Verifying Spatial Properties of Stencil Computations

Abstract

Stencil computations are a common programming pattern in numerical code and typically form the core of graphics and scientific computing applications. However, they are often complex and unwieldy, involving substantial fine-grained manipulation of array indices across multiple arrays and dimensions. This requires careful testing, which is almost always done manually and then discarded. In this paper, we propose an alternate approach: a specification language and automated tool for verifying the spatial behaviour of stencils. The specification language is based on the hypothesis that stencil computations read from arrays with a regular, fixed pattern which can be captured by a simple set of abstract combinators. These specifications can be embedded as annotations in source code, against which code can be checked for conformance. For legacy code, spatial specifications can be inferred, providing documentation and aiding future software maintenance. We evaluate our specification language and verification tool against a corpus of numerical Fortran code ($\sim$ 1 million lines) for which we generate $\sim$60,000 specifications, showing the vast majority of stencil computations indeed have a simple, regular, static shape.

1. Introduction

Stencils are a ubiquitous programming pattern, common in scientific and numerical computing applications. Informally, a stencil computation computes an array whose value at each index i is calculated from a *neighbourhood* of values around i in some input array(s), *e.g.*, the Game of Life, convolutions in image processing, approximations to differential equations. For example, the following computes the one-dimensional discrete Laplace transform (an approximation to a derivative) in Fortran:

```
1  do i = 1, (n-1)
2        b(i) = a(i-1) - 2*a(i) + a(i+1)
3  end do
```


Values `b(i)` are calculated from a neighbourhood of elements around `i` in the input array `a`. In this example, the access pattern is simple and easily understood. More complex stencil computations are more prone to errors from simple lexical mistakes. For example, the following is a small snippet from a Navier-Stokes fluid simulator (based on Griebel et al. [16]) in which two arrays are read with different data access patterns, across two dimensions.

```
20  du2dx = ((u(i,j)+u(i+1,j))*(u(i,j)+u(i+1,j))+   &
21    gamma*abs(u(i,j)+u(i+1,j))*(u(i,j)-u(i+1,j))- &
22    (u(i-1,j)+u(i,j))*(u(i-1,j)+u(i,j))-          &
23    gamma*abs(u(i-1,j)+u(i,j))*(u(i-1,j)-u(i,j))) &
24    /(4.0*delx)
25
26  duvdy = ((v(i,j)+v(i+1,j))*(u(i,j)+u(i,j+1))+   &
27    gamma*abs(v(i,j)+v(i+1,j))*(u(i,j)-u(i,j+1))- &
28    (v(i,j-1)+v(i+1,j-1))*(u(i,j-1)+u(i,j))-      &
29    gamma*abs(v(i,j-1)+v(i+1,j-1))*(u(i,j-1)-     &
30    u(i,j))) / (4.0*dely)
31
32  laplu = (u(i+1,j)-2.0*u(i,j)+u(i-1,j))/delx/delx+ &
33    (u(i,j+1)-2.0*u(i,j)+u(i,j-1))/dely/dely
34
35  f(i,j) = u(i,j)+del_t*(laplu/Re-du2dx-duvdy)
```

The access pattern is much harder to understand than the one-dimensional Laplace. The miasma of indexing expressions of the form `var(i±a, j±b)` is not only hard to read, but is prone to simple textual input mistakes, *e.g.*, swapping `-` and `+`, missing an indexing term, or transforming the wrong variable *e.g.* `(i+1,j)` instead of `(i,j+1)`.

In practice, the typical development procedure for complex stencil computations involves some ad hoc testing to ensure that no transcription mistakes have been made (*e.g.*, by visual inspections, or comparison against an manufactured or analytical solutions [14]). Typically such testing is discarded once the code is seen to be correct.

This is not the only information that is often discarded. The "shape" of the indexing pattern is usually the result of choices made in the numerical-analysis procedure to convert some continuous equations into a discrete approximation. Rarely are these decisions captured in the source code, yet the shape of access is usually uniform and is has a perspicuous and concise description in numerical analysis literature *e.g.*, "*centered in space, to a depth of 1*" referring to indexing terms `a(i)`, `a(i-1)` and `a(i+1)`.

To support the development of correct stencil computations, we propose a simple, abstract specification language

for the data access pattern of stencils. The shape of the Laplace example stencil is specified in our language as:

```
!=  stencil centered(depth=1, dim=1) :: a
```

That is, `a` is accessed with a symmetrical pattern ("centered") to a depth of one in each direction in its first dimension. The Navier-Stokes example has a shape specified as:

```
!= stencil centered(depth=1,dim=1)*pointed(dim=2)
     ↪   + centered(depth=1,dim=2)*pointed(dim=1) :: u

!= stencil forward(depth=1,dim=1)
     ↪   * backward(depth=1,dim=2) :: v
```

The specification requires that, over the whole fragment, `u` is accessed with a centered pattern to depth of 1 in both dimensions (this is known as the *five-point stencil*) and `v` is accessed in a neighbourhood bounded forwards to depth of 1 in the first dimension and backward to a depth of 1 in the second dimension. The specification is relatively small and much more abstract compared to the source code.

We provide a verification tool which can check the spatial correctness of stencil computations against a specification. Furthermore, the tool can automatically infer specifications and insert these as comments into the source code. The main features and benefits of our approach are as follows:

- The primary use case is that a programmer writes a specification first before the stencil code. Our tool then checks conformance between the two. This specify-and-check approach reduces testing effort and future-proofs against bugs introduced during refactoring and maintenance.
- A specification concisely captures the access pattern of a stencil and turns it into documentation. The inference procedure efficiently produces specifications with no programmer intervention, automatically inserting specifications at appropriate places in the source code.
- Our specification format is deliberately abstract, with a small number of combinators that *do not involve any indexing expressions*, *e.g.* `a(i+1,j-1)`. This contrasts with other specification approaches, *e.g.*, deductive verification tools such as ACSL, where specifications must also be in terms of array-indexing expressions. Therefore, any low-level mistakes that could be made whilst programming complex indexing code could also be made when writing its specification. Our specifications are more abstract and lightweight with the aim of aiding adoption by scientists and preventing indexing errors.

Our verification tool does not target a class of bugs that can be detected automatically (*push-button verification*). Instead, stencil computation bugs must be identified relative to a specification of the intended access pattern.

In this paper, we make the following contributions:

- We introduce a specification language for stencils that captures common forms of data access patterns (§ 2);
- We detail checking and inference algorithms for our specifications (§ 5), derived from a denotational model for the specification language (§ 3);
- We provide an implementation of our approach as an extension to CamFort [2], an open-source program analysis tool for Fortran;
- We report on a quantitative study of stencil computations on a corpus of numerical Fortran program (§ 6), totalling one million lines of code. Our tool identifies and infers specifications for roughly 60,000 stencil computations in the corpus. Approximately 5,000 of the stencils we found are non-trivial, corresponding to code which is a possible source of errors. This validates our hypothesis that the majority of stencil computations have a regular shape, and validates the design of our language in its capability to capture many core patterns.
- We give a verification case study of a stencil computation commonly used in scientific computing, simulating programming errors (§ 6.1). Our approach detects all possible simulated programming errors for this problem.

2. Stencil specifications

Our specification language is based on the hypothesis that most forms of array access in numerical code have a fixed, statically-determined access pattern. For example, the *five-point stencil* on a two-dimensional array reads from array indices (i,j), $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, and $(i,j+1)$ for all i,j within the inner boundary of the array (to avoid out-of-bounds access at the edges). We revisit this hypothesis in § 6 with the inference of such regular stencil patterns on our corpus of numerical programs.

We give various definitions used in the paper then § 2.1 outlines the syntax of stencil specifications with an informal introduction. § 3 gives a semantic model via a set-based interpretation which makes concrete the meaning of specifications. § 4 then provides a syntactic equational theory for specifications which is sound with respect to the model.

Notation For the target language, e ranges over expressions (may be impure) and v over its (imperative) variables.

Definition 1 (Induction variable)**.** An integer variable is an *induction variable* if it is the control variable of a "`for`" loop incremented by 1 per iteration. A variable is interpreted as an induction variable only within the scope of the loop body. Throughout, i,j,k range over induction variables.

Definition 2 (Array subscripts and indices)**.** An *array subscript*, denoted $a(\bar{e})$, is an expression which reads from an array a at an *index* specified by a comma-separated sequence of integer expressions denoted $\bar{e}$ or in expanded form as $(e_1,\ldots,e_n)$. An index e_i is called *relative* if the expression involves an induction variable. An *absolute index* is a constant integer expression relative to the enclosing loop.

specification ::= *regionDec* | *specDec*
specDec ::= `stencil` *spec* `::` v
regionDec ::= `region ::` *rvar* `=` *region*

spec ::= [*mult*,] [*approx*,] *region*
mult ::= `readOnce`
approx ::= `atMost` | `atLeast`
region ::= *rvar*
| `pointed` (`dim=`$\mathbb{N}_{>0}$)
| `forward` (`depth=`$\mathbb{N}_{>0}$, `dim=`$\mathbb{N}_{>0}$ [, `nonpointed`])
| `backward` (`depth=`$\mathbb{N}_{>0}$, `dim=`$\mathbb{N}_{>0}$ [, `nonpointed`])
| `centered` (`depth=`$\mathbb{N}_{>0}$, `dim=`$\mathbb{N}_{>0}$ [, `nonpointed`])
| *region* `+` *region* | *region* `*` *region*
rvar ::= [`a-z A-Z 0-9`]+

Figure 1. Specification syntax (EBNF grammar)

Definition 3 (Neighbourhood index)**.** For an array subscript $a(\bar{e})$ an index $e \in \bar{e}$ is a *neighbourhood index* if e is of the form $e \equiv i$, $e \equiv i + c$, or $e \equiv i - c$, where c is an integer constant. That is, a neighbourhood index is a constant translation of an induction variable. (The relation $\equiv$ identifies terms up-to commutativity of $+$ and the inverse relationship of $+$ and $-$ *e.g.*, $(-b) + i \equiv i - b$).

Definition 4 (Stencil computations)**.** Let a be an array of dimensionality n and $\bar{b}$ a collection of arrays of arbitrary (possibly differing) dimensionalities. A *stencil computation* comprises an iteration by a set I of at most n induction variables over a subset of the index space of a. Each iteration determines the elements of a by an assignment $a(\bar{e}) = e_r$ where each $e \in \bar{e}$ is either a neighbourhood or constant index. For all $b \in \bar{b}$ and array subscripts $b(\bar{e}')$ that flow to e_r, each $e' \in \bar{e}'$ is either a neighbourhood or absolute index.

We refer to the assignment $a(\bar{e}) = e_r$ and all the associated statements with dataflow to e_r as the *stencil kernel*.

For now we informally describe what it means for a stencil kernel to be consistent with a specification. This will be formalised in § 3 and algorithmically described in and § 5. Specifications are associated to a stencil kernel and are declared for one or more array variables $\bar{v}$ within that kernel. A stencil specification defines a *region* associated to an array variable a against which a stencil kernel should be *consistent*. The region defined by a specification is an n-dimensional rectilinear shape in a discrete space crossing the origin (which represents a collection of indices which are just induction variables). Consistency is then a two way requirement. All array subscripts on a contributing to the kernel should be consistent with the region. Conversely, all parts of the region should have an index corresponding it.

2.1 Specification syntax

Fig. 1 gives the syntax of stencil specifications, which is detailed below. The entry point is the *specification* production which splits into either a *region declaration* or a *specification declaration*. Regions comprise *region constants* which are combined via region operators `+` and `*`.

Region constants specify a finite contiguous region in a single dimension relative to the origin and are either `pointed`, `forward`, `backward`, or `centered`. The region names are inspired by the scientific terminology, *e.g.* the standard explicit method for approximating PDEs is known as the *Forward Time, Centered Space* (FTCS) scheme [13].

Each region constant has a dimension identifier d given as a positive natural number. Each constant except `pointed` has a depth parameter n given as a positive natural number; `pointed` regions implicitly have a depth of 0.

A `forward` region of depth n specifies a contiguous region in dimension d starting at the origin. This corresponds to specifying neighbourhood indices in dimension d ranging from i to $i + n$ for some induction variable i. Similarly, a `backward` region of depth n corresponds to contiguous indices from i to $i - n$ and `centered` of depth n from $i - n$ to $i + n$. A `pointed` stencil specifies a neighbour index i. For example, the following shows four specifications with four consistent stencil kernels reading from arrays `a`, `b`, `c` and `d`:

```
1  != stencil forward(depth=2, dim=1) :: a
2  e(i, 0) = a(i, 0) + a(i+1, 0) + a(i+2, 0)
3  != stencil backward(depth=2, dim=1) :: b
4  e(i, j) = b(i, j) + b(i-1, j) + b(i-2, j)
5  != stencil centered(depth=1, dim=2) :: c
6  e(i, j) = (c(j-1) + c(j) + c(j+1))/3.0
7  != stencil pointed(dim=3) :: d
8  e(i, j) = d(0, 0, i)
```

Not every dimension needs to be specified, *e.g.*, specifications on lines 1, 3, and 7 leave some dimensions unspecified.

The `forward`, `backward`, and `centered` regions may all have an additional attribute `nonpointed` which marks absence of the origin. For example, the following is a `nonpointed backward` stencil

```
1  != stencil backward(depth=2, dim=1, nonpointed) :: a
2  b(i) = a(i-1) + 10*a(i-2)
```

Combining regions The region operators `+` and `*` respectively combine regions disjunctively and conjunctively.

The conjunction of two regions $r*s$ means that any indices in the specified code must be consistent with both r and s simultaneously. Dually, for every pair of components in the regions r and s there must be an index consistent with both. For example, the following *nine-point stencil* has a specification given by the product of two `centered` regions in each dimension:

```
1  x = a(i, j)   + a(i-1, j)   + a(i+1, j)
2  y = a(i, j-1) + a(i-1, j-1) + a(i+1, j-1)
3  z = a(i, j+1) + a(i-1, j+1) + a(i+1, j+1)
4  != stencil centered(depth=1, dim=1) *
          ↪  centered(depth=1, dim=2) :: a
5  b(i, j) = (x + y + z) / 9.0
```

This pattern is common in image convolution applications. The specification ranges over the values that flow to the array subscript on the left-hand side, and so ranges over the

intermediate assignments to `x`, `y`, and `z`. Each index in the code is consistent with both specifications simultaneously, *e.g.*, `a(i-1, j+1)` is within the centered region in dimension 1 and the centered region in dimension 2.

The product $r*s$ can also be thought of as a bounding box over the two regions r and s.

The disjunction of two regions $r+s$ means that any indices in the specified code must be consistent with either of r or s. Dually, every part of region r must have a corresponding index and similarly for s independently. For example, the following gives the specification of a five-point stencil which is the sum of two compound `pointed` and `centered` regions in each dimension:

```
1  != stencil centered(depth=1, dim=1)*pointed(dim=2)
   ↪   + centered(depth=1, dim=2)*pointed(dim=1) :: a
2  b(i,j) = -4*a(i,j) + a(i-1,j) + a(i+1,j) &
3                     + a(i,j-1) + a(i,j+1)
```

Here the left-hand side of + says that when the second dimension (induction variable j) is fixed at the origin, the first dimension (induction variable i) accesses the immediate vicinity of the origin (to depth of one). The right hand side of + is similar but the dimensions are reversed. This reflects the symmetry under rotation of the five-point stencil.

Region declarations and variables Region specifications can be assigned to region variables via region declarations. For example, the shape of a "*Roberts cross*" edge-detection convolution [12] can be stated:

```
1  != region :: r1 = forward(depth=1, dim=1)
2  != region :: r2 = forward(depth=1, dim=2)
3  != region :: robertsCross = r1*r2
4  != stencil robertsCross :: a
```

This is useful for common stencil patterns, such as the five-point pattern, as the region can be defined once and reused.

Modifiers Region specifications can be modified by *approximation* and *multiplicity* information (in *spec* in Fig. 1).

The `readOnce` modifier enforces that no index appears more than once (that is, its multiplicity is one). For example, all of the previous examples could have `readOnce` added:

```
1  != stencil readOnce, backward(depth=2, dim=1) :: a
2  b(i+1) = a(i) + a(i-1) + a(i-2)
```

This specification would be invalid if any of the array subscripts were repeated. This modifier provides a way to rule out any accidental repetition of array subscripts. The notion is similar to that of linear types [25], where a value must be used exactly once. We opt for the more informative and easily understood name `readOnce`. This modifier is optional, so it need not be present even if the stencil is linear.

In some cases, it is useful to give a lower and/or upper bound for a stencil. This can be done using either the `atMost` or `atLeast` modifiers. This is particularly useful in situations where there is a non-contiguous stencil pattern, which cannot be expressed precisely in our syntax. For example:

```
1  != stencil atLeast, pointed(dim=1)         :: a
2  != stencil atMost, forward(depth=4, dim=1) :: a
3  b(i) = a(i) + a(i+4)
```

3. A model for specifications

We define a simple set-theoretic denotational model of the semantics of our specification language. This has a number of purposes: (1) it serves to explain the meaning of our specifications (2) it is used in the inference and checking algorithms (§ 5); (3) justifies an equational theory for specifications in the next section; and (4) can be used to guide correct implementations. The model domain is over sets of vectors which we call *index schemes*.

Definition 5. An *index scheme* is a vector of size n (an n-vector) with values drawn from $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$. Values in $\mathbb{Z}$ represent the offsets of neighbourhood indices (*i.e.*, the constant integer c in an indexing expression of the form $i \pm c$, Def. 3) and ∞ is a wild-card representing both constant indices and unspecified indices. For an index scheme u, the i^{th} element is denoted u_i. We write $(\mathbb{Z}_\infty)^n$ for the set of all n-dimensional schemes.

Index schemes abstract array subscripts. Our model consists of index schemes explaining the range of possible indexing behaviours which are consistent with the specification. The relationship between source language and index schemes is defined by schematic function, which forms part of checking and inference (§ 5).

Definition 6. Let s_I be a partial function, parameterised by a set of induction variables I, mapping source language expressions to $\mathbb{Z}_\infty$ values:

$$\mathsf{s}_I(e) = \begin{cases} c & e \equiv i + c \ \wedge \ i \in I \\ -c & e \equiv i - c \ \wedge \ i \in I \\ \infty & I \cap \mathit{FreeVariables}(e) = \emptyset \end{cases}$$

In the first two cases, expressions that are a constant offset of an induction variable (*i.e.*, neighbourhood indices) are mapped to the offset c or $-c \in \mathbb{Z}_\infty$. In the third case, if the expression is not defined in terms of an induction variable, it is mapped to $\infty \in \mathbb{Z}_\infty$. Otherwise, $\mathsf{s}_I(e)$ is undefined ($\perp$).

The partial function $\mathsf{schematic}_I$, also parameterised by a set of induction variables I, uses s_I to map from the syntax of indices to index schemes $\in (\mathbb{Z}_\infty)^n$. It is defined on the syntax of indices $\bar{e} = (e_1, ..., e_n)$ as:

$$\mathsf{schematic}_I(e_1, .., e_n) = [\mathsf{s}_I(e_1)...\mathsf{s}_I(e_n)] \ \textit{iff} \ \forall x.\mathsf{s}_I(e_x) \neq \perp$$

Thus, if each expression in an index is a neighbourhood or constant index, $\mathsf{schematic}_I$ maps it to an index scheme.

Definition 7 (Generalisation)**.** Given two schemes v, w of dimensionality n, then w is said to *generalise* v (or conversely v *specialises* w), written $v \preceq_n w$ defined:

$$v \preceq_n w \quad \Leftrightarrow \quad \forall i \in \{1, \ldots, n\}. (w_i = \infty \ \vee \ v_i = w_i)$$

where $\preceq_n$ is a reflexive, transitive relation on index schemes $\mathbb{Z}_\infty^n$. Thus, ∞ generalises any index in the same dimension.

Example 8. Consider the following array subscript to index schemes conversions:

$$\begin{array}{ll} \mathsf{schematic}_I(i, j+1) & = [0\ 1] \\ \mathsf{schematic}_I(1, j+1) & = [\infty\ 1] \\ \mathsf{schematic}_I(i*2, j) & = \bot \end{array}$$

There is one generalisation between these schemes, namely that $[0\ 1] \preceq_2 [\infty\ 1]$. The last example is undefined as its first expression is a non-neighbour relative index.

Shorthand notation We use abbreviations `fwd`, `bwd`, `cen`, and `np` for `forward`, `backward`, `centered`, and `nonpointed` respectively. Furthermore, we quantify over the absence or presence of the `nonpointed` attribute on region constants as a boolean. For example, `fwd` is syntactic sugar defined:

$$\texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, p) := \begin{cases} \texttt{forward}\,(\texttt{depth=}n, \texttt{dim=}d, \texttt{nonpointed}) & \textit{if } \neg p \\ \texttt{forward}\,(\texttt{depth=}n, \texttt{dim=}d) & \textit{if } p \end{cases}$$

or $\texttt{fwd}\,(\texttt{depth=}n, \texttt{dim=}d) := \texttt{forward}\,(\texttt{depth=}n, \texttt{dim=}d)$ and similarly for `bwd` and `cen`.

3.1 Denotational model

An interpretation function $[\![-]\!]_n$ provides the model (where n is the maximum dimensionality of the specification being modelled) mapping closed[1] specifications to sets of n-dimensional index schemes. The interpretation is overloaded on *regions* in Fig. 2 and on the top-level of a specification *spec* in Fig. 3. Various intermediate notions are used.

Definition 9. A *single-entry vector* of size n, denoted $\mathbf{J}_n^r$, is a vector where the r^{th} entry is 1 and all others are ∞, *e.g.*, $\mathbf{J}_2^1 = [1\ \infty]$ and $\mathbf{J}_3^2 = [\infty\ 1\ \infty]$. Thus, a single-entry vector is an index scheme generalising all other schemes that have 1 as their r^{th} entry, describing a neighbour offset of 1.

Definition 10. A *zeroed single-entry vector* of size n denoted $\mathbf{K}_n^r$ is a vector where the r^{th} entry is 0 and all others are ∞ *e.g.*, $\mathbf{K}_2^1 = [0\ \infty]$. Akin to single-entry vectors, zeroed single-entry vectors generalise all other schemes that have 0 as their r^{th} entry, representing indices at the "origin" in r.

The first four equations of Fig. 2 model region constants as sets of indexing schemes. We illustrate with an example.

Example 11. For a 2-dimensional stencil computation, then

$$\begin{array}{ll} & [\![\texttt{cen}\,(\texttt{depth=}2, \texttt{dim=}1)]\!]_2 \\ = & \{i\mathbf{J}_2^1 \mid i \in \{\text{-}2, ..., 2\} \backslash 0\} \cup \{\mathbf{K}_2^1 \mid \textit{true}\} \\ = & \{-2\mathbf{J}_2^1,\ -\mathbf{J}_2^1,\ \mathbf{J}_2^1,\ 2\mathbf{J}_2^1\} \cup \{\mathbf{K}_2^1\} \\ = & \{[-2\ \infty], [-1\ \infty], [0\ \infty], [1\ \infty], [2\ \infty]\} \end{array}$$

Note, ∞ is absorbing for multiplication above *e.g.* $2\infty = \infty$.

[1] That is, we assume there are no occurrences of *rvar* in a specification being modelled. Any *open* specification containing region variables can be made closed by straightforward syntactic substitution with a (closed) *region*.

$$[\![\texttt{fwd}(\texttt{depth=}k, \texttt{dim=}d, p)]\!]_n = \{\mathbf{K}_n^d \mid p\} \cup \{i\mathbf{J}_n^d \mid i \in \{1, ..., k\}\}$$
$$[\![\texttt{bwd}(\texttt{depth=}k, \texttt{dim=}d, p)]\!]_n = \{\mathbf{K}_n^d \mid p\} \cup \{i\mathbf{J}_n^d \mid i \in \{-k, ..., -1\}\}$$
$$[\![\texttt{cen}(\texttt{depth=}k, \texttt{dim=}d, p)]\!]_n = \{\mathbf{K}_n^d \mid p\} \cup \{i\mathbf{J}_n^d \mid i \in \{\text{-}k, ..., k\} \backslash 0\}$$
$$[\![\texttt{pointed}\,(\texttt{dim=}d)]\!]_n = \{\mathbf{K}_n^d\}$$
$$[\![r_1 * r_2]\!]_n = [\![r_1]\!]_n \otimes [\![r_2]\!]_n \qquad [\![r_1 \texttt{ + } r_2]\!]_n = [\![r_1]\!]_n \cup [\![r_2]\!]_n$$

Figure 2. Model of regions, $[\![-]\!]_n : \textit{region} \to \mathcal{P}(\mathbb{Z}_\infty{}^n)$

$$[\![\textit{mult}, \textit{approx}, \textit{region}]\!]_n = [\![\textit{mult}]\!]^m\ ([\![\textit{approx}]\!]^a\ [\![\textit{region}]\!]_n)$$

$$[\![\varepsilon]\!]^a = \mathsf{exact} \quad [\![\texttt{atMost}]\!]^a = \mathsf{upper} \quad [\![\texttt{atLeast}]\!]^a = \mathsf{lower}$$
$$[\![\varepsilon]\!]^m = \mathsf{once} \quad [\![\texttt{readOnce}]\!]^m = \mathsf{mult}$$

Figure 3. $[\![-]\!]_n : \textit{spec} \to \mathsf{Mult}(\mathsf{Approx}(\mathcal{P}(\mathbb{Z}_\infty{}^n)))$

The final two equations of Fig. 2 give the models of + and *. For + this is the union of the models of subterms. The model of * takes a "tensor" $\otimes$ of subterm models. The tensor is more involved. We give some informal intuition first.

$\otimes$ of models The intuition behind $\otimes$ is that it takes all possible pairs of index schemes, and treats each pair as lower and upper bounds of an n-dimensional rectangle. From these bounds, the remaining vertices of the n-dimensional rectangle are generated (like a bounding box) *e.g.*:

$$\{[1\ 2], [5\ 6]\} \otimes \{[3\ 4]\} = \{[1\ 2], [3\ 2], [1\ 4], [3\ 4], [3\ 6], [5\ 4], [5\ 6]\}$$

The tensor takes a Cartesian product of the two models, giving all pairs of schemes which are combined using the *pairwise permutation* written $\bowtie$ (defined later). Pairwise permutation generates all possible interleavings of two vectors. Some care is then taken over how this interacts with ∞. In the presence of ∞, generate all interleavings as if ∞ is an integer, then filter out vectors with ∞ at any coordinate, *e.g.*

$$\{[1\ \infty]\} \otimes \{[3\ 4]\} = \{[1\ 4], [3\ 4]\}$$

where $[1\ \infty]$ and $[3\ \infty]$ have been filtered.

Definition 12. For an n-dimensional model M, its *constrained dimensions* are the set of dimension identifiers where at least one scheme $u \in M$ has a non-∞ value in that dimension:

$$\mathit{constr}(M)_n = \bigcup_{u \in M} \{i \mid i \in \{1, \ldots, n\} \wedge u_i \neq \infty\}$$

For example, $\mathit{constr}(\{[0\ \infty], [\infty\ \infty]\})_2 = \{1\}$.

Definition 13. The tensor $\otimes_n$ of n-dimensional models is:

$$M \otimes_n N = \left\{ x \;\middle|\; \begin{array}{l} (u, v) \in M \times N, \\ i \in \{1, \ldots, 2^n\},\ x = (u \bowtie v)_i, \\ j \in (\mathit{constr}(M)_n \cup \mathit{constr}(N)_n), \\ x_j \neq \infty \end{array} \right\}$$

The second guard binds j to the constrained dimensions for M unioned with the constrained dimensions for N. That is,

every j corresponds to a dimension in which either M or N has a scheme u where $u_j \neq \infty$, and thus is constrained. The *pairwise permutation* $u \bowtie v$ builds a $2^n \times n$ matrix of all possible n-vectors generated by non-deterministically picking for each j^{th} entry either u_j or v_j *e.g.*

$$[0\ 1\ 2] \bowtie [3\ 4\ 5] = \begin{bmatrix} 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 \\ 1 & 1 & 4 & 4 & 1 & 1 & 4 & 4 \\ 2 & 5 & 2 & 5 & 2 & 5 & 2 & 5 \end{bmatrix}$$

The 2^n unique choices for $\bowtie$ on n-vectors correspond to taking all bit-strings of length n and selecting from u for 1 and v for 0. The pairwise permutation matrix is thus defined in terms of the logical matrix b where $b_{i,j}$ is the j-th bit of the integer $i-1$ as:

$$(u \bowtie v)_{i,j} = b_{i,j} u_j + \neg b_{i,j} v_j$$

The permuted schemes $x = (u \bowtie v)_i$ added to the model $M \otimes N$ are only those without ∞ in the constrained dimensions given by each model ($x_j \neq \infty$). This eliminates all permutations with ∞ in a constrained dimension position.

Model of spec Regions are modelled as sets of index schemes, that is $[\![\textit{region}]\!]_n \in \mathcal{P}((\mathbb{Z}_\infty)^n)$. Fig. 3 defines the top-level model of the *spec* syntax, which includes the multiplicity and approximation modifiers. The domain of this interpretation is $\mathsf{Mult}(\mathsf{Approx}(\mathcal{P}((\mathbb{Z}_\infty)^n)))$ where Mult and Approx are labelled variants labelling our set-based model with additional information.

Definition 14. Mult and Approx are parametric labelled variant types with injections given by their definition:

$$\mathsf{Mult}\, a = \mathsf{mult}\, a \mid \mathsf{only}\, a$$
$$\mathsf{Approx}\, a = \mathsf{exact}\, a \mid \mathsf{lower}\, a \mid \mathsf{upper}\, a \mid \mathsf{both}\, a\, a$$

e.g., lower is an injection $\mathsf{lower} : a \to \mathsf{Approx}\, a$ etc. The Mult type corresponds to the presence or absence of the `readOnce` modifier as shown in Fig. 3. The Approx type corresponds to the presence or absence of the spatial approximation modifier, with exact when there is no such modifier and lower and upper for `atLeast` and `atMost`. Later, it will be useful to model pairs of lower and upper bounds provided by both but which is not used here.

4. Equational theory

Specifications have a theory of equality, $\equiv$, providing axiomatic semantics for the language. This enables users to rewrite their specification and provides a way to automatically simplify specifications after inference (§ 5.3). Additionally, specifications have a theory of approximation, $<:$, that can be used by the users to weaken specifications for succinctness in the source.

The equality relation $\equiv$ is defined over *region* syntax which gives the following axioms:

- $+$ is idempotent, commutative, and associative;
- $*$ is commutative, associative, and distributes over $+$, *i.e.*, $R * (S + T) \equiv (R * S) + (R * T)$. Distributivity is used later to give a normal form for specifications akin to Disjunctive Normal Form (§ 5.3.3).
- (*overlapping*) Given two region constants R_1 and R_2 on the same dimension, then some combinations of region constants can be coalesced into a single region constant R_3 if R_1 completely overlaps R_2. There are five possible instantiations of the following rule $R_1 + R_2 \equiv R_3$:

$$\begin{aligned} & R_3(\texttt{depth=}n, \texttt{dim=}d, p \vee q) \\ \equiv\ & R_1(\texttt{depth=}n, \texttt{dim=}d, p)\ \texttt{+}\ R_2(\texttt{depth=}m, \texttt{dim=}d, q) \end{aligned}$$

  where $n \geq m$ and p and q range over booleans representing the absence (true) or presence (false) of `nonpointed` which are combined via disjunction. We denote the possibilities as `fwd+fwd` $\equiv$ `fwd`, `bwd+bwd` $\equiv$ `bwd`, `cen+cen` $\equiv$ `cen`, `cen+fwd` $\equiv$ `cen` and `cen+bwd` $\equiv$ `cen`.
- (*overlapping pointed*) `pointed` regions coalesce with other regions of the same dimension:

$$\begin{aligned} & \textit{region}(\texttt{depth=}n, \texttt{dim=}d) \\ \equiv\ & \textit{region}(\texttt{depth=}n, \texttt{dim=}d, p)\ \texttt{+}\ \texttt{pointed}\,(\texttt{dim=}d) \end{aligned}$$

- *(centered)* a forward and backward region of the same depth and dimension coalesces into a centered region:

$$\begin{aligned} & \texttt{centered}\,(\texttt{depth=}n, \texttt{dim=}d, p \vee q) \\ \equiv\ & \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, p)\ \texttt{+}\ \texttt{bwd}(\texttt{depth=}n, \texttt{dim=}d, q) \end{aligned}$$

- (POINT) $(R * S^{\text{np}}) + (R^{\text{np}} * S) \equiv R * S$ meaning that for the sum of products of two regions R and S, where in each component one of R or S has the non-pointed attribute denoted by R^{np} and S^{np}, the non-pointed attributes cancel to give $R * S$.
- ($*$DIM) $R^d * S^d \equiv R^d + S^d$ where R^d and S^d denote regions whose dimension is d. Thus a product of two regions on the same dimension is equal to their addition.

Stencil specifications in *spec* are considered equal when they have the same modifiers and $\equiv$ equates their regions.

Theorem 15 (Soundness of equational theory)**.**

$$\forall S, T, n. \quad S \equiv T \ \Rightarrow\ [\![S]\!]_n = [\![T]\!]_n$$

See the supplement for the proof.

Fig. 4 defines a notion of sub-specifications via the relation $<:^r$ on *region* syntax and then $<:$ on *spec* syntax inductively. Both relations are reflexive and transitive (we omit these equations). The $<:^r$ relation is congruent with respect to $+$ and $*$ and $<:$ is congruent with respect to `readOnce`, the equations of which we omit for brevity.

The (EQ) rule connects region equations to approximation; (OVER) explains the notion of spatial over-approximation via overlapping regions; (REP) allows the `readOnly` modifier that to be dropped. The (SHRINK) and (GROW) rules turn

$$\frac{m \leq n \qquad r \implies s}{\begin{array}{l}\texttt{fwd}(\texttt{depth}{=}m, \texttt{dim}{=}d, r) <:^r \texttt{fwd}(\texttt{depth}{=}n, \texttt{dim}{=}d, s) \\ \wedge\ \texttt{bwd}(\texttt{depth}{=}m, \texttt{dim}{=}d, r) <:^r \texttt{bwd}(\texttt{depth}{=}n, \texttt{dim}{=}d, s) \\ \wedge\ \texttt{cen}(\texttt{depth}{=}m, \texttt{dim}{=}d, r) <:^r \texttt{cen}(\texttt{depth}{=}n, \texttt{dim}{=}d, s) \\ \wedge\ \texttt{bwd}(\texttt{depth}{=}m, \texttt{dim}{=}d, r) <:^r \texttt{cen}(\texttt{depth}{=}n, \texttt{dim}{=}d, s) \\ \wedge\ \texttt{fwd}(\texttt{depth}{=}m, \texttt{dim}{=}d, r) <:^r \texttt{cen}(\texttt{depth}{=}n, \texttt{dim}{=}d, s)\end{array}}(\textsc{over})$$

$$\frac{R <:^r S}{\texttt{atLeast}\, S <: \texttt{atLeast}\, R}(\texttt{L}) \qquad \frac{R <:^r S}{\texttt{atMost}\, R <: \texttt{atMost}\, S}(\texttt{M})$$

$$\frac{R <:^r S}{R <: \texttt{atMost}\, S}(\textsc{grow}) \qquad \frac{R <:^r S}{S <: \texttt{atLeast}\, R}(\textsc{shrink}) \qquad \frac{R \equiv S}{R <:^r S}(\textsc{eq})$$

$$\frac{\mathsf{dims}(R) \cap \mathsf{dims}(S) = \emptyset}{R * S <: R}(\textsc{Gen}*) \qquad \frac{}{\texttt{readOnce}\, R <: R}(\textsc{rep})$$

Figure 4. Approximation $<:^r$ on *regions* and $<:$ on *specs* (omitting reflexivity, transitivity, and congruences).

the spatial approximation on regions into approximation on specifications via the `atLeast` and `atMost` modifiers. Finally, (GEN$*$) shows that specifications can be generalised by removing parts of a product $*$ which have disjoint dimensions to the rest of the region product (where dims denotes the set of dimensions specified in a region).

Theorem 16 (Approximation soundness)**.** Code consistent with a specification S is consistent with a more general specification T, *i.e.*, for all models M of a piece of code:

$$S <: T \ \wedge (M \textbf{ agree } [\![S]\!]_n) \ \Rightarrow \ (M \textbf{ agree } [\![T]\!]_n)$$

where **agree** implements consistency in checking (§ 5.2). See the supplement for the proof.

5. Analysis, checking, and inference

We outline here the procedures for checking conformance of source code against specifications (§ 5.2) and for inferring specifications from code (§ 5.3). Both rely on a program analysis that converts array subscripts into sets of index schemes. We outline this analysis first (§ 5.1). Note that the analysis can be made arbitrarily more complex and wide-ranging independent of the checking and inference procedures. At the moment, the analysis is largely *syntactic*, with only a small amount of semantic interpretation of the code.

5.1 Analysis of array accesses

The analysis builds on standard program analyses: **(1)** basic blocks (CFG); **(2)** induction variables per basic block; **(3)** (interprocedural) data-flow analysis, providing a *flows to* graph (as shorthand, the function flowTo is used, implicitly parameterised by this graph, mapping an expression to the set of all expressions with forwards data-flow to this expression, based on the transitive closure of the flows graph); **(4)** type information per variable.

The core analysis, captured by the function *analyse* below, then proceeds as follows. For each assignment statement *assg* within a loop whose left-hand side is an array subscript on neighbourhood indices, a finite map is computed mapping array variables to *multisets* of index schemes:

$$\begin{array}{ll} \mathit{analyse}(a(\overline{e_1}) = e_2) := & \textit{where } \mathsf{neigh}(\overline{e_1}) \\ \quad \bigcup\{a' \mapsto \{\mathsf{schematic}(\bar{e})\} \mid a'(\bar{e}) \leftarrow \mathsf{flowsTo}(e_2)\} & \end{array}$$

The predicate $\mathsf{neigh}(\bar{e_1})$ classifies indices where each $e \in \bar{e_1}$ is either a neighbourhood index (based on the induction variables of the loop, implicit here) or a constant index.[2] Thus, we focus on assignments to an array subscript where the LHS indices $\overline{e_1}$ are neighbourhood indices. For all array subscripts that flow to the right-hand side of this statement, a finite map is constructed, mapping each array variable to a multiset of index schemes for its subscripts, computed with schematic (Def. 6). Note schematic is undefined ($\bot$) if $\bar{e}$ contains relative indices which are not neighbourhood indices. The *analysis* function is used by the *modelise* function which filters out those arrays which contain undefined index schemes and adds multiplicity information:

$$\mathit{modelise}(\mathit{assg}) = \mathit{let}\ \ M = \mathit{analyse}(\mathit{assg})(a)\ \ \mathit{in}$$
$$a \mapsto \begin{cases} \bot & \bot \in M \\ \mathsf{only}(|M|) & \forall u \in M.\, \#(M, u) = 1 \\ \mathsf{mult}(|M|) & \forall u \in M.\, \#(M, u) = n \end{cases}$$

where $\#(M, u)$ denotes the multiplicity of an element u of the multiset M and $|M|$ denotes the set of elements in the multiset M. Thus, the analysis maps source code to index schemes, which is then augmented with multiplicity information by $\mathit{modelise}$, *i.e.* values in $\mathsf{Mult}(\mathcal{P}((\mathbb{Z}_\infty)^n))$.

As a small semantic concession to an otherwise syntactic analysis: if the LHS $\overline{e_1}$ contains non-0 neighbour offsets then all RHS schemes are *relativised* by $\mathsf{schematic}(\overline{e_1})$. For example, if `a(i+1) = b(i)` then relativisation of the RHS by the LHS produces an analysis which is equivalent to the analysis of `a(i) = b(i-1)`.

Example 17. We demonstrate the analysis and modelise procedures over the following example kernel which computes the mean value of a five-point stencil:

```
1  x = a(i-1, j)+a(i+1, j); y = a(i, j-1)+a(i, j+1)
2  b(i, j) = (a(i, j) + x + y) / 5.0
```

The multiplicities are all one, and so *modelise* yields:

$$\mathsf{a} \mapsto \mathsf{only}(\{[-1\ 0], [1\ 0], [0\ -1], [0\ 1], [0\ 0]\})$$

5.2 Specification checking

Checking verifies the access pattern of a stencil computation in the source language against its associated specifications. The model of (§ 3) guides this process. Checking proceeds by generating a model from a specification and generating a model from the source code, comparing them for *consistency* base on index scheme *generalisation* (Def. 7).

[2] If all indices $e \in \bar{e}$ are constants, then $\mathsf{neigh}(\bar{e})$ does not hold. Instead $a(\bar{e})$ is considered a constant by our analysis, rather than an array subscript.

$$(\text{up})\frac{\forall u \in M, \exists v \in N.\, u \preceq v}{M\ \textbf{agree}'\ \mathsf{upper}(N)} \quad (\text{low})\frac{\forall v \in N, \exists u \in M.\, u \preceq v}{M\ \textbf{agree}'\ \mathsf{lower}(N)}$$

$$(\text{exact})\frac{M\ \textbf{agree}'\ \mathsf{lower}(N) \qquad M\ \textbf{agree}'\ \mathsf{upper}(N)}{M\ \textbf{agree}'\ \mathsf{exact}(N)}$$

$$(\text{both})\frac{M\ \textbf{agree}'\ \mathsf{lower}(M_1) \qquad M\ \textbf{agree}'\ \mathsf{upper}(M_2)}{M\ \textbf{agree}'\ \mathsf{both}(M_1, M_2)}$$

$$(1)\frac{M\ \textbf{agree}'\ N}{\mathsf{once}(M)\ \textbf{agree}\ \mathsf{once}(N)} \quad (*)\frac{M\ \textbf{agree}'\ N}{\mathsf{mult}(M)\ \textbf{agree}\ \mathsf{mult}(N)}$$

$$(1 < *)\frac{M\ \textbf{agree}'\ N}{\mathsf{once}(M)\ \textbf{agree}\ \mathsf{mult}(N)}$$

Figure 5. Consistency of models (code vs. specification)

Definition 18 (Checking)**.** Let a be an array variable and S a specification associated to a via `!= stencil` S `:: a`.

A stencil kernel $v(\overline{e_1}) = e_2$ conforms to S if the predicate $\textbf{check}_n(S, v(\overline{e_1}) = e_2, a)$ holds, defined:

$$\textbf{check}_n(S, assg, a) = \mathit{modelise}(assg)(a)\ \textbf{agree}_n\ [\![S]\!]_n$$

Thus, checking applies *modelise* (§ 5.1) to extract index schemes from source code, augmented with multiplicity for the whole set of schemes. This is compared with the model of the specification $[\![S]\!]_n$ by the consistency predicate **agree**.

Definition 19. Consistency relates models of source code to models of specifications (which are bounded) via predicates:

$$\textbf{agree}'_n \subseteq \mathcal{P}({\mathbb{Z}_\infty}^n) \times \mathsf{Approx}(\mathcal{P}({\mathbb{Z}_\infty}^n))$$
$$\textbf{agree}_n \subseteq \mathsf{Mult}(\mathcal{P}({\mathbb{Z}_\infty}^n)) \times \mathsf{Mult}(\mathsf{Approx}(\mathcal{P}({\mathbb{Z}_\infty}^n)))$$

where **agree**′ relates models without multiplicity. Fig. 5 defines the two predicates, where for brevity we make implicit the subscript n defining the dimensionality. We denote sets of index schemes $\mathcal{P}((\mathbb{Z}_\infty)^n)$ from source code model by M and from a specification model by M.

Rules (*up*) and (*low*) are the core; (*up*) considers the case where the specification is an `atMost` upper-bound approximation. An upper-bound specification means that for every scheme in the code model, modelling a source-level index, $\forall u \in M$ there exists a generalising scheme in the spec model $\exists v \in N$, *i.e.*, $u \preceq v$. That is, every source-level index scheme is generalised by the specification. The (*down*) rule provides the case of `atLeast` lower-bound specifications. Consistency is dual to the above: for every scheme in the specification model $\forall v \in N$ there exists a scheme in the code model $\exists u \in M$ such that $u \preceq v$. That is, every specification scheme is generalised by some source index scheme.

The (*up*) and (*down*) rules are combined for an exact specification (*exact*) such that a specification's model provides both a consistent lower and consistent upper bound on the source code. Thus, for every scheme in the code model there is a scheme in the spec model that generalises it, and for every scheme in the specification there is a scheme in the source model that specialises it. The (*both*) rule is similar, decomposing a specification with both a lower and upper bound into consistency of these bounds.

Rules (1) and (∗) extend the **agree**′ predicate to **agree** with multiplicity information. Both rules state that consistent models remain consistent if they have the same multiplicity. Relatedly, rule $(1 < *)$ ensures that absence of a `readOnce` modifier in the specification does not prohibit a set of unique stencil accesses in the code from being consistent.

Example 20. Consider the following five-point stencil kernel with a (quite general) specification:

```
1  != region :: fivepoint = centered(depth=1,dim=1) +
                          ↪  centered(depth=1,dim=2)
2  != stencil readOnce, fivepoint :: b
3  a(i,j) = b(i,j-1)+b(i,j)+b(i,j+1)+b(i-1,j)+b(i+1,j)
```

We refer to line three as $l3$ for brevity, which is checked via $\mathit{check}((\texttt{readOnce, fivepoint}), l3, \texttt{b})$ with steps:

1. Generate the model from the specification as in § 3.
$$N_s = \{\begin{bmatrix}-1\\ \infty\end{bmatrix}, \begin{bmatrix}0\\ \infty\end{bmatrix}, \begin{bmatrix}1\\ \infty\end{bmatrix}, \begin{bmatrix}\infty\\ -1\end{bmatrix}, \begin{bmatrix}\infty\\ 0\end{bmatrix}, \begin{bmatrix}\infty\\ 1\end{bmatrix}\}$$
$$[\![\texttt{readOnce, fivepoint}]\!]_2 = \mathsf{once}(\mathsf{exact}(N_s))$$
2. Generate code model $\mathit{modelise}(l3, \texttt{b}) = \mathsf{once}(M_c)$ where:
$$M_c = \{\begin{bmatrix}0\\ -1\end{bmatrix}, \begin{bmatrix}0\\ 0\end{bmatrix}, \begin{bmatrix}0\\ 1\end{bmatrix}, \begin{bmatrix}-1\\ 0\end{bmatrix}, \begin{bmatrix}1\\ 0\end{bmatrix}\}$$
3. By the definition of consistency (Fig. 5) $\mathsf{once}(\mathsf{exact}(N_s))$ is consistent with $\mathsf{once}(M_c)$ when:
$$\mathsf{up}(N_s)\ \textbf{agree}_2\ M_c \ \wedge\ \mathsf{low}(N_s)\ \textbf{agree}_2\ M_c$$
4. This holds by the following sets of scheme generalisations: for the upper bound, $\forall u \in M_c, \exists v \in N_s . u \preceq_2 v$:
$$\begin{array}{lll}[0\ -1] \preceq_2 [0\ \infty] & [0\ 0] \preceq_2 [0\ \infty] & [0\ 1] \preceq_2 [0\ \infty] \\ [-1\ 0] \preceq_2 [\infty\ 0] & [1\ 0] \preceq_2 [\infty\ 0] & \end{array} \quad (1)$$
and for the lower bound: $\forall v \in N_s, \exists u \in M_c . u \preceq_n v$:
$$\begin{array}{lll}[-1\ 0] \preceq_2 [-1\ \infty] & [0\ 0] \preceq_2 [0\ \infty] & [1\ 0] \preceq_2 [1\ \infty] \\ [0\ -1] \preceq_2 [\infty\ -1] & [0\ 0] \preceq_2 [\infty\ 0] & [0\ 1] \preceq_2 [\infty\ 1] \end{array} \quad (2)$$

Since the consistency predicate is satisfied, the array accesses flowing to line three in the stencil computation are valid with respect to the `fivepoint` specification.

Example 21. Consider an erroneous user modification to $l3$ with an additional term `b(i+2,j+2)`, call it $l3'$.

Checking proceeds as in the previous example, except $\mathit{modelise}(l3', \texttt{b})$ generates an additional scheme [2 2]. The generalisations for the lower bound (2) of step 4 above still hold since its consistency condition universally quantifies over the *specification model*. However, for the upper bound, the consistency condition universally quantifies over schemes u of the *code model* for which there must exist a scheme $v \in N_s$ such that $u \preceq_2 v$. Thus, we require a $v \in N_s$ such that $[2\ 2] \preceq_2 v$ but no such generalising scheme exists in the spec model. Hence checking fails.

5.3 Specification inference

To aid adoption of our approach to legacy code, we provide an inference procedure for generating specifications from code, which are inserted automatically as comments. We illustrate inference using the five-point stencil of Example 17.

5.3.1 Step 1: Data access analysis

The first step performs the analysis of § 5.1 to produce a model of the code in terms of index schemes. The *analyse* and *modelise* procedures before yielded the finite map:

$$\mathsf{a} \mapsto \mathsf{only}(\{[-1\ 0]\,,[1\ 0]\,,[0\ -1]\,,[0\ 1]\,,[0\ 0]\})$$

5.3.2 Step 2: Coalesce contiguous schemes into spans

Let U range over the finite maps computed by *modelise*. For each array variable $a \in \mathsf{dom}(U)$, the algorithm constructs a set of n-dimensional rectangles covering all contiguous groups of schemes in $U(a)$.

Definition 22 (Spans)**.** A *span* represents an n-dimensional rectangle as a pair of n-dimensional vectors (or represented as a $2\times n$ matrix) with values drawn from $\mathbb{Z}_\infty$. This gives the co-ordinates of lower-bound and upper-bound vertices. We write spans as $\mathbf{x} = [\mathbf{x}^L\ \mathbf{x}^U]$, where $\mathbf{x}^L$ and $\mathbf{x}^U$ give the lower and upper bounds. In the following, $\mathbf{x}, \mathbf{y}$ range over spans. For a span $\mathbf{x}$, we write $\mathbf{x}_{i:j}$ for the sub-span comprising the subvectors $[\mathbf{x}^L_{i:j}\ \mathbf{x}^U_{i:j}]$ from entry i to entry j (inclusive).

Each index scheme $u \in U(a)$ is converted to a *unit* span $u \mapsto [u\ u]$, defining the lower and upper bound of an n-dimensional cube of side length 1. For the five-point stencil example there are five unit spans, illustrated in Fig. 6(a).

The next step repeatedly coalesces any contiguous spans until a fixed point is reached, resulting in a set of (possibly overlapping) spans covering the original space of schemes. Two n-dimensional spans $\mathbf{x}$ and $\mathbf{y}$ are contiguous if for all but one $j \in \{1,\ldots,n\}$ then $\mathbf{x}_j = \mathbf{y}_j$ and for one j then $\mathbf{x}^U_1 + 1 = \mathbf{y}^L$. For example, spans $[[-1\ 0]\ [-1\ 0]]$ and $[[0\ 0]\ \ [1\ 0]]$ are contiguous in the first dimension and can be coalesced into the span $[[-1\ 0]\ \ [1\ 0]]$.

Definition 23. Span coalescing $\bullet$ is partial, defined for spans contiguous in the first dimension:

$$\mathbf{x} \bullet \mathbf{y} = (\mathbf{x}^L, \mathbf{y}^U) \quad \textit{if} \quad \mathbf{x}^U_1 + 1 = \mathbf{y}^L_1 \ \wedge\ \mathbf{x}_{2:n} = \mathbf{y}_{2:n}$$

To coalesce regions which are contiguous in other dimensions other than the first, we generate all rotations of the spans in n-dimensional space by permutation on spans, and fold together contiguous regions using $\bullet$ (above). This generates all possible ways of coalescing regions. For our example, the unit spans in Fig. 6(a) are coalesced into the contiguous spans in Fig. 6(b). This procedure is iterated until a fixed point is reached (in one step in this example). Then any spans that are completely contained by any other span are deleted, leaving a minimal set of spans (which may overlap, but none of which fully contains another), Fig. 6(c). The

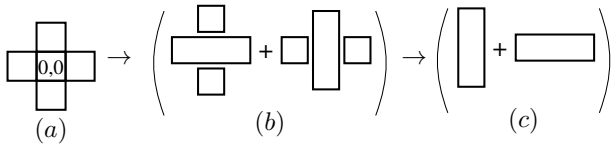

Figure 6. Illustration of grouping spans in inference

final spans are used to synthesise concrete stencil specification syntax (§ 5.3.3), where each span becomes a product $*$ of regions which are combined by the sum $+$.

More formally the algorithm proceeds as follows. For all $a \in \mathsf{dom}(U)$, let $M = U(a)$. Every scheme in M is mapped to a unit span by $\mathsf{unitSpans}(M) = \{[u\ u] \mid u \in M\}$, *e.g.*:

$$N(U(\mathsf{a})) = \begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}1&1\\0&0\end{bmatrix}\begin{bmatrix}-1&-1\\0&0\end{bmatrix}\begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}0&0\\-1&-1\end{bmatrix}$$

In the following we let $M_0 = N(U(\mathsf{a}))$ for our example.

Each set of unit spans is applied to the fixed point of the spans function (defined shortly), which coalesces spans into contiguous regions. That is, we compute $(\mu\,\mathsf{spans})$ on input $N(M)$, where spans is defined by the steps:

1) Compute all permutations on the column vectors in a span, *i.e.*, $[\mathbf{x}^L\mathbf{x}^U] \mapsto [\pi\mathbf{x}^L\,\pi\mathbf{x}^U]$ for a permutation π. For each permutation function π^n_i (the i-th permutation for vectors of size n) we pair the permutation function with the set of permuted spans so that the spans can be un-permuted later.

$$P(M) = \bigcup_{i\in n!} (\pi^n_i,\ \{[\pi^n_i\mathbf{x}^L\ \pi^n_i\mathbf{x}^U] \mid [\mathbf{x}^L\ \mathbf{x}^U] \leftarrow M\}$$

This corresponds to all rotations of the space, *e.g.*:

$$P(M_0) = \{(\pi^2_1, \{\begin{bmatrix}0&0\\0&0\end{bmatrix}, \begin{bmatrix}1&1\\0&0\end{bmatrix}, \begin{bmatrix}-1&-1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\1&1\end{bmatrix}, \begin{bmatrix}0&0\\-1&-1\end{bmatrix}\})$$
$$(\pi^2_2, \{\begin{bmatrix}0&0\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\1&1\end{bmatrix}, \begin{bmatrix}0&0\\-1&-1\end{bmatrix}, \begin{bmatrix}1&1\\0&0\end{bmatrix}, \begin{bmatrix}-1&-1\\0&0\end{bmatrix}\})\}$$

where π^2_1 is the identity permutation and π^2_2 is the permutation that flips the order of the two elements in the vectors.

2) Sort each permutation set into a list, by the ordering:

$$\mathbf{x} \le \mathbf{y} = \exists i\,.\,\mathbf{x}^L_i \le \mathbf{y}^L_i \ \wedge\ (i = n \vee \mathbf{x}_{i+1:n} = \mathbf{y}_{i+1:n})$$

giving the sorting function $\mathsf{sort}(M)$, *e.g.*, $\mathsf{sort}(P(M_0))$:

$$\{(\pi^2_1, [\begin{bmatrix}0&0\\-1&-1\end{bmatrix}, \begin{bmatrix}-1&-1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\0&0\end{bmatrix}, \begin{bmatrix}1&1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\1&1\end{bmatrix}])$$
$$(\pi^2_2, [\begin{bmatrix}0&0\\-1&-1\end{bmatrix}, \begin{bmatrix}-1&-1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\0&0\end{bmatrix}, \begin{bmatrix}1&1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\1&1\end{bmatrix}])\}$$

3) Reduce each list pairwise by $\bullet$ (Def. 23) to coalesce contiguous regions, given by $\mathsf{fold}\bullet$. For our example:

$$\{(\pi^2_1, [\begin{bmatrix}0&0\\-1&-1\end{bmatrix}, \begin{bmatrix}-1&1\\0&0\end{bmatrix}, \begin{bmatrix}0&0\\1&1\end{bmatrix}]), (\pi^2_2, \textit{same as for } \pi_1)\}$$

4) Un-permute and union together, *i.e.*,

$$U(M) = \bigcup\{[\pi\mathbf{x}^L, \pi\mathbf{x}^U] \mid [\mathbf{x}^L, \mathbf{x}^U] \leftarrow S, (\pi, S) \leftarrow M\}$$

For our example:

$$U(M) = \{\begin{bmatrix} 0 & 0 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} -1 & -1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\}$$

5) Filter by the region containment predicate $\sqsubseteq$; if any region is contained within another then remove the smaller:

$$\mathbf{x} \sqsubseteq \mathbf{y} = \mathbf{y}_1^L \leq \mathbf{x}_1^L \wedge \mathbf{x}_1^U \leq \mathbf{y}_1^U \wedge (\mathbf{x}_{2:n}^L, \mathbf{x}_{2:n}^U) \sqsubseteq (\mathbf{y}_{2:n}^L, \mathbf{y}_{2:n}^U)$$

For our example this then yields the final result of:

$$\mathsf{spans}(M) = \{\begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}\}$$

since $\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \sqsubseteq \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \sqsubseteq \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$. Applying spans again yields the same result for our example.

5.3.3 Step 4: Synthesise specifications from spans

Abstract syntax trees of the specification language are synthesised from the set $\mathsf{spans}(M)$ of coverings spans. This conversion is defined in terms of an algebra on *spec* syntax in disjunctive-normal form (which we denote **spec**) and which is wrapped in the Approx type (see Def. 14). The algebra mirrors the shape of the region operators + and $*$ with binary operators $\oplus$ and $\odot$ on $\mathsf{Approx}(\mathbf{spec})$. A function toSpec maps a dimension identifier and a pair of lower and upper bound values (one-dimensional) to a specification with a one-dimensional region.

$$\mathsf{toSpec} : \mathbb{N}_{>0} \to (\mathbb{Z}_\infty \times \mathbb{Z}_\infty) \to \mathsf{Approx}(\mathbf{spec})$$

A span is converted into an n-dimensional specification by applying toSpec to each pair of lower and upper bounds in a span, per dimension, and multiplying these by $\odot$. The resulting specifications for each span are summed by $\oplus$:

$$\mathsf{spec}(M) = \bigoplus_{\mathbf{x} \in \mathsf{spans}(M)} \bigodot_{i \in \{1,\ldots,n\}} \mathsf{toSpec}\ i\ (\mathbf{x}_i^L, \mathbf{x}_i^U)$$

We unpack the definitions of this algebra.

Definition 24 (Specifications in DNF)**.** Let $r_1, ..., r_n$ range over elements of the *region* syntax (see Fig. 1). The set **spec** is a subset of syntax trees *spec*, containing specifications in disjunctive normal form over regions where the disjunct is + and the conjunct is $*$; elements of **spec** are of the form: $(r_1^1 * \ldots * r_n^1) + \ldots + (r_1^m * \ldots * r_p^m)$. In the following, S, T range over elements of **spec** and s^*, t^* range over conjuncts *e.g.* $s^* = r_1^1 * \ldots * r_n^1$.

Definition 25. The toSpec constructor is defined in Fig. 7, where the lower and upper bounds determine region constants; $\mathsf{toSpec}\, d\, (l, u)$ is defined if l less than or equal to u.

Positive lower and upper bound yield `forward` regions, and negative lower and upper bounds yield `backward` regions. The `nonpointed` attribute (abbreviated `np`) is present when the lower bound is at 1 or -1. In the sixth-case, where

$\mathsf{toSpec}\ d\ (l, u) =$

$$\begin{cases} \mathsf{exact}\ (\mathtt{pointed}\,(\mathtt{dim}{=}d)) & l = 0 \wedge u = 0 \\ \mathsf{exact}\ (\mathtt{fwd}\,(\mathtt{depth}{=}u, \mathtt{dim}{=}d)) & l = 0 \wedge u > 0 \\ \mathsf{exact}\ (\mathtt{fwd}(\mathtt{depth}{=}u, \mathtt{dim}{=}d, \mathtt{np})) & l = 1 \wedge u > 0 \\ \mathsf{exact}\ (\mathtt{bwd}\,(\mathtt{depth}{=}|l|, \mathtt{dim}{=}d)) & l < 0 \wedge u = 0 \\ \mathsf{exact}\ (\mathtt{bwd}(\mathtt{depth}{=}|l|, \mathtt{dim}{=}d, \mathtt{np})) & l < 0 \wedge u = -1 \\ \mathsf{exact}\ (\mathtt{cen}\,(\mathtt{depth}{=}u, \mathtt{dim}{=}d)) & l < 0 \wedge u > 0 \wedge |l| = u \\ \mathsf{exact}\ (\mathtt{bwd}\,(\mathtt{depth}{=}|l|, \mathtt{dim}{=}d) & l < 0 \wedge u > 0 \wedge |l| \neq u \\ \quad + \mathtt{fwd}\,(\mathtt{depth}{=}u, \mathtt{dim}{=}d)) & \\ \mathsf{upper}\ (\mathtt{fwd}\,(\mathtt{depth}{=}u, \mathtt{dim}{=}d)) & l > 1 \\ \mathsf{upper}\ (\mathtt{bwd}\,(\mathtt{depth}{=}|l|, \mathtt{dim}{=}d)) & l < (-1) \\ \mathsf{exact}\ (\epsilon) & l = \infty \vee u = \infty \end{cases}$$

Figure 7. toSpec constructor of $\mathsf{Approx}(\mathbf{spec})$ values.

the lower bound is negative and the upper bound is positive and they have the same magnitude, then a `centered` region is constructed with the depth as the common magnitude. If the magnitudes are not the same (seventh case) then separated `forward` and `backward` regions are summed, with their corresponding depths. If the lower bound is greater than 1 (which implies the upper bound is greater than 1 too) then this represents a region that is not at the origin (or next to) the origin and thus cannot be represented in the specification language. Thus, an upper bound is produced showing the maximum extent of the region. The penultimate case is the dual, when the lower bound is less than -1, giving an upper bound on a `backward` specification.

Definition 26. The $\odot$ operation on $\mathsf{Approx}(\mathbf{spec})$ is defined in terms of the intermediate operator $\hat{\odot}$ on **spec**, defined:

$$S \,\hat{\odot}\, \epsilon = S \quad \epsilon \,\hat{\odot}\, S = S \qquad S \,\hat{\odot}\, T = \mathop{+}_{(s^*, t^*) \in (S \times T)} (s^* * t^*)$$

That is, the $\hat{\odot}$ product of two specifications S and T in DNF form is the + sum of all pairwise $*$ products for every pair of $spec^*$ drawn from S and T. The result is indeed in DNF-form, where $\hat{\odot}$ is essentially applying the (DIST) rule of the equational theory. The operation is commutative and associative, and sound with respect to our model (see Lemma 28 below). The $\hat{\odot}$ operation is then lifted to all combinations of Approx for $\odot$ in Fig. 8 where in the last case *inj* corresponds to the unary injections of Approx. For brevity we omit the cases where the arguments above are flipped as $\odot$ is commutative.

Definition 27. The $\oplus$ operation on $\mathsf{Approx}(\mathbf{spec})$ is defined similarly to $\odot$, with the intermediate $\hat{\oplus}$ on **spec**:

$$S \,\hat{\oplus}\, \epsilon = S \quad \epsilon \,\hat{\oplus}\, S = S \quad S \,\hat{\oplus}\, T = S + T$$

Thus, if neither specification is empty, their $\hat{\oplus}$ is just the syntactic sum +. Then $\oplus$ is the lifting of $\hat{\oplus}$ to Approx with the shape as $\odot$ in Figure 8.

$$\begin{aligned}
\mathsf{lower}(S) \odot \mathsf{exact}(T) &= \mathsf{both}\,(S \hat{\odot} T,\ T) \\
\mathsf{upper}(S) \odot \mathsf{exact}(T) &= \mathsf{both}\,(T,\ S \hat{\odot} T) \\
\mathsf{lower}(S) \odot \mathsf{both}(T_l, T_u) &= \mathsf{both}\,(S \hat{\odot} T_l,\ T_u) \\
\mathsf{upper}(S) \odot \mathsf{both}(T_l, T_u) &= \mathsf{both}\,(T_l,\ S \hat{\odot} T_u) \\
\mathsf{both}(S_l, S_u) \odot \mathsf{both}(T_l, T_u) &= \mathsf{both}(S_l \hat{\odot} T_l,\ S_u \hat{\odot} T_u) \\
\mathit{inj}(S) \odot \mathit{inj}(T) &= \mathit{inj}(S \hat{\odot} T)
\end{aligned}$$

Figure 8. Lifting $\hat{\odot}$ to approximations.

Lemma 28. $\forall S, T \in$ ***spec*** *where* $S \neq \epsilon \wedge T \neq \epsilon$*:*

$$[\![S \mathbin{\hat{\odot}} T]\!]_n = [\![S]\!]_n \otimes [\![T]\!]_n \qquad [\![S \mathbin{\hat{\oplus}} T]\!]_n = [\![S]\!]_n \cup [\![T]\!]_n$$

Thus, our model validates specification synthesis.

The final step applies a simplification pass, orienting the equational theory $\equiv$ (§ 4) into rewrite rules based on some simple heuristics. For our example, we infer and then synthesise the final 5-point specification:

```
!= stencil centered(depth=1, dim=1)*pointed(dim=2) +
     ↪  centered(depth=1, dim=2)*pointed(dim=1):: a
```

6. Evaluation

To study the effectiveness of our approach, we built a corpus of around 1 million lines of Fortran code from a range of scientific computing packages: The Unified Model (UM) [26], E3MG [5], BLAS [8], Hybrid4 [15], GEOS-Chem [7], Navier (based on [16]), Computational Physics 2 [18], ARPACK-NG [1], and SPECFEM3D [3].

We first examined how frequently stencil computations occur. We parsed 959,427 lines of Fortran code and found that 10% (97,439) of statements have a left-hand side as an array subscript on neighbourhood indices. This supports the idea that stencil-like computations are common in scientific code. We then used the inference procedure of the previous section to generate specifications for stencils in the corpus to assess the design of the language.

We would not expect to infer a stencil for each array statement we found because our analysis restricts the array-subscript-statements that we classify as a stencil. In fact, we were able to infer a stencil from 30% of these statements. A single statement can involve multiple arrays and we ended up with 60,525 specifications. This shows that we can express a large number of stencil shapes within our high-level abstraction and validates our initial hypothesis that many stencil computations have a regular shape.

The majority of specifications generated were relatively simple but we found significant numbers of more complex shapes. We grouped common patterns into categories:

All pointed 55,504 of the stencils we found involved only pointed regions. 39,681 of these were pointed in all dimensions. Common examples of this were pointwise transformations on data (such as scaling).

Single-action specifications comprise one forward, backward, or centered region constant combined via + or ∗ with any number of pointed regions. We identified 4,837 single-action specifications, of which 1,532 were single-action with a `nonpointed` modifier.

```
!=stencil readOnce,(forward(depth=1,dim=3,nonpointed)) ⌋
     *(backward(depth=1,dim=1))*(backward(depth=1,dim=2)) ⌋
↪    +(forward(depth=1,dim=3))*(backward(depth=1,dim=1)) ⌋
↪    *(pointed(dim=2))+(forward(depth=1,dim=3)) ⌋
↪    *(backward(depth=1,dim=1,nonpointed)) ⌋
↪    *(backward(depth=1,dim=2,nonpointed)) ⌋
↪    +(forward(depth=1,dim=3))*(backward(depth=1,dim=2)) ⌋
↪    *(pointed(dim=1))+(backward(depth=1,dim=1)) ⌋
↪    *(backward(depth=1,dim=2,nonpointed))*(pointed(dim=3)) ⌋
↪    +(backward(depth=1,dim=1,nonpointed)) ⌋
↪    *(backward(depth=1,dim=2))*(pointed(dim=3))::x
```

Figure 9. Complex specification inferred from **UM**

Multi-action specifications comprise at least two forward, backward, or centered regions, combined with any number of pointed regions. We identified 265 multi-action specifications out of which 207 had regions combined only with ∗ and 58 combined with a mix of ∗ and +.

Bounded specifications occured with 157 `atMost` bounds and 36 `atLeast`, the latter of which were always also paired with an upper bound.

The single- and multi-action classes represent more complex stencils with a real possibility for programmer error. As an extreme example, the Unified Model has an implementation of the Smagorinsky subgrid-scale model for calculating turbulence on which our inference yields 39 specs from 340 lines of code. This is a large reduction given the complexity of the algorithm. We show one example in Fig. 9 which specifies the access pattern to a 3-dimensional array (which we have renamed to `x`) arising from a kernel of 93 lines of code involving 142 array subscripts. The specification was the most complex seen in our corpus, yet it still represents a significant abstraction of the spatial behaviour given size and complexity of the kernel it describes.

We measured the frequency at which individual specifications involved multiple occurences of + and ∗:

	0	1	2	3	4	5	6	> 7
+	60,265	220	15	12	5	4	4	0
∗	31,393	15,151	13,457	424	41	42	5	12

e.g., roughly half the specifications did not involve ∗, a quarter use one ∗ and just under a quarter use two ∗ operators.

Limitations There were various reasons why we did not infer specifications on every looped array computation:
1) **Non-subset induction variables** occur when the induction variables on the RHS are not a subset of those in the LHS. These cases are not stencils by our definition. The degenerate case of this is to have only constant indices on the LHS. We see lots of examples of this in loops as accumulators *e.g.* computing the sum over an array;
2) **Derived induction variables** where an index (`x`) is derived from an induction variable (`i`) as in `x = len - i`;
3) **Inconsistent induction dimensions** occur when an induction variable is used to specify more than one array dimension on the RHS or multiple induction variables are used

for the same dimension on the RHS. These are common in matrix operations such as LU-decomposition with assignments such as `a(l) = a(l) - a(m) * b(l, m)`.

6.1 Detecting errors in the 2-D Jacobi iteration

One common example of a stencil computation is the two-dimensional Jacobi iteration that repeatedly goes through each cell in a matrix and computes the average value of the four adjacent cells. The kernel is given by:

```
1     a(i,j) = (a(i-1,j)+a(i+1,j)+a(i,j+1)+a(i,j-1))/4
```

We infer a precise specification of its shape as:

```
 != stencil
↪ pointed(dim=1)*centered(depth=1,dim=2,nonpointed)+
↪ pointed(dim=2)*centered(depth=1,dim=1,nonpointed)::a
```

To test our implementation, we examined whether programmer errors would be detected by replacing the array index offsets with -1, 0, or 1 and running our verification algorithm. Our checking procedure correctly reported a verification failure in each of 6,537 permutations corresponding to an error. The iteration computes the average of four adjacent cells so 24 (4 factorial) of the possible array index perturbations are correct, all of which are accepted by our checker.

7. Related work and conclusions

Various deductive verification tools can express array indexing in their specifications, *e.g.*, ACSL of Baudin et al. [6] for C (*e.g.* Burghardt et al. [11, Example 3.4.1]). A specification can be given for a stencil computation but must use fine-grained indexing as in code and therefore is similarly prone to indexing errors. Our approach is much more abstract– it does not aim to reify indexing in the specification, but provides simple spatial descriptions which capture a large number of common patterns.

Kamil et al. [17] propose *verified lifting* to extract a functionally-complete mathematical representation of low-level, and potentially optimised, stencils in Fortran code. This extracted predicate representation of a stencil is used to generate code for the HALIDE high-performance compiler [22]. Thus they must capture the full meaning of a stencil computation which requires significant program analysis. For example, they report that some degenerate stencil kernels take up to 17 hours to analyse and others require programmer intervention for correct invariants to be inferred.

Our approach differs significantly. Rather than full representation of stencils, we focus on specifying just the spatial behaviour in a lightweight way. Thus, it suffices for us to perform a comparatively simple data-flow analysis which is efficient, scales linearly with code size, and does not require any user intervention. Whilst we do not perform deep semantic analysis of stencils, the analysis part of our approach can be made arbitrarily more sophisticated independent of the rest of the work. Furthermore, Kamil *et al.* do not provide a user-visible syntactic representation of their specifications, and nor do they provide verification from specifications *e.g.*, to future-proof the code against later changes. Even if they were to provide a syntactic representation, for complex stencils such as Navier-Stokes from § 1, it would be as verbose as the code itself, making it difficult for programmer to understand the overall shape of the indexing behaviour.

Our work has similarities with efforts to verify kernels written for General-Purpose GPU programming, such as in Blom et al. [9]. However, their focus is mainly on the synchronisation of kernels and the avoidance of data races., while we are interested in correctness Solar-Lezama et al. [23] give specifications of stencils using unoptimised "reference" stencils, coupled with partial implementations which are completed by a code generation tool. The primary purpose of this tool is optimisation rather than correctness, and the language of specification is more elaborate than ours.

Tang et al. [24] define a specification language for writing stencils embedded in C++ (with Cilk [10] extensions) that are then compiled into parallel programs based on trapezoidal decompositions with hyperspace cuts. Pochoir specifications are used for describing the kernel, boundary conditions, and shape of the stencil. Pochoir is aimed at programmers reluctant to implement the high-performance cache-oblivious "hypertrapezoidal" algorithms. Like much of the related work, the goal is optimisation rather than verifying program correctness.

By contrast, the work of Abe et al. [4] studies correctness bringing a form of model-checking to verify certain stencil computations in the context of parallelism in partitioned global address space languages. Abe *et al.* provide a new language for writing stencil computations. Much of the specification effort goes towards describing the distribution of the computation over multiple processors. The code for the stencil kernel is generated from a relatively high-level specification. In contrast, we integrate directly into existing, legacy codebases and established languages, bringing the benefits of verification more easily to scientific computing.

Concluding remarks There is an increasing awareness of the need for verification techniques in science [19–21]. Our specification language is an approach in this direction, providing a system of lightweight, high-level specification for numerical code. As opposed to existing approaches our language takes inspiration from the numerical literature and provides a concise, abstract description of the stencil.

We evaluated our approach by extending CamFort, an open-source analysis tool for Fortran programs [2], running the inference process over a corpus of a million lines of Fortran code. We showed that our language is capable of capturing many real-world uses of stencils and showed in a case-study that it will detect (simulated) programmer errors.

Our restrictive definition of stencils means we do not generate specifications for a large number of iterated array accesses in the corpus, *e.g.* that incorporate reductions. For future work, we will look at expanding the flexibility of our analysis to include these examples.

References

[1] ARPACK-NG. https://github.com/opencollab/arpack-ng. Accessed: 15 November 2016.

[2] CamFort - refactoring, analysis, and verification tool for scientific Fortran programs. https://github.com/camfort/camfort. Accessed: 15 November 2016.

[3] SPECFEM3D. https://github.com/geodynamics/specfem3d. Accessed: 15 November 2016.

[4] T. Abe, T. Maeda, and M. Sato. Model checking stencil computations written in a partitioned global address space language. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 365–374, May 2013. doi: 10.1109/IPDPSW.2013.90.

[5] Terry Barker, Haoran Pan, Jonathan Kohler, Rachel Warren, and Sarah Winne. Decarbonizing the Global Economy with Induced Technological Change: Scenarios to 2100 using E3MG. *The Energy Journal*, 0(Special I):241–258, 2006. URL https://ideas.repec.org/a/aen/journl/2006se-a12.html.

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi c specification language, 2008.

[7] Isabelle Bey, Daniel J Jacob, Robert M Yantosca, Jennifer A Logan, Brendan D Field, Arlene M Fiore, Qinbin Li, Honguy Y Liu, Loretta J Mickley, and Martin G Schultz. Global modeling of tropospheric chemistry with assimilated meteorology: Model description and evaluation. *Journal of Geophysical Research: Atmospheres*, 106(D19):23073–23095, 2001.

[8] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[9] Stefan Blom, Marieke Huisman, and Matej Miheli. Specification and verification of GPGPU programs. *Science of Computer Programming*, 95, Part 3:376 – 388, 2014. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2014.03.013. URL http://www.sciencedirect.com/science/article/pii/S0167642314001531. Special Section: {ACM} SAC-SVT 2013 + Bytecode 2013.

[10] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[11] Jochen Burghardt, J Gerlach, L Gu, Kerstin Hartig, Hans Pohl, J Soto, and K Völlinger. Acsl by example, towards a verified c standard library. *DEVICESOFT project publication. Fraunhofer FIRST Institute (December 2011)*, 2010.

[12] Larry S Davis. A survey of edge detection techniques. *Computer graphics and image processing*, 4(3):248–270, 1975.

[13] C. Dawson, Q. Du, and T. Dupont. A finite difference domain decomposition algorithm for numerical solution of the heat equation. *Mathematics of Computation*, 57(195), 1991.

[14] PE Farrell, MD Piggott, GJ Gorman, DA Ham, and CR Wilson. Automated continuous verification and validation for numerical simulation. *Geoscientific Model Development Discussions*, 3:1587–1623, 2010.

[15] Andrew D. Friend and Andrew White. Evaluation and analysis of a dynamic terrestrial ecosystem model under preindustrial conditions at the global scale. *Global Biogeochemical Cycles*, 14(4):1173–1190, 2000. ISSN 1944-9224. doi: 10.1029/1999GB900085. URL http://dx.doi.org/10.1029/1999GB900085.

[16] M. Griebel, T. Dornsheifer, and T. Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*, volume 3. Society for Industrial Mathematics, 1997.

[17] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 711–726. ACM, 2016.

[18] H.N. Nicholas J. Giordano. *Computational Physics: 2nd edition*. Dorling Kindersley, 2006. ISBN 9788131766279. URL https://books.google.co.uk/books?id=RCCVN2A_1tQC.

[19] W.L. Oberkampf and C.J. Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[20] Dominic Orchard and Andrew Rice. A computational science agenda for programming language research. *Procedia Computer Science*, 29:713–727, 2014.

[21] D.E. Post and L.G. Votta. Computational science demands a new paradigm. *Physics today*, 58(1):35–41, 2005.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[23] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. *SIGPLAN Not.*, 42(6):167–178, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250754. URL http://doi.acm.org/10.1145/1273442.1250754.

[24] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the twenty-third annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 117–128. ACM, 2011.

[25] Philip Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[26] Damian R Wilson and Susan P Ballard. A microphysically based precipitation scheme for the uk meteorological office unified model. *Quarterly Journal of the Royal Meteorological Society*, 125(557):1607–1636, 1999.