations.

Fig. 6. The unsugared syntax for Ypnos.

means that the programmer can *rely* on those optimisations taking place, as the result of a dependence analysis is encoded directly in the program.

The following discusses some possible transformations that could be performed automatically on Ypnos programs by the compiler:

- *Automatic domain decomposition for distributed parallelism.* Stencil dependency information allows the compiler to find sensible points at which to split a grid over several processors, and it also allows it to determine the amount of information that must be exchanged between processors at each time step. For example, a 2D stencil that only accesses its immediate neighbours means that only one row of data around a join point between two processors must be exchanged every time step (as in Fig. 2), whereas if the stencil looks further into its neighbouring data items then more rows would

be sent. By replacing hand-generated distribution code with code generated by a reliable compile-time transformation, programs becomes more idiomatic while simultaneously encouraging target agnosticism.

- *Automatic optimisation of the trade-off between computation and communication.* The relative latency of communicating elements of a grid between processors versus recomputing them independently will vary depending on the target architecture Ypnos is being compiled for – on a SMP CPU sharing grids is relatively cheap (through the medium of shared memory), whereas on a distributed system with high communication costs recomputation may be strongly preferred. What is more, the structured grid paradigm gives a natural way to make this trade-off. During domain decomposition a larger or smaller overlap region between sub-grids affects latency – the more overlap, the greater the latency tolerance of the computation but the more cycles are "wasted" recomputing results that are already known by other processors. Statically available stencil shape information, combined with some description of the target hardware characteristics, means that Ypnos is well placed to perform this optimisation for the programmer in a reliable way.
- *Automatic cache management.* It is well known that reductions in memory latency have failed to keep step with the pace of increases in instruction execution frequency on modern machines. This means that even on a single-processor effective use of fast cache memory is essential for high performance. Because Ypnos abstracts away from indexes as a memory access method, it is able to perform cache optimisations such as permuting the order in which dimensions are stored in linear memory as is appropriate for the particular computation under consideration. Another possibility is for Ypnos to automatically generate hints in the instruction stream that tell the CPU which memory locations will be demanded next so it may pre-fetch if appropriate, where such hardware support exists.

The rest of this section will discuss one of the possibilities offered by static analysis in detail – the potential to improve storage reuse in the program by updating grids in place (Section 4.2). In order to do this we also need to discuss in detail the form that masks take within the Ypnos runtime system (Section 4.1).

4.1 Representation of execution masks

A mask in used with the `runWithMask` combinator to operate on a subset of a grid. The *iteration space* of a run is the space of indices iterated over by the implementation: $\mathbb{N}^n$ for an n-dimensional grid. The hidden internal representation of masks is a type of affine transformation from indices in the iteration space to indices on a grid. The mask applied to the iteration space gives an *execution mask* of where to perform stencil operations on the grid.

Consider the execution mask shown on the left of Fig. 7. A mask matrix $\boldsymbol{M}$ maps points from a two-dimensional iteration space to the execution mark

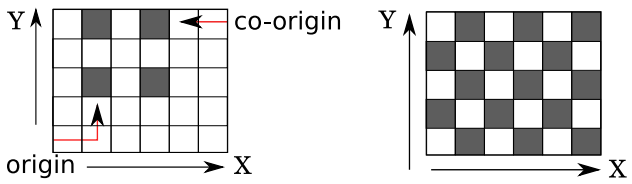


Fig. 7. (Left) Execution mask defined by an *after* offset of 2 from the origin in the Y dimension and 1 in X, a scale factor of 2 in both X and Y, and a *before* offset of 2 from the co-origin in X (Right) Example checkerboard mask formed by disjunction of two scale factor 2 masks at different offsets.

shaded in black. The matrix consists of three components: $\boldsymbol{S}$, a diagonal matrix of scale factors, and two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ that say respectively how many points *after* and *before* the boundaries of the finite grid execution will be allowed in. $\boldsymbol{a}$ defines offsets from the origin and $\boldsymbol{b}$ defines offsets from the *co-origin* — that is, the maximal point in a finite n-dimensional space. Thus together they determine the *finite extent* of an execution mask. For our example, we have the following:

$$\boldsymbol{S} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \qquad \boldsymbol{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

The mask matrix $\boldsymbol{M}$ combines this information so that it can be composed with other mask matrices by matrix multiplication (*scale factors* are multiplied and *after* and *before* indices added after undergoing the appropriate scaling), though this fact is as-yet unexploited by the language.

$$M = \begin{bmatrix} \boldsymbol{S} & \mathbf{0} & \boldsymbol{a} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{S} & \mathbf{0} & \boldsymbol{b} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \begin{bmatrix} \boldsymbol{g} \\ \boldsymbol{b} \\ 1 \\ 1 \end{bmatrix} = \boldsymbol{M} \begin{bmatrix} \boldsymbol{i} \\ \mathbf{0} \\ 1 \\ 1 \end{bmatrix}$$

The implementation of `run` takes indices $\boldsymbol{i}$ from an iteration space $\mathbb{N}^n$ and translates them into the space of indices in the grid ($\boldsymbol{g} \in \mathbb{N}^n$), along with the final co-origin $\boldsymbol{b}$ by the matrix-vector multiplication illustrated above and to the right.

Points in $\boldsymbol{g}$ thus indicate where in a grid the stencil operation being `run` should be applied, subject to two constraints: that $\boldsymbol{g}$ must be within the finite extents of the grid and $\boldsymbol{g}$ must not index a point in the grid that is excluded by $\boldsymbol{b}$.

Masks may also be combined into disjunctions of several masks. This is necessary to model some common execution masks that occur in practice, such as the "checkerboard" pattern illustrated in Fig. 7. Such *mask disjuncts* are represented concretely by sets of mask matrices. Operationally, the `runWithMask` combinator supplied with such a disjunct applies the stencil

operation at a point iff that point is mapped to by any execution masks from the set of the disjunct of masks.

Disjunct masks are the only form of mask the user is presently able to manipulate in the language. This is due to the fact that, unlike mask matrices, it is not clear what it means to compose two disjuncts – instead, they are only subject to an "or" operator. We felt it would be confusing to reflect this disparity in the source language by having two mask types with different operations on each, and so opted to expose only the strictly-more-powerful disjuncts to the user.

4.2 Stencil independence analysis

4.2.1 Motivation

Given a stencil operation function `f` whose data dependencies have been statically analysed by inspecting the relevant pattern we may be interested in answering the question "given the execution mask, do any of the stencils read from locations potentially already written by another stencil"? We call this property *mutual execution independence*.

If the user's program includes an application of `run` on an appropriate computation, grid, and mask such that the mutual execution independence property holds, then it might be possible to transform the program so that instead of somehow finding some new memory to store the updated array the update is simply performed in place. However, this is undesirable as it is not in keeping with the philosophy that the language should have a tractable cost model for the human reader (as mentioned in Section 5). It would be possible for users to inadvertently write programs whose efficient execution depends on in-place update occurring. Performance may then degrade heavily upon a user making some seemingly harmless change that breaks one of the fragile properties that this transformation depends upon.

These is a possible loophole in this argument however: what if we introduced variants of `foldT` that promised to do their work precisely with a fixed number of copies of the array? For example, we could have a `foldT1` that promised an in-place update, and, if during compilation the compiler was unable to prove that for a particular use of `foldT1` this was not possible, it would signal a compile error. This gives some of the benefits of static analysis and automatic program transformation without any of the pain of unpredictable performance, as the expected analysis results are encoded in the program text.

Interestingly, this means that the compiler must ignore information it has discovered that would allow it to make decisions that are "too optimal". If a program made use of the `foldT3` function that promised to use exactly three copies of the array to do its work, then even if the compiler discovers during the course of proving that this it is possible for the operation to be done in place, this information *should* be thrown away and the version of the code that requires three array copies used anyway (though a warning about this

action might usefully be emitted). This prevents users from becoming reliant on results of program analysis that are not explicitly accounted for in the source code.

In order to detect the calls to a `run`-like combinator where in-place update can be used as an alternative to array copying, we need two pieces of information:

i. That for this particular instance of a call to `run` mutual execution independence holds

ii. That the array we intend to do the in-place update on is never read again after `run` has terminated (i.e. `run` consumes the array linearly)

The truth of point i can be determined by performing a static analysis with information on stencil shape and grid setup – this process is described in Section 4.2.2.

The truth of point ii can be established either by a linearity analysis (or similar type system extension, such as with quasi-linear types [9]), or by making use of monads [15] to ensure that updates to a mutable array for grid values are accounted for in the program. Together these proofs enable the in-place update transformation to be safely done by a compiler in appropriate places (i.e. in uses of `foldT1`).

4.2.2 Implementation

In order to carry out the static analysis to determine in-place update we need two principal pieces of information: the stencil of the computation for the `run`, and the mask disjunct supplied to the `run`. We consider masks to be infinitary, so the vector $\boldsymbol{b}$ of "before" indices (that specifies where the mask should stop, relative to the co-origin) does not concern us. Furthermore, for the purposes of this analysis, stencils should be described by sets of n dimensional vectors that each specify an offset from the "write" point of a grid element update operation to a "read" point. For example, the 5-point stencil over a 2D grid shown in Fig. 1 would be described by the following set of vectors:

$$T = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\}$$

We can now formulate a criterion for deciding whether a mask disjunct satisfies the mutual execution independence property by testing whether for all pairs of masks M_1 and M_2 in the disjunct the following holds:

$$\forall \boldsymbol{o}_1 \in T, \boldsymbol{o}_2 \in T. \forall \boldsymbol{i}_1 \in \mathbb{Z}^n, \boldsymbol{i}_2 \in \mathbb{Z}^n. \boldsymbol{a}_1 + \boldsymbol{S}_1 \boldsymbol{i}_1 + \boldsymbol{o}_1 \neq \boldsymbol{a}_2 + \boldsymbol{S}_2 \boldsymbol{i}_2 + \boldsymbol{o}_2$$

Because the matrices S are diagonal we can test this condition by checking that it holds element-wise for all of the n elements of each vector:

$$\forall \boldsymbol{o}_1 \in T, \boldsymbol{o}_2 \in T. \forall \boldsymbol{i}_1 \in \mathbb{Z}^n, \boldsymbol{i}_2 \in \mathbb{Z}^n. \forall j \in \mathbb{Z}_n. \boldsymbol{a}_{1j} + \boldsymbol{S}_{1jj} \boldsymbol{i}_{1j} + \boldsymbol{o}_{1j} \neq \boldsymbol{a}_{2j} + \boldsymbol{S}_{2jj} \boldsymbol{i}_{2j} + \boldsymbol{o}_{2j}$$

The arithmetic expression can be arranged to become an instance of Bézout's identity $ax + by = d$ which has integer solutions iff $gcd(a, b) \mid d$, and hence the condition holds iff:

$$\forall \boldsymbol{o}_1 \in T, \boldsymbol{o}_2 \in T. \forall j \in \mathbb{Z}_n. gcd(S_{1jj}, \boldsymbol{S}_{2jj}) | \boldsymbol{a}_{2j} - \boldsymbol{a}_{1j} + \boldsymbol{o}_{2j} - \boldsymbol{o}_{ij}$$

It would be possible to refine this result given information about the bounds of the grid being considered as some of the conflicts detected by this approach may occur in regions of execution mask interference that are not present in the finite region of the grid actually under consideration. It remains to be seen if such an extension would be useful in practice.

A further consideration is the effect of wrapping. If the grid being `run` on makes use of any wrapping facets then this analysis does not apply, as the stencils which reach over the wrapped boundary may or may not interfere with stencils co-occurring with them in that fraction of the grid, depending on the realised finite lengths of each grid dimension of the grid. Boundaries making use of reflection pose a similar problem. This could be resolved by making grid dimensions known statically, as discussed in Section 3.

5 Related Work

The CAPTools toolkit for Fortran [6] provides an *almost* automatic transformation tool from scalar structured grid problems to a parallelised form, motivated by the need for quicker and easier parallelisation of scalar code. CAPTools offers interactive parallelisation tools as an alternative to automatically parallelising compilers as "the core of any parallelising compiler is the dependence analysis and many decisions in the analysis require knowledge of program variables, about which no information is explicitly available. This results in the conservative assumption of data dependencies, which could influence adversely the quality of the consequent parallel code." [6]. Our initial desideratum at the outset of the Ypnos project was that a language should be sufficiently expressive to provide all information required for automatic parallelisation, and that this could be realised via abstractions and restrictions to allow static analysis.

Chapel is an imperative parallel programming language being developed by Cray as a language for easily expressing algorithms separate from details of parallelisation [5]. Chapel can be used for stencil computations and distribution of arrays for structured grid programming [4]. Arrays in Chapel are typed by a *domain*, a finite space of indices. Arrays can be access by a `forall` iterator which visits each element. Stencils can be 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. Domains can be

specified as distributed domains with abstract distribution concepts: block, cyclic, block-cyclic, or cut. Different dimensionality domain decompositions can not be expressed. Chapel's separation of algorithm from parallelisation is also striven for in Ypnos. Ypnos provides more static information to its parallelisation process for efficient execution.

The dataflow programming language Lucid introduced intensional concepts in order to express iteration declaratively via an implicit time dimension [14]. Intensional expression in Lucid are represented by infinite streams that map integer contexts to values. Lucid evolved from its original conception in the 1970s and increasingly dealt with dimensions. Multidimensional Lucid followed original Lucid where intensional expressions could be written over arbitrary orthogonal dimensions [3].

In Multidimensional Lucid functions and operators can be parametrised by a dimension which is suffixed to the function name. For example the following adds two streams `a` and `b` at contexts in the dimension `x`: `c = a +.x b`. Contextual values of an expression can be accessed at any context via the `@` operator which takes an integer context. The current context of a stream can be accessed via the `#` operator. In Multidimensional Lucid a variable is a mapping from multiple *single* dimension contexts to values, as opposed to multiple *multidimensional* contexts.

Later, Field Lucid appeared with multidimensional contexts and corresponding intensional operators for access to multidimensional streams. However Field Lucid was restricted in that dimensions were preordained and functions were not polymorphic in their dimensionality. Indexical Lucid followed which improved upon Field Lucid's multidimensional abstractions allowing expressions to vary in multiple dimensions at once.

The current Lucid descendant is TransLucid in which dimensions are denoted by integers (e.g. the dimension 0, dimension 1, etc.). TransLucid has notation called explicit tuples which correspond to multidimensional Cartesian coordinates. Explicit tuples can be used for specifying new contexts relative to the current for access to infinite multidimensional arrays [11].

The intensional concepts in Lucid are similar to Ypnos' intensional view of a grid. Lucid, and its descendants, handle dimensions explicitly, as does Ypnos, although in Ypnos typing is stronger and dimensions occur at a type level concept. The earlier Lucid work operating in an implicit time dimension to express declarative iteration inspired the `foldT` combinator. The @ context syntax in Ypnos types is also inspired by Multidimensional Lucid's @ operator, although in Ypnos the @ is a type level denotation of a binding of a context to a value, whereas in Lucid the @ operator allows random access. TransLucid provides multidimensional arrays that are similar to grids, and explicit tuples of multidimensional coordinates which are similar to Ypnos' dimension vectors, although Ypnos dimension vectors are not used for random array access.

Single Assignment C (SAC) is a reduced functional variant of C for efficient

functional array operations [12]. Arrays are represented by a flat data vector and a shape vector defining the array structure and dimensions. SAC allows specification of shape invariant array operations that have constant time random access on arrays. As in APL and Fortran90, SAC provides array operations which apply uniformly to all array elements for managing, generating, and modifying arrays. SAC has a central language construct called a WITH-loop for user-defined operations on each element of an array – equivalent to the Ypnos `run` combinator.

Although arrays are pure in SAC they are implemented efficiently via WITH-loop folding (cf. loop fusion) based upon the well known map equation: `map` $f \circ$`map` $g \equiv$ `map` $(f \circ g)$ and a kind of linearity analysis implemented with reference counting. The types of arrays are specialized as far as possible at compile time in order to discover shape information (dimensionality and array sizes) which aids this process.

SAC is not a parallelising compiler and has a restricted language set, but is very fast for array computation. Its WITH-loops are akin to Ypnos `run` combinators. Ypnos pattern stencils are a further abstraction from the restricted indexing schemes in SAC .

6 Further Work and Conclusions

6.1 Further work

Ypnos is still in its early stages of development and there are a multitude of possible extensions to the current work. The language has many areas that need further exploration.

One obvious area where more work is required is in further development of the analyses described in Section 4 and in testing their effectiveness empirically with an implementation of the language and some structured grid benchmarks. Comparison to other related languages and tools would be valuable.

Another avenue for further work might be to use the concept of stages as used in MetaML [13] to cleanly express the split between compile-time and run-time information in the language, which was informally described in Section 3. This boundary must be explicit in the programming model because static information is essential in order for many of the optimisations Ypnos hopes to perform. It might also make sense to distinguish between two sorts of runtime – imagine a GPGPU multiprocessor machine where we can distinguish between runtime on the host CPU and on the associated GPU processors. The host would then able to generate code on the fly specialised on particular, dynamically available, grid information (e.g. the dimensions) which is suitable for execution by the GPU processors.

Another extension is support for adaptive mesh techniques. Adaptive meshes present grids whose resolution and thus accuracy can be changed for arbitrary regions of a grid during computation. Thus quiescent regions of a

grid can be operated over at a lower resolution compared to more active regions. Whilst Ypnos supports nested grids there is no simple way to increase a the resolution of a region by duplicating the data into a finer grained grid and packing it into the previous space. Exploring a declarative expression of such meshes could yield interesting results.

In this paper we have not discussed input and output. Clearly I/O is required for programs to be useful. Our view is that Ypnos, as it is a domain-specific language, should be embedded within another language that has greater expressivity in terms of I/O so that arrays can be loaded and saved from files for real programs. This avoids contaminating the functional purity of Ypnos, but the details of this embedding have yet to be explored.

6.2 Conclusions

This has paper has explored some of the concepts in Ypnos, the purpose of which is to allow the expression of principled structured grid programs and automatic distribution and parallelisation of those programs. One of the primary design philosophies of Ypnos is that the performance of a compiled Ypnos program should never depend on an unseen compiler transformation that is not explicitly represented in the source code, thus we attempt make clear the behaviour of execution through its abstractions.

Dimensions and grids abstract over arrays, context based pattern matches abstract over stencils for array access. The `run`, `reduce`, and `zipWith` operations abstract over iterations on arrays. Facets and lifted grids abstract over boundary cases on grids. Declarative masks are supported for standard techniques such as checkerboard tiling of a grid. Domain decomposition information is provided declaratively by the programmer through typing of dimensions. Finally efficiency through re-use of allocated memory is considered and provided by lifting grids into an extra dimension of time within the `foldT` combinator. The `foldT` combinator allows destructive writes to the memory in an explicit, but managed way.

We have also discussed how Ypnos programs can be effectively decomposed and distributed, with particular reference to static analyses on such programs.

Ypnos is intended as a prototype that contributes to the discussion about how novel language abstractions can be used to provide simple, compositional constructs to aid programmers in expressing their ideas while simultaneously aiding compilers in producing efficient, parallel implementations of programs.

Acknowledgements

The authors would like to thank Microsoft Research and the EPSRC for their monetary assistance in undertaking this research. Thanks are also due to Alan Mycroft for reading early drafts of this paper and for hours of productive discussion. Thanks are also due to Robin Message for his thoughts and reading

of drafts.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Krste Asanovic, Ras 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.

[4] Richard F. Barret, Philip C. Roth, and Styephen W. Poole. Finite difference stencils implemented using Chapel. Technical Report TM-2007/119, 2007.

[5] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[6] 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.

[7] Paul Hudak and Adrienne 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.

[8] Gabriele Keller. Transformation-based implementation of nested data parallelism for distributed memory machines, 1999.

[9] Naoki Kobayashi. Quasi-linear types. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42, New York, NY, USA, 1999. ACM.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[11] John Plaice, Blanca Mancilla, and Gabriel Ditu. From lucid to translucid: Iteration, dataflow, intensional and cartesian programming. *Mathematics in Computer Science*, 2008.

[12] Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[13] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM.

[14] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[15] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.

$$\boxed{\tau \prec \sigma}$$

$$\text{SUB}\ \frac{}{\tau\,[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\,[d_1/\delta_1, \ldots, d_m/\delta_m] \prec \forall\ \delta_1, \ldots, \delta_m\ \alpha_1, \ldots, \alpha_n.\tau}$$

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\text{SIGMA}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall(\text{fdn}(\tau) \setminus \text{fdn}(\Gamma))\ (\text{ftv}(\tau) \setminus \text{ftv}(\Gamma)).\tau}$$

$$\boxed{\Gamma \vdash e : \tau \text{ (selected rules)}}$$

$$\text{POLY-VAR}\ \frac{\Gamma(x) = \sigma \qquad \tau \prec \sigma}{\Gamma \vdash x : \tau} \qquad \text{SUB}\ \frac{\Gamma \vdash e : \tau \qquad \tau \overset{\tau}{<:} \tau'}{\Gamma \vdash e : \tau'}$$

$$\text{RECORD}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 = \tau_1, \ldots, l_n = \tau_n\}}$$

$$\text{VECTOR}\ \frac{\Gamma \vdash e_1 : \tau \qquad \cdots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash \langle \delta_1 = e_1, \ldots, \delta_n = e_n \rangle : \langle \prod_{i=1}^{n} \delta_i \rangle\, \tau}$$

$$\text{LAMBDA}\ \frac{\overset{pat}{\vdash}\ p : \Gamma', \tau_1 \qquad \Gamma \cup \Gamma' \vdash e : \tau_2}{\Gamma \vdash \lambda p.e : \tau_1 \to \tau_2} \qquad \text{BUILT-INS}\ \frac{\sigma = \kappa(\textit{built-in}) \qquad \tau \prec \sigma}{\Gamma \vdash \textit{built-in} : \tau}$$

$$\text{LETREC}\ \frac{\Gamma \cup \{x \rightarrowtail \sigma_1\} \vdash e_1 : \sigma_1 \qquad \Gamma \cup \{x \rightarrowtail \sigma_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2}$$

$$\text{CAST}\ \frac{D\ \tau \overset{\tau}{<:} D'\ \tau \qquad D''\ \tau \overset{\tau}{<:} D'\ \tau}{\Gamma \vdash \textit{distributionCast} : D\ \tau \to D''\ \tau}$$

$$\text{FACETS}\ \frac{A = \text{field-names-to-facet-dimensions}(l_1, \tau_1, \ldots, l_n, \tau_n)}{\Gamma \vdash \textit{facets} : \{l_1 = \tau_1, \ldots, l_n = \tau_n\} \to \texttt{Facet}\ A\ \tau}$$

Fig. 8. Type rules for Ypnos

$$
\begin{aligned}
\kappa(\texttt{run}) &= \forall A\ \alpha, \beta.(@A\ \alpha \to \beta) \to A^{\infty}\ \alpha \to A^{\infty}\ \beta \\
\kappa(\texttt{runWithMask}) &= \forall A\ \alpha, \beta.\texttt{Mask}\ A \to (@A\ \alpha \to \beta) \to A^{\infty}\ \alpha \to A^{\infty}\ \beta \\
\kappa(\texttt{zipWith}) &= \forall A\ \alpha, \beta, \gamma.(@A\ \alpha \to @A\ \beta \to \gamma) \to A^{\infty}\ \alpha \to A^{\infty}\ \beta \to A^{\infty}\ \gamma \\
\kappa(\texttt{foldT}) &= \forall A\ \alpha.(@T(A\ \alpha) \to \mathit{Continue}\ A\ \alpha) \to A\ \alpha \to A\ \alpha \\
\kappa(\texttt{grid}) &= \forall A\ \alpha.\ \langle A \rangle\ \texttt{int} \to \alpha \to A\ \alpha \\
\kappa(\texttt{reduce}) &= \forall A, B\ \alpha.(\alpha \to \alpha \to \alpha) \to A\ \alpha \to (A \times B)\ a \\
\kappa(\texttt{lift}) &= \forall A\ \alpha.A\ \alpha \to \texttt{Facet}\ A\ \alpha \to A^{\infty}\ \alpha \\
\kappa(\texttt{unlift}) &= \forall A\ \alpha.A^{\infty}\ \alpha \to A\ \alpha \\
\kappa(\texttt{mask}) &= \forall A.\ \{\text{after} : \langle A \rangle\ \texttt{int}, \text{before} : \langle A \rangle\ \texttt{int}, \text{scale} : \langle A \rangle\ \texttt{int}\} \\
&\qquad \to \texttt{Mask}\ A \\
\kappa(\texttt{constantBoundary}) &= \forall A\ \alpha.(\langle A \rangle\ \texttt{int} \to \alpha) \to \texttt{Boundary}\ A\ \alpha \\
\kappa(\texttt{wrapBoundary}) &= \forall A\ \alpha.\texttt{Boundary}\ A\ \alpha \\
\kappa(\texttt{flippedWrapBoundary}) &= \forall A\ \alpha.\texttt{Boundary}\ A\ \alpha \\
\kappa(\texttt{reflectBoundary}) &= \forall A\ \alpha.\texttt{Boundary}\ A\ \alpha \\
\kappa(\texttt{proceed}) &= \forall \alpha.\alpha \to \texttt{Continue}\ \alpha \\
\kappa(\texttt{terminate}) &= \forall \alpha.\texttt{Continue}\ \alpha
\end{aligned}
$$

Fig. 9. Type rules for the built-in functions

$$\boxed{\overset{pat}{\vdash}\ p : \Gamma, \tau}$$

$$\textsc{pattern-top-dim}\ \frac{\overset{pat}{\vdash}\ p : \Gamma, D, \tau \qquad D \neq \emptyset \qquad \text{exactly one context-qualified variable or wildcard in } p}{\overset{pat}{\vdash}\ p : \Gamma, @D\ \tau}$$

$$\textsc{pattern-top-plain}\ \frac{\overset{pat}{\vdash}\ p : \Gamma, \emptyset, \tau}{\overset{pat}{\vdash}\ p : \Gamma, \tau}$$

$$\boxed{\overset{pat}{\vdash}\ p : \Gamma, \mathcal{P}(d), \tau}$$

$$\textsc{wildcard}\ \frac{}{\overset{pat}{\vdash}\ \_ : \epsilon, D, \tau} \qquad \textsc{ctxt-wildcard}\ \frac{}{\overset{pat}{\vdash}\ @\_ : \epsilon, D, \tau}$$

$$\textsc{var}\ \frac{}{\overset{pat}{\vdash}\ x : \{x \rightarrowtail \tau\}, \emptyset, \tau} \qquad \textsc{ctxt-var}\ \frac{}{\overset{pat}{\vdash}\ @x : \{x \rightarrowtail \tau\}, \emptyset, \tau}$$

$$\textsc{1d}\ \frac{\overset{pat}{\vdash}\ p_1 : \Gamma_1, D, \tau \quad \cdots \quad \overset{pat}{\vdash}\ p_n : \Gamma_n, D, \tau}{\overset{pat}{\vdash}\ (\delta) : |\, p_1 \ldots p_n \,| : \biguplus_{i=1}^{n} \Gamma_i, D \cup \{\delta\}, \tau}$$

$$\textsc{2d}\ \frac{\overset{pat}{\vdash}\ p_{1,1} : \Gamma_{1,1}, D, \tau \quad \cdots \quad \overset{pat}{\vdash}\ p_{m,n} : \Gamma_{m,n}, D, \tau}{\overset{pat}{\vdash}\ (\delta_1,\ \delta_2) : \begin{vmatrix} p_{1,1} & \cdots & p_{m,1} \\ \vdots & \ddots & \vdots \\ p_{1,n} & \cdots & p_{m,n} \end{vmatrix} : \biguplus_{i=1}^{m} \biguplus_{j=1}^{n} \Gamma_{i,j}, D \cup \{\delta_1, \delta_2\}, \tau}$$

Fig. 10. Pattern matching rules

$$\boxed{d \overset{d}{<:} d}$$

$$\text{D-REFL}\ \frac{}{\delta \overset{d}{<:} \delta} \qquad \text{D-LOCAL}\ \frac{}{(\!|\delta|\!) \overset{d}{<:} \delta} \qquad \text{D-DIST}\ \frac{}{[\delta] \overset{d}{<:} \delta}$$

$$\text{D-PROD}\ \frac{\delta_1' \overset{d}{<:} \delta_1 \qquad \delta_2' \overset{d}{<:} \delta_2}{\delta_1' \times \delta_2' \overset{d}{<:} \delta_1 \times \delta_2}$$

$$\boxed{d^{top} \overset{d^{top}}{<:} d^{top}}$$

$$\frac{\text{TOP-D}}{d' \overset{d}{<:} d} d' \overset{d^{top}}{<:} d \qquad \frac{\text{TOP-D-INFTY}}{d' \overset{d}{<:} d} d'^{\infty} \overset{d^{top}}{<:} d^{\infty}$$

c

$$\boxed{\tau \uplus \tau^{-} \uplus \tau^{+} \uplus \tau^{prim} \overset{\tau}{<:} \tau \uplus \tau^{-} \uplus \tau^{+} \uplus \tau^{prim}}$$

$$\text{REFL}\ \frac{}{\tau \overset{\tau}{<:} \tau} \qquad \text{FUN}\ \frac{\tau_1 \to \tau_1' \qquad \tau_2' \to \tau_2}{\tau_1' \to \tau_2' \overset{\tau}{<:} \tau_1 \to \tau_2}$$

$$\text{GRID}\ \frac{d^{top\prime} \overset{d^{top}}{<:} d^{top} \qquad \tau' \overset{\tau}{<:} \tau}{@\, d^{top\prime}\ \tau' \overset{\tau}{<:} @\, d^{top}\ \tau} \qquad \text{GRID-CTXT}\ \frac{d^{top\prime} \overset{d^{top}}{<:} d^{top} \qquad \tau' \overset{\tau}{<:} \tau}{@\, d^{top\prime}\ \tau' \overset{\tau}{<:} @\, d^{top}\ \tau}$$

$$\text{RECORDS}\ \frac{\tau_1' \overset{\tau}{<:} \tau_1 \qquad \ldots \qquad \tau_n' \overset{\tau}{<:} \tau_n}{\{l_1 : \tau_1', \ldots, l_n : \tau_n'\} \overset{\tau}{<:} \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\text{VECTORS}\ \frac{\tau' \overset{\tau}{<:} \tau \qquad d^{top\prime} \overset{d^{top}}{<:} d^{top}}{\langle d^{top\prime} \rangle\ \tau' \overset{\tau}{<:} \langle d^{top} \rangle\ \tau}$$

Fig. 11. Subtyping relationships