Efficient and Correct Stencil Computation via Pattern Matching and Static Typing

Dominic Orchard

Computer Laboratory, University of Cambridge, UK

`dominic.orchard@cl.cam.ac.uk`

Alan Mycroft

Computer Laboratory, University of Cambridge, UK

`am@cl.cam.ac.uk`

Stencil computations, involving operations over the elements of an array, are a common programming pattern in scientific computing, games, and image processing. As a programming pattern, stencil computations are highly regular and amenable to optimisation and parallelisation. However, general-purpose languages obscure this regular pattern from the compiler, and even the programmer, preventing optimisation and obfuscating (in)correctness. This paper furthers our work on the *Ypnos* domain-specific language for stencil computations embedded in Haskell. Ypnos allows declarative, abstract specification of stencil computations, exposing the structure of a problem to the compiler and to the programmer via specialised syntax. In this paper we show the decidable safety guarantee that well-formed, well-typed Ypnos programs cannot index outside of array boundaries. Thus indexing in Ypnos is safe and run-time bounds checking can be eliminated. Program information is encoded as types, using the advanced type-system features of the Glasgow Haskell Compiler, with the safe-indexing invariant enforced at compile time via type checking.

1 Introduction

Stencil computations, otherwise known as *stencil codes* [29, p221] or *structured grid computations* [1], are a ubiquitous pattern in programming, particularly in scientific computing, games, image processing, and similar applications. Stencil computations are characterised by array operations where the elements of an output array are computed from corresponding elements, and their surrounding neighbourhood of elements, in an input array, or arrays. An example is the *discrete Laplace* operator which, for two-dimensional problems, can be written in C as the following, for input array A and output array B:

```
for (int i=1; i<(N+1); i++) {
  for (int j=1; j<(M+1); j++) {
    B[i][j] = A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1] - 4*A[i][j]
  }
}
```

where A and B are arrays of size $(N+2) \times (M+2)$ with indices ranging from $(0,0)$ to $(N+1, M+1)$. The *iteration space* of the `for`-loops ranges from $(1,1)$ to (N,M) where the elements outside of the iteration space provide a *halo* of the *boundary conditions* for the operation. The discrete Laplace operator is an example of a *convolution* operation in image processing [8].

Indexing in C is unsafe: any indexing operation may, without any static warnings, address memory "outside" the allocated memory region for A or B, causing a program crash or, even worse: affecting the numerical accuracy of the algorithm without crashing the program. Bounds-checked array access is provided by many languages and libraries to prevent unsafe program behaviour due to out-of-bounds access. However, bounds checking every array access has considerable performance overhead [16]; we

measured an overhead of roughly 20% per iteration for safe-indexing over unsafe-indexing in Haskell for a two-dimensional discrete Laplace operator on a 512×512 image (see Appendix A for details).

If it is known, or can be proved, that memory accesses are within the bounds of allocated regions, then costly bounds checking can be safely eliminated. In the presence of *general*, or *random*, indexing, such as in most language and libraries, automatic proof of the absence of out-of-bounds access is in general undecidable as indices may be arbitrary expressions.

This paper presents our latest work on *Ypnos*, an *embedded domain-specific language* in Haskell. Ypnos programs express abstract, declarative specifications of stencil computations which are guaranteed free from out-of-bounds array access, and thus array indexing operations without bounds checking can be used by the implementation. Ypnos lacks general array indexing but instead provides a form of relative indexing via a novel syntactic construction called a *grid pattern*, as introduced in [17]. Grid patterns provide decidable compile-time information about array access.

Our previous paper said little about boundary conditions and safety, instead focussing on expressivity and parallelisation. In this paper, we introduce language constructs in Ypnos for specifying the boundary conditions of a problem and the mechanism by which array indexing is guaranteed safe. Grid patterns are a typed construction which encode array access information in the types. Safe indexing is enforced via type checking at compile time, facilitated by Haskell type system features such as *type classes* [7, 20] and more advanced type system features found in the Glasgow Haskell Compiler (GHC) including: *generalised algebraic data types* (GADTs) [19, 18] and *type families* [4]. Dependent types have been used to enforce safe indexing in more general settings [25, 28]. However, a well-developed dependent type system is not required to enforce safe indexing in Ypnos due to the relative indexing scheme provided by the grid pattern syntax; instead GADTs suffice.

This paper was inspired by related work on the Repa array library for Haskell [10]. The most recent paper [14] introduces a mechanism for the abstract specification of convolution masks and a data structure for describing stencils. Given an array with adequate boundary values a stencil application function can safely elide bounds checking. We believe that the Ypnos offers greater flexibility and expressivity, particularly for higher-dimensionality problems and complex boundary conditions, facilitated by *grid patterns* and Ypnos's approach to boundaries and safety (see Section 5 for more on Repa).

Section 2 introduces the core aspects of the Ypnos language relevant to an end-user. We do not discuss all Ypnos features here. In particular we elide reductions, iterated stencil computations, and automatic parallelisation, details of which can be found in [17]. Section 3 discusses details of the Ypnos implementation, in particular the type-level techniques used for encoding and enforcing safety. The desugaring of Ypnos's specialised syntax is also informally described in this section. Section 4 provides a quantitative analysis of the effectiveness of Ypnos for producing correct, efficient stencil programs, along with performance numbers. Section 5 considers related work of other DSLs and languages for array programming. Section 6 discusses further work and Section 7 concludes with a discussion on the relevance of the techniques presented in this paper to the wider DSL audience. Appendix C provides a proof sketch of the soundness of the approach to safety presented in this paper.

The Ypnos implementation, and the source code for the programs used in this paper, can be downloaded from `http://github.com/dorchard/ypnos`.

A deep knowledge of Haskell is not required to understand the paper, although knowledge of ML or other functional language is helpful. A brief introduction to the more advanced GHC/Haskell type system features employed in the implementation is provided in Appendix B. Throughout, types starting with a lower-case letter are *type variables* and are implicitly universally quantified; types starting with an upper-case letter are *type constructors*, which are (possibly nullary) constructors e.g. *Int*.

2 Introduction to Ypnos

We introduce the main aspects of Ypnos that would be relevant to an end-user, using the example of the two-dimensional discrete Laplace operator:

```
[dimensions| X, Y |]

laplace2D = [fun|  X*Y:| _    t  _ |
                      | l @c  r |
                      | _    b  _ | -> t + l + r + b - 4.0*c  |]

laplaceBoundary = [boundary| Double from (-1, -1) to (+1, +1) -> 0.0 |]

grid = listGrid (Dim X :* Dim Y) (0, 0) (w, h) img_data laplaceBoundary
grid' = runA grid laplace2D
```

Here `img_data`, `w`, and `h` are free variables defined elsewhere, providing the data of a grid as a list of double-precision floating point values, and the horizontal and vertical size of the grid respectively.

Ypnos mostly comprises library functions thus the syntax of Ypnos programs is largely that of Haskell. However there is some important Ypnos-specific syntax written inside of *quasiquoting* brackets [15] and expanded by macro functions, with the forms:

```
[dimension| ...ypnos code... |]
[fun| ...ypnos code... |]
[boundary| ...ypnos code... |]
```

The macros essentially give an executable semantics to the specialised syntax of Ypnos (discussed further in Section 3). Within the quasiquoted brackets any expression following an arrow symbol `->` is a Haskell expression i.e. in Lisp terminology, expressions following `->` are *backquoted*.

2.1 Grids and Dimensions

Ypnos is built around a central data type of *n*-dimensional immutable arrays. The term *grid* is used instead of *array* to escape implementational connotations. There are a number of grid constructors, one of which is used in the above example: `listGrid`, which takes five arguments: a *dimension term*, the lower extent of the grid, the upper extent of the grid, a list of element values, and a structure describing the grid's *boundary behaviour*.

Dimension terms define the *dimensionality* of a grid, naming the dimensions. Any dimension names used in a program must first be declared in a single declaration via the `[dimension| ...  |]` macro, e.g. in the example the X and Y dimensions are declared. A dimension term is constructed from one or more dimension identifiers, prefixed by `Dim` constructors, where many dimension can be composed by the binary `:*` dimension-tensor operation.

2.2 Indexing: Grid Patterns

Ypnos has no general indexing operations on grids. Indexing is instead provided by a special pattern matching syntax called a *grid pattern*, first introduced in the earlier Ypnos paper [17]. A grid pattern is a group of variable or wildcard patterns which are matched onto a subset of elements in a grid relative to a particular element. For example, the following is a one-dimensional grid pattern on the *X* dimension:

```
X:| l @c r |
```

This grid pattern, which matches on a one-dimensional grid of dimension X, binds the variable pattern c to the element of the grid which is indexed by a *cursor* index internal to the grid data structure. The cursor is the index of the current iteration of an array operation. The variable patterns l and r are bound to elements relative to the cursor element i.e. l to the preceding element and r to the succeeding element. The above grid pattern is analogous to the following bindings in C: l = A[i-1]; c = A[i]; r = A[i+1]; for an array A and index i, where i is the *cursor* index.

Every grid pattern must contain exactly one sub-pattern prefixed by @ which is matched to the grid element indexed by the grid's cursor. The relative lexical position of the remaining patterns to the cursor pattern determines to which element in the grid each pattern should be matched.

One-dimensional grid patterns can be nested to provide *n*-dimensional patterns. For example, a pattern match on a two-dimensional grid, of dimensionality Dim X :* Dim Y, can be defined:

```
Y:|   X:| lt @lc lb |
   @ X:| ct @cc cb |
     X:| rt @rc rb | |
```

Here the outer grid pattern has exactly one pattern prefixed by @ and the inner grid patterns also have exactly one pattern marked by @. Grid patterns are prefixed by a dimension term to disambiguate the dimension being matched upon by a stencil. For syntactic convenience, Ypnos provides a two-dimensional grid pattern syntax which is considerably easier to read and write than nested patterns. In two-dimensional grid pattern syntax, the above pattern can be written:

```
X*Y:|  lt    lc   lb |
    |  ct @ cc   cb |
    |  rt    rc   rb |
```

The layout of the grid pattern syntax is essentially pictorial in its representation of a stencil's access pattern, where the spatial relationships of patterns match the spatial relationships of data.

The above access pattern is known as a *nine-point* two-dimensional stencil. The Laplace example uses a *five-point* stencil where a wildcard pattern is used to ignore the corner elements.

Grid patterns restrict the programmer to relative array-indexing, expressed as a static construction that requires no evaluation or reduction. Grid patterns thus provide decidable compile-time information about the access pattern of a program without the need for analysis further than parsing. Because indexing is static, the access pattern of a stencil function can be encoded as a type (discussion further in Section 3). The fun macro allows a *stencil function* to be defined by a grid pattern and, following the arrow symbol ->, a Haskell expression which defines the body of the stencil function.

2.3   Applying Stencil Functions and Boundaries

There are several functions for applying stencil functions to grids. The example uses runA which applies a stencil function at each position in the parameter grid by instantiating the internal cursor index of the grid to each index in the range $(0,0)$ to $(w-1, h-1)$ (inclusive), writing the results into a new grid. This range of indices is called the *extent* of the grid.

The grid pattern of `laplace2D` accesses indices $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, and $(i,j+1)$ relative to the cursor index (i,j) thus application of the stencil function at indices around the *edge* of the grid, e.g. $(0,0)$, results in out-of-bounds access. Errors or undefined behaviour are avoided by specification of boundary behaviour by `laplaceBound`, which is passed to the `listGrid` constructor.

Boundary behaviour can be specified in a number of ways inside the `boundary` quasiquoted macro. In the example we defined the boundary behaviour by:

```
[boundary| Double from (-1, -1) to (+1, +1) -> 0.0 |]
```
---

which specifies a one-element deep *boundary region*, or *halo*, around a grid of element type `Double` with a default value of `0.0`. The `from ...  to ...` syntax is itself short-hand for a more verbose description specifying smaller sub-regions of the boundary. The above is short-hand for the following:

```
[boundary| Double   (-1, -1) -> 0.0
                    (*i, -1) -> 0.0
                    (+1, -1) -> 0.0
                    (-1, *j) -> 0.0
                    (+1, *j) -> 0.0
                    (-1, +1) -> 0.0
                    (*i, +1) -> 0.0
                    (+1, +1) -> 0.0 |]
```

Each case defines the value of a boundary region where the left-hand side of `->` is a *region descriptor*. Figure 1 gives a pictorial representation of boundary regions, and their descriptors, for a two-dimensional grid with a one-element boundary region. Note, `*v` denotes a variable binding for a variable `v`.

$$
\begin{array}{ll}
a = \texttt{(-1, -1)} & e = \texttt{(+1, *j)} \\
b = \texttt{(*i, -1)} & f = \texttt{(-1, +1)} \\
c = \texttt{(+1, -1)} & g = \texttt{(*i, +1)} \\
d = \texttt{(-1, *j)} & h = \texttt{(+1, +1)}
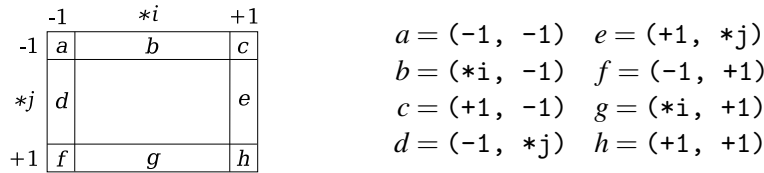\end{array}
$$

Figure 1: Boundary region descriptors for a two-dimensional grid with a one element-deep boundary

The grammar of boundary terms is the following, where *n* are natural numbers greater than one, *e* are Haskell expressions, and *v* are variables:

$$
\begin{aligned}
B &::= \texttt{from } I_1 \texttt{ to } I_2 \texttt{ -> } e \mid I \texttt{ -> } e \mid I \; g \texttt{ -> } e \\
I &::= P \mid \texttt{(}P\texttt{, }P\texttt{)} \mid \texttt{(}P\texttt{, }P\texttt{, }P\texttt{)} \mid \texttt{(}P\texttt{, }P\texttt{, }P\texttt{, }P\texttt{)} \mid \ldots \\
P &::= \texttt{-}n \mid \texttt{+}n \mid \texttt{*}v
\end{aligned}
$$

A boundary definition consists of an explicit element type and a number of *B* terms which specify either a range of boundary regions with the same value for the elements of each region or a specific boundary region. The three forms are called the *range form*, the *specific form*, and the *specific parameterised form*. The terminal +*n* denotes a boundary region *n* elements after the upper extent of a grid dimension; −*n* denotes a boundary region *n* elements before the lower extent of a grid dimension; *v denotes any index inside the extent of a grid dimension where the value of the index is bound to *v* (see Figure 1).

As an example of a more complicated boundary definition, the following specifies the boundary of a two-dimensional grid where the "left" side of the grid (i.e. regions *a*, *d*, and *f* in Figure 1) has default value 1.0 to a depth of one element, the "right" side (i.e. regions *c*, *e*, and *h* in Figure 1) has default value 2.0, one-element deep, and the remaining edges have default value 0.0, one-element deep:

```
[boundary| Double  from (-1, -1) to (-1, +1) -> 1.0
                   from (+1, -1) to (+1, +1) -> 2.0
                   (*i, -1) -> 0.0
                   (*i, +1) -> 0.0 |]
```

More complex boundary behaviour such as *reflecting* or *wrapping* boundaries can be described by the *specific parameterised form* which is the same as the *specific form* with an additional parameter for the grid for which the boundary is being defined. As an example of the specific parameterised form, the following defines the boundary of a two-dimensional grid where the "top" edge reflects the values inside the grid to a depth of one element, the "left" and "right" edges are wrapped, the "bottom" edge has a constant value up to a depth of two elements, and the corner cases, $(-1, -1)$ and $(+1, -1)$, reflect the corresponding corner elements inside the grid:

```
[boundary| Double  (*i, -1) g -> g!!!(i, 0)                    -- top
                   (-1, *j) g -> g!!!(fst (size g) - 1, j)     -- left
                   (+1, *j) g -> g!!!(0, j)                    -- right
                   from (-1, +1) to (+1, +2) -> 0.0            -- bottom
                   (-1, -1) g -> g!!!(0, 0)                    -- top corners
                   (+1, -1) g -> g!!!(fst (size g) - 1, 0) |]
```

The (!!!) operator is a safe general indexing operation (i.e. out-of-bounds access raises an exception) that is only available within a boundary expression. Bounds checks for (!!!) do not have a significant effect on program performance as such a boundary is a relatively small percentage of the overall grid size and is only calculated once per application of a stencil function to update the boundary values from the updated grid values. Note, a boundary without any parameterised regions is constructed just once overall as its values do not, and cannot, depend on a grid's inner values.

The boundary definition of a grid must provide sufficient cover for a stencil function such that the stencil function does not access an element which is outside of the grid or boundary region. Since grid patterns and boundary definitions are static this safety property can be checked statically. Any program without appropriate boundaries for a stencil application is rejected.

3   Statically Enforcing Safe Indexing via Types

Ypnos provides efficiency and correctness guarantees by enforcing safe indexing statically via typing. Thus, well-typed Ypnos programs are safe, and bounds-check free.

Type-level information in Ypnos is constructed, composed, and manipulated using various type-system features of Haskell and the Glasgow Haskell Compiler. *Generalised algebraic data types* (GADTs) [19, 18] are used as a form of lightweight dependent-typing, propagating information from data-level to type-level via the constructors of a data type. *Type families* [4] are used as simple type-level functions to manipulate/compose type-level information. *Type classes* [7] are used as predicates and relations on types. Appendix B provides a brief introduction to these type-system features for the unfamiliar reader.

We introduce here the core aspects of the Ypnos implementation that are most relevant to enforcing safe indexing. This section also explains, mostly by example, the desugaring of Ypnos syntax into Haskell. We begin with an informal high-level overview of how safety is enforced before delving into details and definitions. So far, Ypnos programs have been typeset in `monospace` to present a clear view of the concrete syntax. The Haskell code shown here is typeset more elaborately to ease reading.

3.1 Safety Overview

The key to enforcing safety in Ypnos is the encoding of relative indices at the type-level. Type-level relative indices are used in the types of stencil functions and grids in two different ways: respectively, to encode the relative indexing pattern of a stencil function and to encode the relative position of boundary regions with respect to the edge of a grid.

The type of a grid is parameterised by type-level information about the grid's boundary regions. The grid data type takes four type parameters:

$$Grid\ d\ b\ dyn\ a$$

where d is a dimensionality type, b is a type-level list of the relative indices (with respect to the grid's edge) of the grid's boundary regions, and a is the element type of the grid (*dyn* will be discussed later).

Stencil functions include in their type the relative indices accessed by the function, as described by a grid pattern. Grid patterns are desugared via the `fun` macro into relative indexing operations on a grid, with type approximately of the form:

$$index :: Safe\ i\ b \Rightarrow i \rightarrow Grid\ d\ b\ dyn\ a \rightarrow a$$

where *index* takes a relative index of type i and a grid of element type a, returning a single value of type a. The *Safe i b* constraint enforces safety by requiring that there are sufficient boundary regions described by b such that a relative index i accessed from anywhere within the grid's extent has a defined value.

The boundary information that parameterises a grid type (type parameter b above) is generated from a boundary description used in constructing a grid. The `boundary` macro desugars a boundary description into a special data structure which some grid constructors take as a parameter.

The rest of this section is structured as follows: Section 3.2 introduces *dimensionality* types, which determine the index type of a grid. Section 3.3 introduces absolute index types and the important relative-index type representation. Section 3.4 discusses boundary definitions and boundary types, which leads into Section 3.5 on grid constructors where boundary information propagates to grid types. Section 3.6 brings together the different type-level information, showing how *Safe* is defined to match relative indices to grid boundaries. Section 3.7 completes by showing application of stencil functions to grids.

3.2 Dimensions

As described above, the *Grid* data type has a type parameter representing the *dimensionality* of a grid. In the example of Section 2, one of the parameters to the constructor `listGrid` was a dimensionality term: `Dim X :* Dim Y` specifying that the grid is two-dimensional with named dimensions X and Y.

Dimension terms are defined by the following GADT, with a constructor *Dim* for single dimensions and a constructor :* for the tensor of a single dimension and a dimension term[1]:

---

[1]This tensor operator is theoretically associative but the implementation is simplified by defining a right-associating operator.

data *Dim d*
data $(:^*) \, d \, d'$

data *Dimensionality d* **where**
  $Dim :: DimIdentifier \, d \Rightarrow d \rightarrow Dimensionality \, (Dim \, d)$
  $(:^*) :: Dimensionality \, (Dim \, d) \rightarrow Dimensionality \, d' \rightarrow Dimensionality \, (Dim \, d :^* d')$

In the first two lines, the data types *Dim* and $(:^*)$ are *empty declarations* defining type constructors without data constructors. The data constructors of *Dimensionality* reuse these names[2] and use these empty types in the result type's parameter as type representatives for the data constructors. The types of dimensionality terms are *singleton types*, i.e. types with just one inhabitant, thus their type uniquely determines their value. For example, the term *Dim X :* Dim Y* has type *Dimensionality (Dim X :* Dim Y)*.

The *DimIdentifier* class (which constraints the *Dim* constructor) is that of valid dimension identifier types. Valid dimension identifiers are declared in Ypnos by a [dimension| ... |] macro, which expands a list of dimension identifiers into singleton data types, representing dimension identifiers, and instances of *DimIdentifier* e.g.

$$[dimension| \; X, \; Y \; |] \quad \rightsquigarrow \quad \begin{array}{l} \textbf{data } X = X \\ \textbf{data } Y = Y \\ \textbf{instance } DimIdentifier \; X \\ \textbf{instance } DimIdentifier \; Y \end{array}$$

3.3 Absolute and Relative Indices

Internally, grids are indexed by tuples of *Int* values. Given the dimensionality of a grid the type of *absolute indices* is calculated by the *Index* type family with instances (of which we show the first three):

type family *Index t*
type instance $Index \, (Dim \, x) \qquad\qquad\qquad = Int$
type instance $Index \, ((Dim \, x) :^* (Dim \, y)) \qquad = (Int, Int)$
type instance $Index \, ((Dim \, x) :^* ((Dim \, y) :^* (Dim \, z))) = (Int, Int, Int)$

Unfortunately, tuples in Haskell are not inductively defined thus a fully general implementation of Ypnos must have an instance of *Index* for every possible arity of dimensionality i.e. an infinite number. In practice, many stencil computations do not require more than four dimensions. We discuss this issue further in Section 6 and in the rest of the paper give tuple-related definitions up to two or three dimensions.

As mentioned, *relative indices* have a type-level encoding which is key to enforcing safety. Relative indices are given by tuples of a data type encoding of integers, *IntT*, providing a type-level representation of integers. *IntT* is defined in terms of a data type of inductively defined natural numbers, *Nat*, for which the result type of each data constructor provides a type-level representation of the natural number constructed, as defined by the following GADT:

data *Z*
data *S n*

data *Nat n* **where**
  $Z :: Nat \, Z$
  $S :: Nat \, n \rightarrow Nat \, (S \, n)$

---

[2]The data constructors could have been given different names to their type representatives, e.g. *DimIdent* $:: d \rightarrow$ *Dimensionality* (*Dim d*), but we find the correspondence between data constructors and type representatives more instructive.

Integers are defined by *IntT* in terms of *Nat* as either a negative or positive natural number:

> **data** *Neg n*
> **data** *Pos n*
> **data** *IntT n* **where**
>   *Neg* :: *Nat* (*S n*) → *IntT* (*Neg* (*S n*))
>   *Pos* :: *Nat n* → *IntT* (*Pos n*)

Note that zero has a unique representation: *Pos Z*. The expression *Neg Z* is not well-typed as the *Neg* data constructor expects a natural number of at least one i.e. of type *Nat* (*S n*).

The *Grid* data type is parameterised by a type-level list of the relative positions of boundary regions. An example type-level list construction is the following GADT of heterogeneous lists:

> **data** *Nil*
> **data** *Cons x xs*
> **data** *List t* **where**
>   *Nil* :: *List Nil*
>   *Cons* :: *x* → *List xs* → *List* (*Cons x xs*)

The *List* type thus encodes the types of its elements in its type parameter *t* as a type-level list e.g.

> (*Cons* 1 (*Cons* "hello" (*Cons* 'a' *Nil*))) :: (*List* (*Cons Int* (*Cons String* (*Cons Char Nil*))))

The type-level representation of a grid's boundary regions uses the same type-level list representation, but the list contains relative indices describing the position of boundary regions relative to the grid's edge. For example, a one-dimensional grid with boundary regions of −1 and +1 has a type like:

> *Grid* (*Dim d*) (*Cons* (*IntT* (*Neg* (*S Z*)))) (*Cons* (*IntT* (*Pos* (*S Z*))) *Nil*)) *dyn a*

3.4 Boundaries

A relative index cannot be accessed safely from everywhere within a grid's extent unless the grid has appropriate boundary values, provided by a boundary definition. Ypnos boundary definitions are desugared by the `boundary` macro into a data structure describing the boundary which encodes in its type the positions of the boundary elements defined, relative to the edge of the grid. This type-level boundary information is propagated to a grid's type (the *b* type parameter in the *Grid* type). The *Safe* constraint in the type of relative indexing operations uses this boundary information to ascertain if a grid has adequate boundaries for the relative index to be safe when accessed from anywhere within the grid's extent.

Section 2.3 introduced Ypnos's syntax for describing boundaries, with the grammar:

$$B ::= \texttt{from } I_1 \texttt{ to } I_2 \texttt{ -> } e \mid I \texttt{ -> } e \mid I \texttt{ } g \texttt{ -> } e$$
$$I ::= P \mid (P, P) \mid (P, P, P) \mid (P, P, P, P) \mid \ldots$$
$$P ::= -n \mid +n \mid *v$$

The syntax, *I g -> e*, describes boundary values that are dependent on the contents of a grid g. Such a boundary is described as *dynamic*. After the application of a stencil function to a grid with a dynamic boundary, the boundary values must be recomputed so that its values are consistent with the grid's (possibly changed) inner elements. Conversely, a non-parameterised boundary description is *static*, since it

does not depend on the values of the grid and thus does not need recomputing. Since the `from...to...` syntax is desugared into a number of specific forms `...->...`, as described in Section 2.3, boundary definitions consist of a number of *B* terms of either the second or third syntactic form.

Boundary definitions are desugared by the `boundary` macro into a *BoundaryList* structure which comprises a list of *BoundaryFun* values. Each *B* term in a boundary definition is desugared into a function mapping one or more boundary indices, determined by the region descriptor, to a value. From such functions a *BoundaryFun* value is constructed which encodes information in its type about the boundary region it defines. *BoundaryFun* has two constructors for static and dynamic regions:

> **data** *Static*
> **data** *Dynamic*
>
> **data** *BoundaryFun d ix a dyn* **where**
>   *Static*   :: $(ix \rightarrow a)$ $\rightarrow$ *BoundaryFun d ix a Static*
>   *Dynamic* :: $((ix, Grid\ d\ Nil\ Static\ a) \rightarrow a) \rightarrow$ *BoundaryFun d ix a Dynamic*

Both the *Static* and *Dynamic* constructors of *BoundaryFun* have a single parameter: a function mapping one or more boundary indices of type *ix* to a value of type *a*. In the case of a *Dynamic* boundary function, this index is paired with a grid of element type *a*. *BoundaryFun* has four type parameters: *d* the grid's dimensionality, *ix* the boundary indices type, *a* the element type, and *dyn* denoting whether the boundary is static or dynamic.

The region descriptor of a boundary (non-terminal *I* in the grammar) determines the type of boundary indices[3]. The components of a boundary index[3] are either relative or absolute: $-n$ and $+n$ are relative indices, relative to the lower or upper bound respectively of the dimension's extent, and $*v$ is an absolute index (i.e. an *Int* value) within the the dimension's extent. The region descriptor syntax is desugared into pattern matches on relative/absolute indices in the following way:

Syntax	Pattern	Type
`-n`	*Neg* $[\![n]\!]$	*IntT* (*Neg* $[\![n]\!]$)
`+n`	*Pos* $[\![n]\!]$	*IntT* (*Pos* $[\![n]\!]$)
`*v`	v	*Int*

where $[\![0]\!] \rightsquigarrow Z$

            $[\![n]\!] \rightsquigarrow S\ [\![n-1]\!]$

at both the data- and type-level.

Figure 2 provides an example of the types of the boundary indices for the boundary regions of a two-dimensional grid with a one-element deep boundary. The following shows the desugaring of two

$a = (IntT\ (Neg\ (S\ Z)), IntT\ (Neg\ (S\ Z)))$ $e = (IntT\ (Pos\ (S\ Z)), Int)$
$b = (Int, IntT\ (Neg\ (S\ Z)))$ $f = (IntT\ (Neg\ (S\ Z)), IntT\ (Pos\ (S\ Z)))$
$c = (IntT\ (Pos\ (S\ Z)), IntT\ (Neg\ (S\ Z)))$ $g = (Int, IntT\ (Pos\ (S\ Z)))$
$d = (IntT\ (Neg\ (S\ Z)), Int)$ $h = (IntT\ (Pos\ (S\ Z)), IntT\ (Pos\ (S\ Z)))$

Figure 2: Boundary indices types for a two-dimensional grid with one-element boundary

---

[3]The components of an index are the individual elements of the index's tuple for each dimension.

example boundary definitions into *BoundaryFun* values, with explicit type signatures:

$$\llbracket (\text{-1, +1}) \text{ -> } 0.0 \rrbracket$$
$$\rightsquigarrow (Static \ (\lambda \ (Neg \ (S \ Z), Pos \ (S \ Z)) \rightarrow 0.0))$$
$$:: (BoundaryFun \ (Dim \ d :^* Dim \ d') \ (IntT \ (Neg \ (S \ Z)), IntT \ (Pos \ (S \ Z))) \ Double \ Static)$$
$$\llbracket (\text{*i, +2}) \text{ g -> g!!!(i, 1)} \rrbracket$$
$$\rightsquigarrow (Dynamic \ (\lambda \ ((i, Pos \ (S \ (S \ Z))), g) \rightarrow g \, !!! \, (i, 1)))$$
$$:: (BoundaryFun \ (Dim \ d :^* Dim \ d') \ (Int, IntT \ (Pos \ (S \ (S \ Z)))) \ Double \ Dynamic)$$

Functions with GADT parameters require explicit type signatures [18] thus the `boundary` macro generates such type signatures for the boundary functions it defines, of which we showed two examples above. The boundary index types and the type for static or dynamic boundaries can be easily generated from the syntax of a boundary definition during desugaring. However, the element type of a grid/boundary cannot be constructed without performing type inference. To simplify the implementation the user must provide a monomorphic type for the boundary values at the start of a boundary definition, as seen in the Laplace example where the *Double* type is specified.

BoundaryFun values are composed via the *BoundaryList* structure which is similar to the *List* GADT seen earlier but with additional type-level information, defined:

> **data** *BoundaryList b dyn lower upper d a* **where**
>   *NilB* :: *BoundaryList Nil Static* (*Origin d*) (*Origin d*) *d a*
>   *ConsB* :: *BoundaryFun d ix a dyn*
>   → *BoundaryList b dyn' lower upper d a*
>   → *BoundaryList* (*Cons* (*AbsToRel ix*) *b*) (*Dynamism dyn dyn'*)
>   (*Lower ix lower*) (*Upper ix upper*) *d a*

The *BoundaryList* structure has six type-parameters encoding inductively-defined information about the boundary regions via the *NilB* constructor for the base case and *ConsB* constructor for the inductive case. The first type-parameter *b* is a type-level list, defined using the *Nil* and *Cons* type constructors, which encodes the boundary indices defined by the *BoundaryList*. In the inductive case, the *AbsToRel* type family is applied to the boundary indices type *ix* of the *BoundaryFun*. *AbsToRel* converts a boundary indices type into relative index type. Boundary indices have components that are either absolute or relative indices. *AbsToRel* converts absolute indices or type *Int*, which are indices within the extent of a dimension, to zero-relative indices as an index within the extent of a dimension is neither greater than or less than the extent of a dimension. *AbsToRel* is defined:

> **type family** *AbsToRel t*
> **type instance** *AbsToRel Int* = *IntT* (*Pos Z*)   -- *Int* to zero (*IntT* (*Pos Z*))
> **type instance** *AbsToRel* (*IntT n*) = *IntT n*   -- *IntT* to *IntT*
> **type instance** *AbsToRel* (*a, b*) = (*AbsToRel a, AbsToRel b*)   -- Two-dimensional case

The type-level list of boundary indices is thus a list of relative indices with respect to the grid's lower or upper extent, where a zero-relative index component means any index within the grid's extent. This representation is useful for matching relative indices from indexing operations with the boundary information of a grid, to which we return in Section 3.6).

The remaining parameters to *BoundaryList* encode other information which is used by the implementation but is not involved in enforcing safety. Briefly, *dyn* describes whether all boundary sub-definitions

are static or if at least one is dynamic. The base case is that all are *Static*. The inductive case applies the *Dynamism* type family, which is monotonic (in the two element domain {*Static*, *Dynamic*}), calculating the dynamic property for the entire boundary:

> **type family** *Dynamism t t′*
> **type instance** *Dynamism Static Static* = *Static*
> **type instance** *Dynamism Static Dynamic* = *Dynamic*
> **type instance** *Dynamism Dynamic Static* = *Dynamic*
> **type instance** *Dynamism Dynamic Dynamic* = *Dynamic*

The *lower* and *upper* parameters encode the lower and upper bounds of the boundary regions, relative to the lower and upper bounds of the grid's extent, which is used by the implementation to construct the data array for a grid. The base case for both is the *Origin* for a particular dimensionality, which is the zero-relative index e.g. *Origin* ((*Dim d*) :* (*Dim d′*)) = (*IntT* (*Pos Z*), *IntT* (*Pos Z*)). The inductive cases are calculated by the *Lower* and *Upper* type families which compute the lower and upper bound of a pair of boundary indices, defined as the minimum and maximum of each component of an index.

Lastly, the *d* parameter of *BoundaryList* is the dimensionality of the grid for which this is the boundary, and *a* is the element type of the grid/boundary.

In the Laplace example of Section 2, the type of the boundary definition, `laplaceBoundary`, is:

> *BoundaryList* (*Cons* (*IntT* (*Neg* (*S Z*)), *IntT* (*Neg* (*S Z*)))   -- (-1, -1)
>   (*Cons* (*IntT* (*Neg* (*S Z*)), *IntT* (*Pos Z*))   -- (-1, *j)
>   (*Cons* (*IntT* (*Neg* (*S Z*)), *IntT* (*Pos* (*S Z*)))   -- (-1, +1)
>   (*Cons* (*IntT* (*Pos Z*),   *IntT* (*Neg* (*S Z*)))   -- (*i, -1)
>   (*Cons* (*IntT* (*Pos Z*),   *IntT* (*Pos* (*S Z*)))   -- (*i, +1)
>   (*Cons* (*IntT* (*Pos* (*S Z*)), *IntT* (*Neg* (*S Z*)))   -- (+1, -1)
>   (*Cons* (*IntT* (*Pos* (*S Z*)), *IntT* (*Pos Z*))   -- (+1, *j)
>   (*Cons* (*IntT* (*Pos* (*S Z*)), *IntT* (*Pos* (*S Z*)))   -- (+1, +1)
>   *Nil*))))))))) *Static*
>   (*IntT* (*Neg* (*S Z*)), *IntT* (*Neg* (*S Z*))) (*IntT* (*Pos* (*S Z*)), *IntT* (*Pos* (*S Z*)))
>   (*Dim X* :* *Dim Y*) *Double*

Fortunately such a type is never constructed by an end-user, although it is possible for such a type to be seen as an error message (see Section 6 for further discussion). Note that for *Dynamism*, *Lower*, and *Upper* a corresponding value is not computed, thus these computations occur only at compile-time.

In the original paper there was some mention of a boundary structure called a *facets* structure, this corresponds to the *BoundaryList* structure which has been elaborated here following recent research.

3.5   Grids and Grid Constructors

A *BoundaryList* structure can be passed to grid constructor to define the boundary of a grid. The first type parameter of a *BoundaryList* structure provides type-level information of the boundary it defines as a list of relative indices of the boundary regions. A grid constructed with a *BoundaryList* adopts this type-level boundary information in its type.

The *Grid* data type has four type parameters: *Grid d b dyn a*, where *d* is a dimensionality type, *b* is the boundary information, *dyn* describes whether the grid's boundary is static or dynamic, and *a* is the element type of the grid. *Grid* is defined by the following GADT with just one constructor:

```
data Grid d b dyn a where
   Grid :: (IArray UArray a)
         ⇒ (UArray (Index d) a)              -- Array of values
         → Dimensionality d                  -- Dimensionality term
         → Index d                           -- Cursor ("current index")
         → (Index d, Index d)                -- Lower and upper bounds of grid extent
         → BoundaryList b dyn lower upper d a -- Boundary definition
         → Grid d b dyn a
```

The type parameters *b* and *dyn* of *BoundaryList*, which encode boundary regions and dynamism information, are transferred to the constructed *Grid*. The *IArray UArray a* constraint is a consequence of the underlying implementation of grids as Haskell unboxed array data types.

The *Grid* data constructor is hidden from the end-user. Instead a number of different constructor functions are provided, the most common of which are *listGrid* (seen earlier) and *grid*, which have type:

```
grid ::   (IArray UArray a, Reifiable upper (Index d), Reifiable lower (Index d)) ⇒
          Dimensionality d → Index d → Index d → [(Index d, a)] →
          BoundaryList b dyn lower upper d a → Grid d b dyn a

listGrid :: (IArray UArray a, Reifiable upper (Index d), Reifiable lower (Index d)) ⇒
          Dimensionality d → Index d → Index d → [a] →
          BoundaryList b dyn lower upper d a → Grid d b dyn a
```

The first three parameters of both constructors are a dimensionality term, a lower-bound index, and an upper-bound index. The fourth parameter for *listGrid* is a list of elements which define the data of the grid, and for *grid* a list of index-element pairs. The fifth parameter of both is a boundary definition.

Recall that relative indices are singleton types, thus the type uniquely determines the value. The *Reifiable* class provides functions for constructing values from relative index types. In this case, the *upper* and *lower* bounds of a boundary can be reified as absolute indices. *Reifiable* is used elsewhere in the implementation to construct relative index values, or *Int*-valued relative indices, from a type.

Ypnos provides two further constructors *listGridNoBoundary* and *gridNoBoundary*, not shown here. Both are similar to *listGrid* and *grid* but are not parameterised by a *BoundaryList* structure, returning a grid of type *Grid d Nil Static a*, where *Nil* boundary information implies no boundary.

3.6 Grid Patterns, Indexing, and Safety

Grid patterns are desugared into a number of relative indexing operations which are internally implemented without bounds-checks. The one- and two-dimension relative indexing operations have type:

```
index1D :: Safe (IntT n) b ⇒
           IntT n → Int → Grid (Dim d) b dyn a → a
index2D :: Safe (IntT n, IntT n′) b ⇒
           (IntT n, IntT n′) → (Int, Int) → Grid (Dim d :* Dim d′) b dyn a → a
```

Note that, in both cases, the first parameter is a relative index and the second parameter is an absolute index. The `fun` macro generates at compile time both an inductively defined relative index, to produce the correct type constraints, and an *Int*-valued relative index which is used to perform the actual access.

This second parameter is provided so that an inductive relative index does not have to be (expensively) converted into an *Int* representation for actual indexing at run-time.

The *Safe* constraint enforces safety by requiring that a grid's boundary provides sufficient elements outside of a grid such that a relative index has a defined value if accessed from anywhere within the grid. As a first approximation *Safe* is essentially defined as membership of a relative index to the list of boundary regions for a grid, which are the relative indices of boundary regions from the edge of the grid.

The *InBoundary* class is used by *Safe* as a predicate to test whether a relative index matches a relative position of a boundary region in the list of boundary regions:

> **class** *InBoundary i ixs*
> **instance** *InBoundary i* (*Cons i ixs*)   -- List head matches
> **instance** *InBoundary i ixs* ⇒ *InBoundary i* (*Cons i' ixs*)   -- List head does not match, recurse

For example, consider a two-dimensional grid with a one-element deep boundary:

	-1	*i	+1
-1	a	b	c
*j	d		e
+1	f	g	h

The relative index $(0, +1)$, represented by type (*IntT* (*Pos Z*), *IntT* (*Pos* (*S Z*))), requires the *g* boundary to be safe. The boundary function for *g* pattern matches on boundary indices of type (*Int*, *IntT* (*Pos* (*S Z*))). As *AbsToRel* maps *Int* to a zero-relative index in a boundary indices type (see Section 3.4), the relative position of *g* is encoded in the type-level boundary information as: (*IntT* (*Pos Z*), *IntT* (*Pos* (*S Z*))) i.e. the same representation as the relative index $(0, +1)$. Thus, our first approximation of safety as membership of a relative index to the list of boundary regions would in this situation be sound. However, other relative indices require several boundary regions to be defined. For example, safety of the relative index $(+1, +1)$ requires the boundaries *g*, *h*, and *e*.

Safe is implemented in the following way: for every relative index we must test the space of possible indices it may access outside of a grid boundary. For the one-dimensional case *Safe* is defined as such:

> **class** *Safe i b*
> **instance** *Safe* (*IntT* (*Pos Z*)) *b*
> **instance** (*Safe* (*IntT* (*Pred n*)) *b*, *InBoundary* (*IntT n*) *b*) ⇒ *Safe* (*IntT n*) *b*

The first instance defines that zero-relative indices are always safe. The second instance defines that a relative index is safe if it is a member of the boundary regions for a grid and if its predecessor is also safe. *Pred* is defined:

> **type family** *Pred n*
> **type instance** *Pred* (*Neg* (*S* (*S n*))) = *Neg* (*S n*)   -- *Pred* $-(n+1) = -n$
> **type instance** *Pred* (*Neg* (*S Z*)) = *Pos Z*   -- *Pred* -1 $= 0$
> **type instance** *Pred* (*Pos Z*) = *Pos Z*   -- *Pred* 0 $= 0$
> **type instance** *Pred* (*Pos* (*S n*)) = *Pos n*   -- *Pred* $+(n+1) = +n$

Thus, for a relative index of say (-2) there must be a boundary region for (-2) and (-1), and for a relative index of $(+2)$ there must be a boundary region $(+2)$ and $(+1)$. *Pred* is thus the predecessor of a relative index, approaching zero from both the negative and positive directions.

For the two-dimensional case *Safe* is defined similarly:

> **instance** *Safe* (*IntT* (*Pos Z*), *IntT* (*Pos Z*)) *b*
>
> **instance** (*Safe* (*IntT* (*Pred n*), *IntT* *n′*) *b*, *Safe* (*IntT* *n*, *IntT* (*Pred n′*)) *b*,
>   *InBoundary* (*IntT* *n*, *IntT* *n′*) *b*) ⇒ *Safe* (*IntT* *n*, *IntT* *n′*) *b*

Again, the first instance states that a zero-relative index is safe. The second instance states that a relative index is safe if it is a member of the boundary regions for a grid and if its predecessors in both dimensions are safe. Appendix C provides a proof-sketch of the soundness of enforcing safe indexing via *Safe*.

The Laplace operator of the introductory example is desugared by the `fun` macro into the code shown in Figure 3(a) with the type shown in Figure 3(b). The *Safe* type constraints of *laplace* thus encode its relative access pattern and constrain the grid's boundary regions such that the grid *at least* satisfies the relative indices of the grid pattern.

$$
\begin{aligned}
laplace\ g = \quad & (index2D\ (Neg\ (S\ Z), Pos\ Z)\ (-1,0)\ g) \\
+\ & (index2D\ (Pos\ (S\ Z), Pos\ Z)\ (1,0)\ g) \\
+\ & (index2D\ (Pos\ Z, Pos\ (S\ Z))\ (0,1)\ g) \\
+\ & (index2D\ (Pos\ Z, Neg\ (S\ Z))\ (0,-1)\ g) \\
-\ & 4*(index2D\ (Pos\ Z, Pos\ Z)\ (0,0)\ g)
\end{aligned}
$$

> (*Safe* (*IntT* (*Neg* (*S Z*)), *IntT* (*Pos Z*)) *b*,
> *Safe* (*IntT* (*Pos* (*S Z*)), *IntT* (*Pos Z*)) *b*,
> *Safe* (*IntT* (*Pos Z*), *IntT* (*Pos* (*S Z*))) *b*,
> *Safe* (*IntT* (*Pos Z*), *IntT* (*Neg* (*S Z*))) *b*,
> *Safe* (*IntT* (*Pos Z*), *IntT* (*Pos Z*)) *b*, *Num a*)
> ⇒ *Grid* (*Dim d* :* *Dim d′*) *b dyn a → a*

(a) Code   (b) Type

Figure 3: Desugared Laplace example

3.7   Applying Stencil Functions to Grids

In the Laplace example, the *laplace* stencil function is applied to a grid using *runA*. The *runA* function is overloaded on the type parameter of a grid which encodes whether the grid's boundary is static or dynamic, and is defined by the type class:

> **class** *RunGridA dyn* **where**
>   *runA* :: (*Grid d b dyn a → a*) → *Grid d b dyn a → Grid d b dyn a*

The instance for *Dynamic* applies the stencil function to a grid and then, from the *BoundaryList* of the parameter grid, recomputes the boundary elements. The instance for *Static* applies the stencil function but preserves the boundary elements as they do not require recomputation from the updated grid values.

A non-overloaded operation, called *run*, is also provided which applies stencil functions which change the element type of a grid, with type:

> *run* :: (*IArray UArray y*) ⇒ (*Grid d b dyn x → y*) → *Grid d b dyn x → Grid d Nil Static y*

The element type of the parameter grid differs from the element type of the return grid therefore the boundary information of the parameter grid must be discarded as the boundary elements are of type *x* but the grid elements are now of type *y*. The boundary information on the return grid is thus empty i.e. *Nil*.

As a theoretical aside: Ypnos grid's are *comonadic* structures, where *run* and *runA* provide the *extension* operation. The structuring of Ypnos by *comonads* was discussed in our earlier paper [17].

4   Results & Analysis

We provide a quantitative comparison of Ypnos versus Haskell with two benchmark programs of the two-dimensional Laplace operator and a Laplacian of Gaussian operator, which has a larger access pattern. All experiments were performed on an Intel Core 2 Duo 2Ghz, with 2GB DDR3 memory, under Mac OS X version 10.5.8 using GHC 7.0.1[4]. All code was compiled using the `-O2` flag for the highest level of "non-dangerous" (i.e. no-worse performance) optimisations. The Ypnos library is imported into a program via an `#include` statement, allowing whole-program compilation for better optimisation.

   All experiments are performed on a grid of size 512×512. The execution time of the whole program is measured for one iteration of the stencil computation and for 101 iterations of the stencil computation. The mean time per iteration is computed by the difference of these two results divided by a 100. The mean of ten runs is taken with all results given to four significant figures.

Laplace operator   This benchmark uses the Laplace example of Section 2.

	Haskell	Ypnos
1 iteration (s)	3.957	5.179
101 iterations (s)	6.297	7.640
Mean time per iteration (s)	0.0234	0.0246

$$\text{Ratio of mean time per iteration for Ypnos over Haskell} = \frac{(7.640 - 5.179)/100}{(6.297 - 3.957)/100} = \frac{0.0246}{0.0234} \cong 1.051$$

i.e. the Ypnos implementation is approximately 5% slower per iteration than the Haskell implementation.

Laplacian of Gaussian   The Laplacian of Gaussian operation combines Laplace and Gaussian convolutions. We use here a 5×5 Laplacian of Gaussian convolution operator with coefficients [8]:

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

	Haskell	Ypnos
1 iteration (s)	3.962	5.195
101 iterations (s)	7.521	8.662
Mean time per iteration (s)	0.0356	0.0347

$$\text{Ratio of mean time per iteration for Ypnos over Haskell} = \frac{0.0347}{0.0356} \cong 0.975$$

i.e. the Ypnos implementation is roughly 3% faster per iteration than the Haskell implementation.

Discussion   In terms of whole program execution time, Ypnos performance is worse than Haskell, mostly due to the overhead of the general grid and boundary types and the cost of constructing boundary elements from boundary definitions. However, per iteration we see that Ypnos performance is comparable to that of Haskell. In the Laplace example, performance is slightly worse, where the benefits of bounds-check elimination do not offset any overhead incurred by the Ypnos infrastructure. Given

---

[4] http://www.haskell.org/ghc/download_ghc_7_0_1

that safe indexing has an overhead of roughly $20 - 25\%$ over unsafe indexing, for the two-dimensional Laplace operator on a 512×512 grid (see Appendix A), then the overhead of Ypnos in the Laplace experiment is roughly $25 - 30\%$ per iteration. For the Laplace of Gaussians example performance is however slightly better (roughly 3%). Given intensive data access the benefits of bounds-check elimination begin to overshadow any overheads from Ypnos. Given a larger program with more stencil computations we conjecture that performance of Ypnos could considerably surpass that of Haskell's.

Whilst Ypnos incurs some overhead it provides the additional benefits of static correctness guarantees, the comparable ease of writing stencil computations compared to Haskell or some other language, and the generality of extending easily to higher-dimensional problems. We are currently in the process of trying larger benchmarks in Ypnos, such as a Navier-Stokes fluid simulator, and implementing parallel implementations of the stencil-function application operations.

5   Related Work

The benefit to performance from eliminating or reducing bounds checking is well known, first appearing in [16] where various optimisations were used to reduce the cost of bounds checking in loops. There has been research on analysis and transformation techniques for eliminating bounds checks in various languages, such as in Java [3, 26]. The idea of using types to ensure safety of array operations thus allowing array bounds checking to be eliminated is also not new. In Xi's thesis, *Dependent Types in Practical Programming*, dependently-typed array operations enforce safety of array indexing to allow bounds-check elimination in a general setting [27]. The approach in Ypnos is similar, but using the more lightweight approach of GADTs. Recent work by Swierstra and Altenkirch has modelled distributed array access in the dependently-typed language Agda [25] with safe indexing. Sheard et. al provide a good discussion of the relation between GADTs and dependent-types [23].

The approach of using GADTs, type families, and classes to encode invariants of a DSL is not novel. For example, Guillemette and Monnier implement a compiler for System F inside Haskell using GADTs for a typed-representation of System F terms [6]. Various transformations on the typed-terms are mechanised via type families, such as substitution and CPS conversion. Our approach is similar, although we do not use GADTs to construct typed syntax trees, instead our embedding is *shallow*, where Ypnos terms are either Haskell expressions or are desugared into Haskell expressions, in the case of grid patterns and boundary definitions. Sackman and Eisenbach use GADTs along with type classes to encode and enforce invariants of DSLs at the type-level [21].

In [11], type classes and regular ADTs are used to encode type-level natural numbers and lists, instead of GADTs. Ypnos could have used a similar scheme but we found GADTs to be more concise, clearer, and easier to write functions over. The class/ADT approach requires any function on the data types to be written via a type class with each pattern matching case encoded as a class instance.

We know of two approaches to stencil programming in Haskell that are closely related to Ypnos: PASTHA [13] and Repa [10, 14].

Repa provides a library of higher-order functions for programming parallel regular, multi-dimensional arrays in Haskell [10]. Repa allows *shape polymorphic* array operations via a type-level representation of shape, or dimensionality, that is somewhat similar to the dimensionality types of Ypnos, although Repa does not name its dimensions (the reason for named dimensions in Ypnos is discussed in Section 6).

Latest work on Repa [14] provides a data type for describing stencils in terms of stencil size and a function from indices within the stencil's range to values. The current implementation allows a constant or wrapped boundary to be specified which permits a bounds-check free implementation of a stencil

application function. Ypnos allows more fine-grained boundary behaviour than the current Repa implementation via its flexible boundary definitions and type-level safety encoding. However, it seems likely Repa will be extended with further boundary options in the future. Whilst the Repa implementation is currently specialised to two-dimensional arrays and stencils, Ypnos supports higher-dimensionality stencils via the grid pattern syntax, which is also easy to read and write. In the future, Repa may prove a useful implementation platform for parallel Ypnos programs due to Repa's very good parallel support.

PASTHA is a similar library for parallelising stencil computations in Haskell [13]. Stencils in PASTHA are described via a data structure and are restricted to relative indexing in two spatial dimensions but can also index the elements of previous iterations. The approach of Ypnos is much more general and provides more static guarantees about correctness.

There are many other languages designed for fast and parallel array programming such as Chapel [2], Titanium [30], and Single Assignment C (SAC) [22]. Ypnos differs from these in that it is sufficiently restricted to a problem domain, and sufficiently expressive within that problem domain, that all information required for guaranteeing safety, optimisation, and parallelisation is decidable and available statically without the need for complex compiler analysis and transformation.

Titanium, a parallel dialect of Java developed for over a decade, provides support for stencil computations via array loop constructs (`foreach`) and operations on indices (such as translations) [30]. Bounds-checking can be eliminated in some cases via compiler analyses and transformation. Any checks not eliminated provide runtime out-of-bounds errors. A special compilation mode allows bounds-checks to be removed entirely [30], which, whilst a useful software engineering technique for production code, does not improve the robustness or inherent efficiency of the language.

There has been much work on the optimisation of stencil computations particularly in the context of optimising cache performance (see for example [9]) and in the context of parallel implementations [5, 12]. Ypnos does not currently use any such optimisation techniques but could certainly be improved by the use of more sophisticated implementations of the stencil application functions. Various optimisation techniques such a loop tiling, skewing, or specialised data layouts, could be tuned from the symbolic information about access patterns statically provided by grid patterns.

6   Further Work

The first Ypnos paper [17] discussed Ypnos's *implementation agnosticism*, where the implementation of the language constructs is parameterisable, allowing implementations tailored to different architectures and execution strategies. In this paper we did not discuss back-end parameterisability, but instead used a single implementation for unboxed arrays. Currently we are in the process of generalising the typed approach of this paper to a parameterisable back-end. We discuss briefly here other areas of further work.

Slices   One of the main reasons that Ypnos was designed with named dimensions, e.g. *X*, *Y*, etc. was to ease multi-dimensional programming. Named dimensions are akin to using records to provide *named* rather than *positional* parameters to a function or procedure in a language.

Further work is to add array *slicing* operations to Ypnos, for which the named dimension are particularly useful. Slicing can be added to Ypnos by varying the behaviour of grid patterns depending on the dimensionality of the grid matched upon. For example, consider the hypothetical type of the following function which uses a one-dimensional grid pattern on the *X* dimension:

```
foo :: Grid (X :* d) b dyn a -> (Grid d b dyn a, Grid d b dyn a, Grid d b dyn a)
foo = [fun| X:| l @c r | -> (l, c, r) |]
```

Applying `foo` to a grid of type *Grid* (*Dim X :* *Dim Y*) *b dyn a* would yield a triple of successive grid *slices* in the *Y* dimension; applying `foo` to a one-dimensional grid of type *Grid X b dyn a* would yield a triple of single *a* values, requiring a unit for the dimension tensor e.g. *NoDim* such that:

$$NoDim :^* Dim\ X \ = \ Dim\ X \ = \ Dim\ X :^* NoDim$$

Additionally, *Grid NoDim b dyn a = a*. Named dimensions allow `foo` to be applied to a grid of the type *Grid* (*Dim Y :* *Dim X*) *b dyn a* yielding the same slicing on the *X* dimension.

Grid slicing eases higher-dimensional programming as grid patterns need not match all dimensions at once, but could slice a grid into lower-dimensional subgrids to be operated on by further stencil functions.

Inductive Tuples   Ypnos is not fully general in the dimensionality of its grids due to the non-inductive nature of Haskell tuple types. The implementation specifies operations usually up to four dimensions, however it is conceivable that some applications might require higher-dimensionality.

Inductively defined tuples, essentially lists, could be used as indices to provide a fully general implementation, however such a scheme is inefficient. In preliminary experiments we found that indexing with even just a two-element list took roughly four times as long per iteration than indexing with a tuple of two elements for a two-dimensional Laplace operation on a 512×512 grid (see Appendix A for details).

An ideal system would use an inductively defined implementation with a compiler transformation mapping inductive lists to fixed-size tuple types in the compiled Haskell code.

Errors   A well-known problem with building embedded DSLs in statically-typed languages is the generation of type errors which are lengthy and confusing for the end-user. In Ypnos, if a stencil function is applied to a grid which does not have sufficient boundary regions for safe indexing then a type error is produced. Unfortunately this type error is long, confusing, and intimidating.

A potential remedy for such problems could be to allow error messages from a compiler to be passed to a pre-processor function before they are shown to a user. For example, GHC could be extended to allow a function to be specified which, in the event of a type-error, is passed an AST of the error message, returning a more helpful message to the end-user.

7   Conclusion and Reflection

In our work on Ypnos our slogan has been: to develop languages to improve *the four "Rs"* of programs: *reading*, *(w)riting*, *reasoning*, and *running*. That is, to improve the readability of programs in the language, to improve the ease with which a program can be expressed in the language, to increase the ability to reason about programs, and to provide an efficient implementation. In a general purpose language it is harder to excel in all four areas at once. Well-designed DSLs however can simultaneously provide problem specific expressivity, improve programmer productivity, provide stronger reasoning, and provide more appropriate optimisation than general-purpose languages [24].

In Ypnos, problem specific expressivity is provided by grid patterns and boundary definitions. General array indexing constructions, as found in most languages, have low information-content with respect to program properties. Program properties relating to indexing must often be inferred via analysis, which may be undecidable in general. Grid patterns and boundary definitions provide a restricted but information-rich method of indexing. We believe that Ypnos's syntax eases *reading* and *writing* of stencil computations, although this is difficult to measure objectively. It is however easier to quantify the effect on *reasoning* and *running*. This paper has shown that grid patterns and boundary regions provide enough

information to statically prove safety of Ypnos programs, which also permits optimisation via bounds-check elimination. More complex optimisations, such as tiling, skewing, and cache sensitive data-layout, could be performed since grid patterns provide decidable static information about array access.

Other embedded DSLs have used the type system of the host language to enforce invariants of the DSL (of which a few were mentioned in Section 5). In our approach GADTs were used to lift data-level information to the type-level; type families were used to manipulate and compose such information; and type classes were used to enforce language properties/invariants based on this information.

One lesson learnt was in leveraging the semantics of type classes. A first attempt at encoding safety for Ypnos in Haskell used a special stencil data-type for grid patterns that included all relative index information in its type. Safety was then enforced by unification between this stencil type and a boundary type. However, since there is no particular order in which boundary definitions should be defined, a type-level normalisation algorithm, essentially a kind of sorting, was required for unification of boundary and stencil types. This approach was extremely complicated and error-prone. Type class constraints proved to be much more useful in the end, as they comprise a conjunction of constraints that is implicitly associative and commutative, thus ordering is not relevant. Safety in Ypnos is fortunately easily expressed via a conjunction of individual safety constraints (as shown in Appendix C). Encoding disjunctions of invariants would be more difficult and is not something we have experimented with or required yet.

Another lesson learnt is that a working type-level encoding of a language invariant does not imply soundness of the encoding. For a time we had a type-level encoding of safety which appeared correct; it was simple and elegant, rejected the programs we thought it should reject, and accepted the programs we thought it should accept. Unfortunately it was unsound, accepting some programs that should have been rejected. Overconfidence in the infrastructure meant it was a while before this unsoundness was discovered. Writing a proof of soundness beforehand is preferable, and may even give an indication of how to implement a type-level encoding, as does the soundness proof-sketch in Appendix C.

There are certainly many more areas of programming that could benefit from DSLs. *Structured grids*, or *stencil computations*, as discussed in this paper, are amongst a number of programming patterns that have been identified as key motifs in parallel programming [1]. We have already begun considering extensions to Ypnos to support *unstructured grids*, also called *meshes*. We would be interested to see if other restricted syntactic forms such as grid patterns could be invented for other programming patterns.

Acknowledgments

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams & Katherine A. Yelick (2006): *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[2] R.F. Barrett, P.C. Roth & S.W. Poole (2007): *Finite Difference Stencils Implemented Using Chapel. Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122* .

[3] R. Bodík, R. Gupta & V. Sarkar (2000): *ABCD: Eliminating Array Bounds Checks on Demand*. *ACM SIGPLAN Notices* 35(5), pp. 321–333, doi:10.1145/358438.349342.

[4] Manuel M. T. Chakravarty, Gabriele Keller & Simon Peyton Jones (2005): *Associated type synonyms*. In: *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ACM, New York, NY, USA, pp. 241–253, doi:10.1145/1086365.1086397.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf & K. Yelick (2008): *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, pp. 1–12.

[6] L.J. Guillemette & S. Monnier (2008): *A type-preserving compiler in Haskell*. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ACM, pp. 75–86, doi:10.1145/1411203.1411218.

[7] C.V. Hall, K. Hammond, S.L. Peyton Jones & P.L. Wadler (1996): *Type classes in Haskell*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18(2), pp. 109–138, doi:10.1145/227699.227700.

[8] R. Jain, R. Kasturi & B.G. Schunck (1995): *Machine vision*. 5, McGraw-Hill New York.

[9] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf & K. Yelick (2006): *Implicit and explicit optimizations for stencil computations*. In: *Proceedings of the 2006 workshop on Memory system performance and correctness*, ACM, pp. 51–60, doi:10.1145/1178597.1178605.

[10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones & Ben Lippmeier (2010): *Regular, shape-polymorphic, parallel arrays in Haskell*. ICFP '10, ACM, New York, NY, USA, pp. 261–272, doi:10.1145/1863543.1863582.

[11] Oleg Kiselyov, Ralf Lämmel & Keean Schupke (2004): *Strongly typed heterogeneous collections*. Haskell '04, ACM, New York, NY, USA, pp. 96–107, doi:10.1145/1017472.1017488.

[12] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev & P. Sadayappan (2007): *Effective Automatic Parallelization of Stencil Computations*. *ACM Sigplan Notices* 42(6), pp. 235–244, doi:10.1145/1250734.1250761.

[13] Michael Lesniak (2010): *PASTHA: parallelizing stencil calculations in Haskell*. In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10, ACM, New York, NY, USA, pp. 5–14, doi:10.1145/1708046.1708052.

[14] Ben Lippmeier, Gabriele Keller, & Simon Peyton Jones (2011): *Efficient Parallel Stencil Convolution in Haskell*. Available at `http://www.cse.unsw.edu.au/~benl/papers/stencil/stencil-icfp2011-sub.pdf`. Draft manuscript. Retrieved April, 2011.

[15] Geoffrey Mainland (2007): *Why it's Nice to be Quoted: Quasiquoting for Haskell*. In: *Haskell '07*, ACM, New York, NY, USA, pp. 73–82, doi:10.1145/1291201.1291211.

[16] V. Markstein, J. Cocke & P. Markstein (1982): *Optimization of range checking*. In: *ACM SIGPLAN Notices*, 17, ACM, pp. 114–119.

[17] Dominic Orchard, Max Bolingbroke & Alan Mycroft (2010): *Ypnos: Declarative, Parallel Structured Grid Programming*. In: *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, ACM, NY, USA, pp. 15–24, doi:10.1145/1708046.1708053.

[18] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich & Geoffrey Washburn (2006): *Simple unification-based type inference for GADTs*. In: *Proceedings of ACM SIGPLAN international conference on Functional programming*, ICFP '06, ACM, New York, NY, USA, pp. 50–61, doi:10.1145/1159803.1159811.

[19] Simon Peyton Jones, Geoffrey Washburn & Stephanie Weirich (July 2004): *Wobbly Types: Type Inference for Generalised Algebraic Data Types*. Technical Report.

[20] Simon Peyton Jones et al. (2003): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

[21] M. Sackman & S. Eisenbach: *Safely Speaking in Tongues*. In: *Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications LDTA 2009*, p. 34.

[22] Sven-Bodo Scholz (2003): *Single Assignment C: efficient support for high-level array operations in a functional setting*. Journal of Functional Programming 13(6), pp. 1005–1059, doi:10.1017/S0956796802004458.

[23] T. Sheard, J. Hook & N. Linger: *GADTs + extensible kinds = dependent programming.* `http://www.cs.pdx.edu/~sheard/papers/GADT+ExtKinds.ps` .

[24] Don Stewart (2009): *Domain Specific Languages for Domain Specific Problems*. In: *Workshop on Non-Traditional Programming Models for High-Performance Computing, LACSS*.

[25] W. Swierstra & T. Altenkirch (2008): *Dependent types for distributed arrays*. In: *Proceedings of the Ninth Symposium on Trends in Functional Programming*, Citeseer.

[26] T. Würthinger, C. Wimmer & H. Mössenböck (2007): *Array bounds check elimination for the Java HotSpot(TM) client compiler*. In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, ACM, pp. 125–133, doi:10.1145/1294325.1294343.

[27] H. Xi (1998): *Dependent Types in Practical Programming*. Ph.D. thesis, Carnegie Mellon University.

[28] H. Xi & F. Pfenning (1999): *Dependent Types in Practical Programming*. In: *Conference Record Of The ACM Sympsosium On Principles Of Programming Languages*, 26, pp. 214–227, doi:10.1145/292540.292560.

[29] L.T. Yang & M. Guo (2006): *High-performance computing: Paradigm and Infrastructure*. John Wiley and Sons.

[30] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella & T. Wen (2007): *Parallel Languages and Compilers: Perspective from the Titanium experience*. International Journal of High Performance Computing Applications 21(3), p. 266, doi:10.1177/1094342007078449.

A   Supporting Experiments

Each experiment is set up and measured as the experiments in Section 4.

Comparing Safe & Unsafe Indexing   A 2D-Laplace operation was applied to a 512×512 image using both safe and unsafe memory access to test the effect on execution performance. The results were:

	Safe indexing	Unsafe indexing
1 iteration (s)	3.982	3.952
101 iterations (s)	6.328	5.865
Mean time per iteration (s)	0.0234	0.0191

The overhead of safe indexing is approximately $\frac{0.0234}{0.0191} \cong 1.23$ i.e. approximately $20 - 25\%$.

Comparing Tuple Indices & Inductively-defined Indices   A 2D-Laplace operation was applied to a 512×512 image using an array indexed by a pair of *Int*s and an array indexed by an inductively defined *Int* list structure of length two.

	Tuples	Lists
1 iteration (s)	3.982	4.133
101 iterations (s)	6.328	13.941
Mean time per iteration (s)	0.0234	0.0981

$\frac{0.0981}{0.0234} \cong 4.19$, therefore the overhead of the list index is considerable at approximately 300%.

B   Haskell and GHC Type-System Features

Type Classes   *Type classes* in Haskell provide a mechanism for overloading, also known as *ad-hoc polymorphism* or *type-indexed functions* [7]. Consider the equality operation (==). Many types have

a well-defined notion of equality, thus an overloaded $(==)$ operator is useful, where the argument type determines the actual equality operation used. The equality type class is defined:

> **class** *Eq a* **where**
> $(==) :: a \to a \to Bool$

A type is declared a member of a class by an *instance* definition, which provides definitions of the class functions for a particular type. For example:

> **instance** *Eq Int* **where**
> $x == y = eqInt\ x\ y$

where *eqInt* is the built-in equality operation for *Int*. By this instance definition, *Int* is a member of *Eq*.

Usage of $(==)$ on polymorphically-typed arguments imposes a *type-class constraint* on the polymorphic type of the arguments. Type-class constraints enforce that a type is an instance of a class and are written on the left-hand side of a type, preceding an arrow $\Rightarrow$. For example, the following function:

> $f\ x\ y = x == y$

has type $f :: Eq\ a \Rightarrow a \to a \to Bool$, where *Eq a* is the type-class constraint that *a* be a member of *Eq*.

GADTs *Generalised algebraic data types*, or GADTs, [19, 18] have become a powerful technique for dependent-type like programming in Haskell. Consider the following algebraic data type in Haskell which encodes a simple term calculus of products and projections over some type:

> **data** *Term a* $=$ *Pair* (*Term a*) (*Term a*) | *Fst* (*Term a*) | *Snd* (*Term a*) | *Val a*

The ADT is polymorphic in the type *a* and is recursive. For each *tag* in a the definition of an ADT (e.g. *Pair*, *Fst*, etc.) a constructor is generated; e.g. for the first two data constructors of *Term*:

> $Pair :: Term\ a \to Term\ a \to Term\ a$
> $Fst :: Term\ a \to Term\ a$

Thus, every constructor returns a value of type *Term a*. We can define another ADT of values *Val* and write an evaluator from *Term* to *Val*:

> **data** *Val a* $=$ *ValPair* (*Val a*) (*Val a*) | *ValConst a*
>
> $eval :: Term\ a \to Val\ a$
> $eval\ (Pair\ x\ y) = ValPair\ (eval\ x)\ (eval\ y)$
> $eval\ (Fst\ x)\quad =$ **case** $(eval\ x)$ **of** $ValPair\ a\ b \to a;\ \_ \to error$ `"whoops"`
> $eval\ (Snd\ x)\quad =$ **case** $(eval\ x)$ **of** $ValPair\ a\ b \to b;\ \_ \to error$ `"whoops"`
> $eval\ (Val\ x)\quad = ValConst\ x$

Unfortunately, the *Term* ADT allows non-sensical terms in the *Term* calculus to be constructed, such as *Fst* (*Val* 1), therefore error handling cases must be included for *eval* on *Fst* and *Snd*.

The crucial difference between ADTs and GADTs is that GADTs specify the full type of a data constructor, allowing the result type of the constructor to have arbitrary type-parameters for the GADT's type constructor. For example, *Term* can be rewritten as the following on the left-hand side:

```
data Term t where                               eval :: Term a → a
    Pair :: Term a → Term b → Term (a,b)        eval (Pair x y) = (eval x, eval y)
    Fst  :: Term (a,b) → Term a                 eval (Fst x)    = fst (eval x)
    Snd  :: Term (a,b) → Term b                 eval (Snd x)    = snd (eval x)
    Val  :: a → Term a                          eval (Val x)    = x
```

The type parameter of *Term* encodes the type of a term as a Haskell type. Because a term type is known, *eval* can be rewritten as on the right-hand side, where a well-typed Haskell value is returned as opposed to the *Val* datatype encoding *Term* values used previously. Furthermore, *eval* cannot be applied to malformed *Term* terms, thus error handling cases are unnecessary. GADTs thus allow a form of lightweight dependent-typing by permitting types to depend on data constructors.

Type Families   *Type families* in Haskell, also called *type-indexed families of types*, are simple type-level functions providing a limited form of computation at the type-level, evaluated during type checking [4]. Type families describe a number of rewrite rules from types to types, consisting of a *head declaration* specifying the name and arity of the type family and a number of *instance declarations* defining the rewrite rules. For example, the following type family provides a projection function on pair types:

```
type family Fst t
type instance Fst (a,b) = a
```

Type families are *open* allowing further instances to be defined throughout a program or in another module. Instances of a family are therefore *unordered* thus a application of a family to an argument does not result in ordered pattern matching of the argument against the family's instances. Consequently, to preserve uniqueness of typing, type family instances must not *overlap* or at least must be *confluent* i.e. if there are two possible rewrites for a type family then the rewrites are equivalent.

Type families may be recursive, as long as the size of the type parameters of the recursive call are less than the size of the type parameters in the instance making the recursive call. For example, the following defines an append operation on the type-level list representation shown in Section 3.3:

```
type family Append x y
type instance Append Nil z = z
type instance Append (Cons x y) z = Cons x (Append y z)
```

C   Soundness Proof

The safe-indexing invariant of Ypnos is *sound* if well-typed programs cannot index undefined elements, that is, any out-of-bounds access always has a defined value. We provide a (semi-formal) proof of soundness here for two-dimensional grids, although the proof generalises easily to arbitrary dimensions.

Consider a grid of two-dimensions with finite extent $(0,0)$ to (N,M) (exclusive) and a relative index (i,j) accessed by some stencil function. We assume for this proof that i and j are both positive relative indices without loss of generality as all definitions are symmetric in the sign of relative indices.

Let V be a predicate on index ranges which denotes indices which have a defined value. As an axiom, all indices inside the grid's extent have a defined value, thus:

$$\text{(axiom)} \qquad V[0,N][0,M] \tag{1}$$

Let S be a predicate of the safe relative indices for a grid where $S(x,y)$ means (x,y) is safe. For the positive relative index (i,j), S and V are related thus:

$$V[0,N+i][0,M+j] \Leftrightarrow S(i,j) \tag{2}$$

Axiom (1) is therefore equivalent to $S(0,0)$ i.e. zero-relative indices are always safe.

The range of V can be separated into a conjunction of subranges, thus we can express (1) as:

$$
\begin{aligned}
& V[0,N][0,M] \\
\wedge\ & V[0,N][M,M+j-1] \\
\wedge\ & V[N,N+i-1][0,M] \\
\wedge\ & V[N,N+i][M,M+j]
\end{aligned}
\quad \Rightarrow \quad S(i,j)
\tag{3}
$$

(Note: by (1) we could eliminate the first conjunct) Since logical conjunction is *involutive* (i.e. $A \wedge A = A$) we can overlap the regions of V, thus we could rewrite (3) as:

$$
\begin{aligned}
& V[0,N+i-1][0,M+j] \\
\wedge\ & V[0,N+i][0,M+j-1] \\
\wedge\ & V[N+i,N+i][M+j,M+j]
\end{aligned}
\quad \Rightarrow \quad S(i,j)
\tag{4}
$$

Now consider another predicate of defined boundary regions where $B(x,y)$ means that the boundary region (x,y) is defined. The boundary region predicate B is related to V in the following way:

$$
\begin{aligned}
B(+x,+y) &\Rightarrow V[N+x,N+x][M+y,M+y] & B(+x,*) &\Rightarrow V[N+x,N+x][0,M] \\
B(+x,-y) &\Rightarrow V[N+x,N+x][-y,-y] & B(*\mathrm{v},+y) &\Rightarrow V[0,N][M+y,M+y] \\
B(-x,+y) &\Rightarrow V[-x,-x][M+y,M+y] & B(-x,*\mathrm{v}) &\Rightarrow V[-x,-x][0,M] \\
B(-x,-y) &\Rightarrow V[-x,-x][-y,-y] & B(*\mathrm{v},-y) &\Rightarrow V[0,N][-y,-y]
\end{aligned}
\tag{5}
$$

Boundary regions thus map to a single defined index e.g. $V[N+x,N+x][N+y,N+y]$, or to a range over the whole inner extent in one dimension with the other dimension fixed at a single index outside of the extent e.g. $V[N+x,N+x][0,M]$ or $V[0,N][-y,-y]$.

By (5) we can replace $V[N+i,N+i][M+j,M+j]$ in (4) with $B(i,j)$, thus:

$$
\begin{aligned}
& V[0,N+i-1][0,M+j] \\
\wedge\ & V[0,N+i][0,M+j-1] \\
\wedge\ & B(i,j)
\end{aligned}
\quad \Rightarrow \quad S(i,j)
\tag{6}
$$

We can eliminate V from (6) entirely by the relation of V to S (2):

$$S(i-1,j) \wedge S(i,j-1) \wedge B(i,j) \Rightarrow S(i,j) \tag{7}$$

Equation (7) thus specifies a kind of recurrence relation on S, which will eventually reach the axiomatic base case $S(0,0)$. The definition of *Safe* is exactly the predicate S as defined by the recurrence (7) where *InBoundary* is the predicate B:

instance $(Safe\ (IntT\ (Pred\ n), IntT\ n')\ b, Safe\ (IntT\ n, IntT\ (Pred\ n'))\ b,$
$\qquad InBoundary\ (IntT\ n, IntT\ n')\ b) \Rightarrow Safe\ (IntT\ n, IntT\ n')\ b$

The axiom (1), $S(0,0)$ is satisfied by the case that all zero-relative indices are safe:

instance $Safe\ (IntT\ (Pos\ Z), IntT\ (Pos\ Z))\ b$

By induction *Pred* reduces relative indices towards zero, acting as the minus operation in (7), and is symmetrical in its treatment of negative and positive relative indices.

Safe thus provides a sound encoding of safe-indexing for grids. $\square$