Verification of Stencil Computations with Spatial Specifications

Abstract

Verifying the correctness of numerical computations relies on first giving a specification of their behaviour. In many cases, a numerical computation is derived from an underlying mathematical model, but this is often infeasible as a specification since it is highly decoupled from the implementation. In this paper, we consider a lightweight specification technique between low-level and high-level specifications for numerical code. We target *stencil computations*—a common idiom in numerical code, but one which is error-prone due to fine-grained indexing errors. We show that in practice, stencil computations tend to have a regular shape which is amenable to static analysis and simple specification. We describe an abstract spatial specification language for stencil computations that can be embedded as annotations in source code, along with inference, checking, and specification synthesis. We evaluate our language against a corpus of numerical Fortran code, and show that the vast majority of stencil computations indeed have a simple, regular, static shape. Around 10% of the stencils found evoke a programming pattern for which indexing errors are plausible. We simulate relevant programming errors in two stencil examples, and show that we are able to detect the vast majority of potential errors. This paper details our stencil specification language, a denotational model, inference, checking, and synthesis procedures, our implementation, and our evaluation studies.

Keywords program verification, specification, stencils

1. Introduction

Stencils are a ubiquitous programming pattern, common in scientific and numerical computing applications. Informally, a stencil computation outputs an array, where the value at each index i of this array 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 iter = 0, itermax
2     do i = 1, (n-1)
3        b(i) = a(i-1) - 2*a(i) + a(i+1)
4     end do
5     a = b
6  end do
```

Line 3 is the core of the stencil computation, calculating values at `b(i)` from a neighbourhood of elements about `a(i)`. Line 5 swaps `a` and `b` between iterations, where `b` becomes the input for the next iteration. The shape of the data access pattern on line 3 determines other aspects of the program and its efficient implementation: how much "boundary" is needed for the array, the most cache-efficient layout in memory, and the partitioning shape for parallel implementations.

In this example, the access pattern is simple, static, and easily understood. More complex stencil computations are much more prone to error from simple lexical mistakes. For example, Figure 1 shows three lines from a Navier-Stokes fluid simulator in which two arrays are read with different data access patterns, across two dimensions. The interaction is much harder to understand, with the potential for the developer to introduce an error via simple textual mistakes, for example writing `(i-1,j)` instead of `(i+1,j)`.

In practice, most stencil computations have a regular shape that can be described simply and abstractly with a small set of coarse-grained spatial descriptions. We introduce a simple specification language of such descriptions, abstracting over the fine-grained detail of stencil access patterns. For the Laplace example, our inference procedure provides the following specification to line 3:

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

This describes that `a` is accessed with a symmetrical pattern ("centered") to a depth of one in each direction in its first (and only) dimension. The inferred specification for the Navier-Stokes example is shown in Fig. 1(b). 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 by forward to depth of 1 in the first dimension and backward to a depth of 1 in the second dimension. Fig. 1(c) illustrates the specification pictorially.

In this paper, we make the following contributions:

- we introduce a specification language for stencils that captures many common forms of data access pattern (Section 2);
- we detail inference, checking, and specification synthesis algorithms for our language (Section 4), derived from a denotational model for the language (Section 3);
- we provide an implementation of our approach as an extension to CamFort, an open-source program analysis tool for Fortran;
- we report on a quantitative study of stencil computations on a corpus of numerical Fortran programs (ranging from small to large), totalling one million lines of code. Our tool identifies and synthesises specifications for 35,000 stencil computations in the corpus. Approximately 10% of the stencils we found are non-trivial, corresponding to code which is a possible source of errors;
- we give a detailed verification case study of two particular stencil computation, simulating programming errors. Our approach is able to detect the vast majority of possible indexing errors.

```
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
```

(a) Excerpt of a Navier-Stokes fluid simulator (based on Griebel et al. (1997)) showing highly-detailed stencil computation.

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

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

(b) Inferred and synthesised specification for (a) by CamFort.

(c) Pictorial representation of the two stencil specifications. The horizontal dimension is `dim=1` and the vertical is `dim=2`.

Figure 1. Fragment of a Navier-Stokes fluid simulator and its specification in our language.

2. Stencil specification language

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 Section 5 with the inference of such regular stencil patterns on our corpus of numerical programs.

Section 2.1 outlines the syntax. Section 2.2 provides an equational theory for specifications and a notion of sub-specification. Section 3 gives a semantic model via a set-based interpretation, for which the equality and sub-specification relation are proven sound.

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

Definition 1 (Base induction variable)**.** A integer variable v is a *base induction variable* if it is the control variable of a "for" loop,[1] incremented by 1 per iteration. The variable is marked as an induction variable only within the body of the loop. Since we ignore non-base (derived) induction variables, we simply say *induction variable*. Induction variables are ranged over by i, j, k throughout.

Definition 2 (Array subscript and index)**.** An *array subscript* is an expression that indexes an array, which we denote by $v(\bar{e})$ for the source language where $\bar{e}$ is shorthand for a syntactic list of indexing expressions. A *relative index* is a list $\bar{e}$ where each $e \in \bar{e}$ is defined in terms of base induction variables.

Definition 3 (Neighbourhood index)**.** For an array subscript $v(\bar{e})$, we say that $e \in \bar{e}$ is a *neighbourhood index* if it is of the form $e \equiv i$, $e \equiv i + a$ or $e \equiv i - a$ where a is an integer constant. That is, a neighbourhood index is a relative index defined as constant translations of base induction variables. (The relation $\equiv$ here identifies terms up-to commutativity of $+$ and the inverse relation of $+$ and $-$ *e.g.*, $(-b)+i \equiv i-b$). We classify neighbourhood indices by the predicate neigh_I where I is a set of induction variables. We omit I when clear from the context.

[1] Or equivalent, *e.g.*, `do` in Fortran, for our implementation.

2.1 Specification syntax

Figure 2 gives the syntax of stencil specifications, which we introduce in stages below. The top-level is given by the *specification* production which splits into either a *regionDec* (region declaration) or a *specDec* (specification declaration). Specification declarations associate (via the operator `::`) a specification to one or more program variables $\bar{v}$, describing how the array variables $\bar{v}$ are accessed.

Regions are the central building blocks of spatial specifications. Regions can either be declared alone with a *regionDec*, assigning a region specification *region* to a region variable *rvar* or given directly within a *spatial* specification.

Region constants Regions have as terminals the *region constants*, denoted by `reflexive`, `forward`, `backward`, or `centered`. Each region constant except `reflexive` is given a depth parameter n (natural number greater than 0). Every region constant receives a dimension identifier d (also a natural number greater than 0). Region constants specify that an array is read from by a collection of indices which in the d^{th} dimension are neighbourhood indices ranging from $i + 0$ up to $i \pm n$ inclusively, *e.g.*, the following is a valid stencil, reading from `a` and writing to `b`:

```
1  != stencil forward(depth=2, dim=1) :: a
2  b(i, 0) = a(i, 0) + a(i+1, 0) + a(i+2, 0)
```

where `i` is an induction variable. The specification is associated to the array variable `a`. Note that the second dimension is constant.

The `forward`, `backward`, and `centered` regions can all receive an additional optional attribute `irreflexive` which marks that the region does not include the origin (offset 0). For example, the following is a `backward` stencil, which is similar to `forward` but has negative neighbourhood indices, and which is irreflexive:

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

The `irreflexive` attribute cannot be used on `reflexive` regions.

A `centered` stencil is a combination of `forward` and `backward` stencils to the same depth, *e.g.*,

```
1  != stencil centered(depth=1, dim=1) :: a
2  b(i) = ( a(i) + a(i-1) + a(i+1) ) / 3.0
```

Sum and product of regions Region terms can be combined using the sum operator `+` or the product operator `*`.

The product of two regions $r{*}r'$ specifies that an array is read with neighbourhood indices drawn from the bounding box created by the regions two regions r and r'. For example, the following code (defining a *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 applicaitons (for example for edge detection). 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`.

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

Figure 2. Specification syntax (EBNF grammar)

The region constructor `*` can also be interpreted as a kind of conjunction on specifications since they constrain indices to be within both regions simultaneously.

The sum of two regions r+r' specifies that an array is read using the neighbourhood indices described by either r or r'; it acts as a disjunction on specifications. For example, the following gives the specification of a five-point stencil which is the sum of two compound `reflexive` and `centered` regions in each dimension:

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

As another example, the following implements and specifies the *Roberts cross* edge-detection convolution (Davis 1975):

```
do j=0, jmax-1
  do i=0, imax-1
    x = a(i,j)-a(i+1,j+1); y = a(i+1,j)-a(i,j+1)
    != stencil forward(depth=1,dim=1) *
            ↪  forward(depth=1,dim=2) :: a
    b(i,j) = sqrt(x*x+y*y)
  end do
end do
```

Region declarations and variables Region specifications can be assigned to region variables via a region declaration, which can be used later to form a spatial specification. For example, the specification from the previous example can be restated as:

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

This is especially useful for common stencil patterns, such as Roberts cross, as the region can be defined once and reused.

Modifiers Region specifications can be modified by *approximation* and *multiplicity* information (in the *spec* rule of Fig. 2).

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

```
!= stencil readOnce, backward(depth=2, dim=1) :: a
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 *linearity* in linear type systems (Wadler 1990), 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 specification syntax. For example:

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

Note on the design The names "forward", "backward" and "centered" are inspired by standard terminology in numerical analysis for the shape of discretisation schemes. For example, the standard *explicit method* for approximating PDEs is known as the *Forward Time, Centered Space* (FTCS) scheme (Dawson et al. 1991). For the one-dimensional heat equation, an FTCS discretisation provides code with the following stencil (Recktenwald 2004):

```
do i=2, n-1
  u(i) = r*v(i-1) + r2*v(i) + r*v(i+1)
end do
```

where `r` and `r2` are constants. A valid specification for this is:

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

Such a specification can be inferred, or can be inserted by a user and checked against the code.

2.2 Equational theory and sub-specifications

Figure 3 lists the equational theory for specifications given by the binary relation $\equiv$ on *region* terms. The equations use abbreviations `refl`, `fwd`, `bwd`, `cen`, and `irrefl` for `reflexive`, `forward`, `backward`, `centered`, and `irreflexive` respectively. Furthermore, the abbreviations include syntactic sugar for quantifying over the absence or presence of the `irreflexive` attribute on region constants as a boolean. For example, `fwd` syntactic sugar is defined:

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

$$\textit{or } \texttt{fwd}\,(\texttt{depth=}n, \texttt{dim=}d) := \texttt{forward}\,(\texttt{depth=}n, \texttt{dim=}d)$$

and similarly for `bwd` and `cen`.

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

$$(\textit{region} \equiv \textit{region'}) \Rightarrow \texttt{stencil}\ [\textit{approx},]\ [\textit{mult},]\ \textit{region} \equiv \texttt{stencil}\ [\textit{approx},]\ [\textit{mult},]\ \textit{region}'$$

The equations reveal a part of the semantics for our specification language, which we briefly unpack.

Overlapping regions with + The first nine equations (with labels of the form . . . + . . .) explain the notion of *region overlapping* with sum `+`. In the first six, r and s range over booleans denoting the absence (true) or presence (false) of `irreflexive`. These are combined via disjunction. For example, the rule (B + F) explains that the sum of `backward` and `forward` regions in the same dimension and the same depth is equivalent to a `centered` region of that depth. If the `forward` part is irreflexive ($r = \textit{false}$) but `backward`

$$
\begin{array}{rl}
\text{(F + F)} & \texttt{fwd}(\texttt{depth=}n \max m, \texttt{dim=}d, r \vee s) \\
& \equiv \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{fwd}(\texttt{depth=}m, \texttt{dim=}d, s) \\
\text{(B + B)} & \texttt{bwd}(\texttt{depth=}n \max m, \texttt{dim=}d, r \vee s) \\
& \equiv \texttt{bwd}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{bwd}(\texttt{depth=}m, \texttt{dim=}d, s) \\
\text{(C + C)} & \texttt{cen}(\texttt{depth=}n \max m, \texttt{dim=}d, r \vee s) \\
& \equiv \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{cen}(\texttt{depth=}m, \texttt{dim=}d, s) \\
\text{(C + F)} & \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r \vee s) \\
m \leq n & \equiv \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{fwd}(\texttt{depth=}m, \texttt{dim=}d, s) \\
\text{(C + B)} & \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r \vee s) \\
m \leq n & \equiv \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{bwd}(\texttt{depth=}m, \texttt{dim=}d, s) \\
\text{(B + F)} & \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r \vee s) \\
& \equiv \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, r) + \texttt{bwd}(\texttt{depth=}n, \texttt{dim=}d, s) \\
\text{(R + F)} & \texttt{fwd}\,(\texttt{depth=}n, \texttt{dim=}d) \\
& \equiv \texttt{refl}(\texttt{dim=}d) + \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, r) \\
\text{(R + B)} & \texttt{bwd}\,(\texttt{depth=}n, \texttt{dim=}d) \\
& \equiv \texttt{refl}(\texttt{dim=}d) + \texttt{bwd}(\texttt{depth=}n, \texttt{dim=}d, r) \\
\text{(R + C)} & \texttt{cen}\,(\texttt{depth=}n, \texttt{dim=}d) \\
& \equiv \texttt{refl}(\texttt{dim=}d) + \texttt{cen}(\texttt{depth=}n, \texttt{dim=}d, r) \\
\text{(IRREFL)} & (R * S^{\texttt{irrefl}}) + (R^{\texttt{irrefl}} * S) \;\equiv\; R * S \\
\text{(+IDEM)} & S + S \;\equiv\; S \\
\text{(+COMM)} & S + T \;\equiv\; T + S \\
\text{(+ASSOC)} & R + (S + T) \;\equiv\; (R + S) + T \\
\text{(*COMM)} & S * T \;\equiv\; T * S \\
\text{(*ASSOC)} & R * (S * T) \;\equiv\; (R * S) * T \\
\text{(DIST)} & R * (S + T) \;\equiv\; (R * S) + (R * T)
\end{array}
$$

Figure 3. Equations on specifications (*regions*)

is not ($s = \mathit{true}$) then their sum is equal to a `centered` region with no `irrefl` modifier by the disjunction ($r \vee s = \mathit{true}$), *i.e.*:

$$
\begin{array}{rl}
& \texttt{cen}\,(\texttt{depth=}n, \texttt{dim=}d) \\
\equiv & \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, \texttt{irrefl}) + \texttt{bwd}\,(\texttt{depth=}n, \texttt{dim=}d)
\end{array}
$$

That is, the `fwd` region is irreflexive describing code which is missing a zero-offset neighbourhood index (*e.g.*, missing `a(i)`), but the addition of the `bwd` has no irreflexive attribute and thus provides the missing zero offset index.

The equation (IRREFL) provides an annihilation of `irrefl` using a further syntactic sugar. We write $R^{\texttt{irrefl}}$ to denote a region constant with an irreflexive attribute, *e.g.*,

$$\texttt{fwd}\,(\texttt{depth=}n, \texttt{dim=}d)^{\texttt{irrefl}} := \texttt{fwd}(\texttt{depth=}n, \texttt{dim=}d, \texttt{irrefl})$$

The left-hand side of (IRREFL) has a sum of products of two regions R and S, where in each component of the sum one of R or S has the irreflexive attribute. These cancel to give just $R * S$.

Interaction between + and ∗ The last six equations (from (+IDEM) inclusive), explain the algebraic behaviour of + and ∗. Together, we see that + is an associative, commutative, idempotent binary operator that distributes with ∗, which is associative and commutative. This distribution of ∗ over + is key as it is used internally to give a normal form for stencil specifications akin to Disjunctive Normal Form (DNF) (where + is taken as "disjunction" and ∗ as the "conjunction"). We revisit this normalisation in Section 4.1.4.

The equational theory provides not only an axiomatic semantics for the language, which the user can use to rewrite their specification, but it is also used extensively in the inference procedure (Section 4.1) to simplify specifications.

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

$$
\frac{P_1 <:^r R_1 \quad P_2 <:^r R_2}{P_1 + P_2 <:^r R_1 + R_2}\text{CONG+} \qquad \frac{P_1 <:^r R_1 \quad P_2 <:^r R_2}{P_1 * P_2 <:^r R_1 * R_2}\text{CONG*}
$$

$$
\frac{}{\texttt{readOnce}\,R <: R}\text{REP} \qquad \frac{}{\texttt{atLeast}\,R <: \texttt{atMost}\,R}\text{BOUND}
$$

$$
\frac{P <:^r R}{\texttt{atLeast}P <: \texttt{atLeast}R}\text{CONGL} \qquad \frac{P <:^r R}{\texttt{atMost}P <: \texttt{atMost}R}\text{CONGM}
$$

$$
\frac{P <:^r R}{\texttt{readOnce}\,P <: \texttt{readOnce}\,R}\text{CONGR} \qquad \frac{R \equiv S}{R <:^r S}\text{EQ}
$$

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

Inequations: sub-specifications Figure 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) and congruent with respect to +, ∗, `atLeast`, `atMost`, and `readOnce`. The (EQ) rule connects region equations with inequations. The (OVER) rule explains the notion of spatial over-approximation via overlapping regions. The (REP) allows the modifier that enforces that every index is read just once to be dropped as an approximation; (BOUND) gives a connection between the `atLeast` and `atMost` approximation modifiers.

3. Semantics of specifications; a model

We define a denotational model of the semantics of our stencil specification language. This model has a number of purposes: (1) it is used in the inference, checking, and synthesis algorithms (Section 4); (2) justifies the equational theory and approximations of the previous section; and (3) can be used to guide correct implementations. The model domain is over sets of vectors which we call *index schemes*, which we introduce first before the model in Section 3.2.

3.1 Indexing schemes and generalisation

Definition 4. 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 offsets of a neighbourhood index (indexing expressions of the form $i + a$ where i is an induction variable and a is a constant offset, Def.3, p.2). The additional value ∞ represents any indexing behaviour— it is a wildcard. Throughout u, v, w range over index schemes and u_i denotes the i^{th} element of u. We write $(\mathbb{Z}_\infty)^n$ for the set of n-dimensional schemes.

Indexing schemes give an abstract representation of index expressions in the source language. Our model represents the meaning of stencil specifications as indexing schemes, which explain the range of possible indexing behaviours allowed under the specification. The relationship between indexing schemes and source language terms is given precisely by the partial function, schematic, which maps syntactic array-index terms to indexing schemes. This operation is used by the inference and the checking procedures.

Definition 5. The partial function $\mathsf{schematic}_I$ (parameterised by a set of induction variables I) maps source indexing terms $(e_1, .., e_n)$ to n-dimensional index schemes, defined:

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

where s_I is partial and maps individual indexing expressions to indexing scheme components in $\mathbb{Z}_\infty$ defined:

$$\mathsf{s}_I(e) = \begin{cases} a & e \equiv i + a \ \wedge \ \mathsf{neigh}_I(e) \\ -a & e \equiv i - a \ \wedge \ \mathsf{neigh}_I(e) \\ \infty & \mathsf{IV}(e) = \emptyset \\ \bot & \textit{otherwise} \end{cases}$$

where $\mathsf{IV}(e)$ is the set of free induction variables in an expression, and neigh_I classifies expressions which are constant offsets from an induction variable $i \in I$ (Defn. 3). In the third case, if the expression is not a neighbourhood index but is not a relative index either (*i.e.*, not defined in terms of an induction variable) then it is mapped to ∞, otherwise $\mathsf{s}_I(e)$ is undefined (returns $\bot$). The schematic function is undefined if s_I of any of its components e_i is undefined. We omit the parameter set of induction variables I to schematic when clear from the context.

Definition 6 (Generalisation)**.** Given two schemes v, w of dimensionality n, then w is said to *generalise* v (or conversely v *specialises* w), written $v \preceq_n w$, iff the following holds:

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

That is, two schemes are related by the (reflexive and transitive) binary relation $\preceq_n$ over $(\mathbb{Z}_\infty)^n$ if in each dimension the two schemes are equal or $w_i = \infty$ in place of some $\mathbb{Z}$ in v_i for $v \preceq_n w$. Thus, the element ∞ generalises any index in the same dimension.

Example 7. The following three array indexing terms $(i, j+1)$, $(1, j+1)$ and $(3, j)$ are mapped by schematic to the index schemes $[0\ 1]$, $[\infty\ 1]$ and $[\infty\ 0]$ respectively. The following relationships of generalisation then hold: $[\infty\ 0] \ \not\preceq_2 \ [0\ 1]$ and $[\infty\ 1] \ \preceq_2 \ [0\ 1]$.

As an example of undefined schematic, $\mathsf{schematic}(i{*}2, j) = \bot$ as its first index is a non-neighbour relative index, which is outside the scope of our specification language.

Our terminology of *indexing schemes* and *generalisation* is by analogy with the *type schemes* and *type generalisation* of polymorphic types in ML (Milner 1978). Our notion is much more rigid however, as our schemes do not bind names.

From schematic we now give a formal characterisation of stencil computations for our purposes:

Definition 8 (Stencil computations)**.** Let a be an array of dimensionality n and $\bar{b}$ a collection of arrays of arbitrary (possible 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. Elements of a are determined by an assignment $a(\bar{e}) = e_r$ where $\bar{e}$ is an array index such that $\mathsf{schematic}_I(\bar{e}) \neq \bot$ and for all subscripts $b(\bar{e}') \in e_r$. $\mathsf{schematic}_I(\bar{e}') \neq \bot$.

3.2 Denotational model

The model is given by the interpretation function $[\![-]\!]_n$ (where n is the maximum dimensionality of the specification being modelled) mapping closed[2] specifications to sets of n-dimensional indexing schemes. The interpretation is overloaded on *regions* in Figure 5 and on the top-level of a specification *spec* in Figure 6.

We first define some intermediate notations and definitions.

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

[2] 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*.

$$\begin{aligned}
[\![\texttt{fwd(depth=}k\texttt{, dim=}d\texttt{, }r\texttt{)}]\!]_n &= \{i\mathbf{J}^d_n \mid i \in \{1, ..., k\}\} \cup \{\mathbf{K}^d_n \mid r\} \\
[\![\texttt{bwd(depth=}k\texttt{, dim=}d\texttt{, }r\texttt{)}]\!]_n &= \{i\mathbf{J}^d_n \mid i \in \{-k, ..., -1\}\} \cup \{\mathbf{K}^d_n \mid r\} \\
[\![\texttt{cen(depth=}k\texttt{, dim=}d\texttt{, }r\texttt{)}]\!]_n &= \{i\mathbf{J}^d_n \mid i \in \{\text{-}k, ..., k\} \setminus 0\} \cup \{\mathbf{K}^d_n \mid r\} \\
[\![\texttt{refl (dim=}d\texttt{)}]\!]_n &= \{\mathbf{K}^d_n\} \\
[\![r_1 * r_2]\!]_n &= [\![r_1]\!]_n \otimes [\![r_2]\!]_n \\
[\![r_1 + r_2]\!]_n &= [\![r_1]\!]_n \cup [\![r_2]\!]_n
\end{aligned}$$

Figure 5. Model of regions, $[\![-]\!]_n : \textit{region} \to \mathcal{P}({\mathbb{Z}_\infty}^n)$

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

The first four equations of Fig. 5 give the model of the region constants as sets of indexing schemes. We illustrate with an example.

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

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

Note the absorbing behaviour of ∞ with respect to multiplication.

The final two equations of Fig. 5 give the models of + and ∗. For + this is the straightforward union of the models of subterms. The model of ∗ takes the tensor $\otimes$ of subterm models. The tensor is non-trivial; we give some informal intuition first.

$\otimes$ ***of models*** The intuition behind $\otimes$ is that it takes all possible pairs of indexing schemes, treating them as the lower and upper bounds of an n-dimensional rectangle from which the remaining vertices of the 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 (*i.e.*, all possible non-deterministic choices between u_i and v_i in each dimension). Some care is then taken over how this interacts with ∞: pairwise permutation corresponds to non-deterministic choice, in each dimension i, between two values u_i and v_i, but if either is ∞ then the resulting pairwise permutation is filtered out if the dimension i has *any other scheme in the model where* $u_i \neq \infty$, *e.g.*:

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

where $[1\ \infty]$ and $[3\ \infty]$ have been filtered. The notion of which dimensions i have that *any scheme in the model has* $u_i \neq \infty$ is represented by the *constrained dimensions* intermediate function:

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

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

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

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

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

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

with interpretation on modifiers to Mult and Approx injections:

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

Figure 6. *spec* model, $[\![-]\!]_n : spec \to \mathsf{Mult}(\mathsf{Approx}(\mathcal{P}(\mathbb{Z}_\infty{}^n)))$

The first guard takes the Cartesian product of the models and quantifies over all pairings (u, v). The second guard binds j which ranges over the constrained dimensions for M union 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 corresponds 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 which have do not have ∞ in the constrained dimensions given by each model ($x_j \neq \infty$). This filters out all permutations which have ∞ in a constrained dimension position.

Model of spec We have explained the model of regions as sets of index schemes, that is $[\![region]\!]_n \in \mathcal{P}(\mathbb{Z}_\infty{}^n)$. Figure 6 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 injection function given by their definition:

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

i.e., lower is an injection $\mathsf{lower} : a \to \mathsf{Approx}\ a$, and so on. The Mult type corresponds to the presence or absence of the `readOnce` modifier as shown in Figure 6. 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 which is provided by both (but which isn't used here). These are used for labelled models, and other components later in Section 4.

3.3 Soundness

Given the above model, we can consider soundness for the equational theory of specifications and the notion of spatial approximation. The proofs are given in our supplement.

Theorem 15 (Soundness of equational theory (Figure 3))**.**

$$S \equiv T \ \Rightarrow\ [\![S]\!]_n = [\![T]\!]_n$$

Theorem 16 (Soundness of inequations (Figure 4))**.** Let $\leq$ be an ordering on $\mathsf{Mult}(\mathsf{Approx}(\mathcal{P}(\mathbb{Z}_\infty{}^n)))$ and $\leq'$ be an ordering on $\mathsf{Approx}(\mathcal{P}(\mathbb{Z}_\infty{}^n))$ defined by the following clauses:

$$(M \leq' N \Rightarrow inj\, M \leq inj\, N) \quad (\mathsf{once}\, M \leq' \mathsf{mult}\, M)$$
$$(M \subseteq N \Rightarrow inj'\, M \leq' inj'\, N) \quad (\mathsf{exact} M \leq \mathsf{lower} M)$$
$$(\mathsf{lower} M \leq \mathsf{upper} M)$$

where *inj* ranges over the unary injections of Approx and *inj*′ ranges over the injections of Mult. Then, soundness of $<:$ is that:

$$S <: T \ \Rightarrow\ [\![S]\!]_n \leq [\![T]\!]_n$$

4. Inference, checking, and synthesis

We briefly sketch the algorithms for inferring and synthesising specifications (Section 4.1), for checking code against specifications (Section 4.2). These utilise the semantic model of Section 3.

Syntax helper functions The function rhsExp maps source statements to a set of expressions which occur in right-hand positions (*i.e.*, not the target of an assignment). The function var maps expressions to a set of the variables used in right-hand positions.

4.1 Inference and specification synthesis

We demonstrate the inference procedure over the following example program which computes the mean value of a five-point stencil:

```
1  do i = 1, (n-1)
2    do j = 1, (m-1)
3      x = a(i-1, j)+a(i+1, j); y = a(i, j-1)+a(i, j+1)
4      b(i, j) = (a(i, j) + x + y) / 5.0
5    end do
6  end do
```

4.1.1 Step 1: Analyse standard control and data-flow

The inference relies on some standard program analyses, computed before the main inference procedure: **(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, where we use the predicate array to classify variables of array type.

4.1.2 Step 2: Analyse data-access

For each assignment statement whose left-hand side is an array subscript on neighbourhood indices, a finite map is computed which maps array variables to a set of vectors representing array subscripts. This finite map contains all array subscript expressions which flow to this statement. More formally, a function *analyse* is applied to each statement in a program with the following clause:

$$\mathit{analyse}(v(\overline{e_1}) = e_2) := \qquad \textit{where}\ \mathsf{neigh}(\overline{e_1}) \wedge \mathsf{array}(v)$$
$$\bigcup\{v' \mapsto \{\mathsf{schematic}(\bar{e})\} \mid v'(\bar{e}) \leftarrow \mathsf{flowsTo}(e_2), \mathsf{array}(v')\}$$

That is, we focus on assignments to an array subscript where the LHS indexing expression $\overline{e_1}$ is a neighbourhood index. 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 set of indexing schemes for its subscripts, computed with schematic (Defn. 5, p. 4). Note that schematic is undefined if $\bar{e}$ contains relative indices which are not neighbourhood indices (*i.e.*, these are not stencil computations we can handle).

For our example, the *analyse* function matches on line 5, with the following set for flowsTo(`a(i, j) + x + y`):

$$\{\texttt{a(i-1, j)}, \texttt{a(i+1, j)}, \texttt{a(i, j-1)}, \texttt{a(i, j+1)}, \texttt{a(i, j)}\}$$

Subsequently the result of *analyse* on line 5 yields the map:

$$\mathtt{a} \mapsto \{[-1\ 0], [1\ 0], [0\ -1], [0\ 1], [0\ 0]\}$$

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)`.

4.1.3 Step 3: Coalesce contiguous schemes into spans

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

Definition 17 (Spans)**.** A *span* represents an n-dimensional box (*hyper-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 a lower-bound vertex (first component) and the upper-bound (second component). 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 subspan comprises the subvectors $[\mathbf{x}^L_{i:j}\ \mathbf{x}^U_{i:j}]$ from entry i to entry j (inclusive).

Each indexing scheme in $U(x)$ is converted to a *unit* span $u \mapsto [u\ u]$, defining the lower and upper bound. For the five-point stencil example there are five 1×1 spans. Figure 7(a) illustrates these.

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 $\mathbf{x}^U_1 + 1 = \mathbf{y}^U$. 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 18. Span coalescing $\bullet$ is defined for spans contiguous in the first dimension:

$$\mathbf{x} \bullet \mathbf{y} = \begin{cases} (\mathbf{x}^L, \mathbf{y}^U) & \mathbf{x}^U_1 + 1 = \mathbf{y}^L_1 \ \wedge\ \mathbf{x}_{2:n} = \mathbf{y}_{2:n}) \\ \bot & \textit{otherwise} \end{cases}$$

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 Figure 7(a) are coalesced into the contiguous spans in Figure 7(b). This procedure is iterated until a fixed point is reached. In this example this is reached in the first step. 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), Figure 7(c). The final spans are used to synthesise concrete stencil specification syntax (Section 4.1.4), where each span becomes a product $*$ of regions which are combined by the sum `+`.

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

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

In the following we let $M_0 = N(U(\mathtt{a}))$ 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. For our example:

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

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

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

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

giving the sorting function $\mathsf{sort}(M)$. For our example:

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

3) Reduce each list pairwise by $\bullet$ operation (Defn. 18) to coalesce contiguous regions, given by $\mathsf{fold}\bullet$. For our example, this yields:

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

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 $\sqsubseteq$, that is, if any region is contained within another then remove the smaller, defined:

$$\mathbf{x} \sqsubseteq \mathbf{y} = \mathbf{y}^L_1 \leq \mathbf{x}^L_1 \wedge \mathbf{x}^U_1 \leq \mathbf{y}^U_1 \wedge (\mathbf{x}^L_{2:n}, \mathbf{x}^U_{2:n}) \sqsubseteq (\mathbf{y}^L_{2:n}, \mathbf{y}^U_{2:n})$$

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; the fixed point is reached.

4.1.4 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 Defn. 14, p. 6), which we write as A for brevity. The algebra mirrors the shape of the region combinators for sum and product, with operators $\oplus$ and $\odot$ and a constructor for **spec**:

$$\oplus : \mathsf{A}(\mathbf{spec}) \to \mathsf{A}(\mathbf{spec}) \to \mathsf{A}(\mathbf{spec})$$
$$\odot : \mathsf{A}(\mathbf{spec}) \to \mathsf{A}(\mathbf{spec}) \to \mathsf{A}(\mathbf{spec})$$
$$\mathsf{toSpec} : (\mathbb{Z} \cup \{\infty\}) \to (\mathbb{Z} \cup \{\infty\}) \to \mathsf{A}(\mathbf{spec})$$

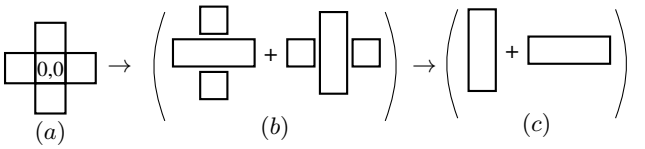


Figure 7. Illustration of grouping spans in specification inference

$\mathsf{toSpec}\ l\ u =$

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

Figure 8. toSpec constructor of A(**spec**) values.

The toSpec function maps a pair of lower and upper bound values (one-dimensional) to a specification with a one-dimensional region. An entire 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 specifications by $\odot$. The resulting specifications for each span are then summed by $\oplus$:

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

We unpack the definitions of this algebra.

Definition 19 (Specifications in DNF)**.** Let $r_1, ..., r_n$ range over elements of the *region* syntax (see Figure 2, p. 3). The set **spec** is a subset of syntax trees *spec*, containing specifications in disjunctive normal form over regions where the disjunct is + (since its model is in terms of set union) and the conjunct is $*$ (since its model is a form of tensor). That is elements of **spec** are of the form: $(r_1^1 * \ldots * r_n^1) + \ldots + (r_1^m * \ldots * r_p^m)$.

More formally, **spec** $\subseteq (\mathit{spec} \cup \{\epsilon\})$, defined:

$$\begin{aligned} \textbf{spec} &::= \textbf{spec}^+ \mid \epsilon \\ \textbf{spec}^+ &::= \textbf{spec}^* + \textbf{spec}^+ \mid \textbf{spec}^* \\ \textbf{spec}^* &::= \mathit{region} * \textbf{spec}^* \mid \mathit{region} \end{aligned}$$

We include the empty element ϵ which is not present in the syntax. In the following, S, T range over elements of **spec** and s^*, t^* range over elements of **spec*** (for example, $s^* = r_1^1 * \ldots * r_n^1$).

Definition 20. The toSpec constructor is defined in Figure 8. The first two cases map to `forward` stencils since the lower bound is 0 or 1, with the depth determined by the upper bound of the span. If the lower bound is 1, then the 0-offset point is missing so the region is given an `irreflexive` attribute. The second two cases are similar for `backward` but now the depth of the region is determined by the absolute of the lower bound. In the fifth-case, where 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 (sixth 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 instead returned 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.

Finally, if either bound is ∞, representing a non neighbourhood index, then an empty specification is given with no regions, indicating that there is not even reflexive 0-offset access in this dimension.

Note, $\mathsf{toSpec}\ l\ u$ is always defined if l less than or equal to u.

Definition 21. The $\odot$ operation on A(**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 our equational theory (Fig. 3,p. 4). The operation is commutative and associative, and sound with respect to our model (see Lemma 23 below). The $\hat{\odot}$ operation is then lifted to all combinations of Approx for $\odot$:

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

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 22. The $\oplus$ operation on A(**spec**) is defined in a similar way to $\odot$, with the intermediate $\hat{\oplus}$ on **spec** given by:

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

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 the previous definition.

Lemma 23 ($\hat{\odot}$, $\hat{\oplus}$ soundness)**.** $\forall S, T \in$ ***spec*** *where* $S \neq \epsilon \wedge T \neq \epsilon$*:*

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

Thus our model validates the specification synthesis process which concludes the inference procedure. For our example, we infer and then synthesise the 5-point specification:

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

4.2 Specification checking

In checking mode, the access pattern of a stencil computation in the source language is verified against a specification. The model of Section 3 again aids this process. Checking proceeds by generating a model M from a specification and generating a set N of indexing schemes from the source code. The source code is then valid with respect to the specification if N is *consistent* with M. The notion of consistency is built out of index scheme *generalisation* (Sect. 3.1). We first set up some intermediate definitions.

Definition 24 (Checking)**.** Let `a` be an array variable and let $S \in \mathit{spec}$ be a specification with the *specDec* `!= stencil` S `:: a` in our syntax. For a array subscript assignment statement $\mathit{assg} = v(\overline{e_1}) = e_2$ in our source language (with all the appropriate data flow information in place) where e_2 has at most n-dimensional array access, then the assignment conforms to the specification S if

$$cons_n(\mathsf{once}(A_s), \mathsf{once}(M_c)) = cons_n(\mathsf{mult}(A_s), \mathsf{mult}(M_c))$$
$$cons_n(\mathsf{mult}(\mathsf{up}(M_s)), \mathsf{mult}(M_c)) = \forall u \in M_c, \exists v \in M_s . u \preceq_n v$$
$$cons(\mathsf{mult}(\mathsf{low}(M_s)), \mathsf{mult}(M_c)) = \forall v \in M_s, \exists u \in M_c . u \preceq_n v$$
$$cons_n(\mathsf{mult}(\mathsf{exact}(M_s)), N_c) = cons_n(\mathsf{mult}(\mathsf{low}(M_s)), N_c) \wedge cons_n(\mathsf{mult}(\mathsf{up}(M_s)), N_c)$$
$$cons_n(\mathsf{mult}(\mathsf{both}(M_1, M_2)), N_c) = cons_n(\mathsf{mult}(\mathsf{low}(M_1)), N_c) \wedge cons_n(\mathsf{mult}(\mathsf{up}(M_2)), N_c)$$

Figure 9. Consistency between a model and indexing schemes

the predicate $check_n(S, assg, \mathtt{a})$ holds, defined:

$$check_n(S, assg, x) = cons_n([\![S]\!]_n, \mathit{modelise}(assg, x))$$
$$\mathit{modelise} : \mathit{statement} \times \mathit{variable} \to \mathsf{Mult}((\mathbb{Z}_\infty)^n)$$
$$\mathit{modelise}(assg, x) = \begin{cases} \mathsf{once}(\mathit{analyse}(assg)(x)) & \mathit{unique}(assg, x) \\ \mathsf{mult}(\mathit{analyse}(assg)(x)) & \mathit{otherwise} \end{cases}$$

Thus, checking proceeds by reusing the *analyse* function from inference (Sec. 4.1) to extract index schemes from source code. This is then augmented with multiplicity information by $\mathit{modelise}$ where $\mathit{unique}(assg, x)$ is a predicate indicating whether indices to array x in an assignment *assg* are unique or not.

Subsequently the binary predicate for consistency, at the heart of checking is defined over the sets:

$$cons_n \subseteq \mathsf{Mult}(\mathsf{Approx}(\mathbb{Z}_\infty{}^n)) \times \mathsf{Mult}(\mathbb{Z}_\infty{}^N)$$

where first component is a model of a specification, and the second component is a set of indexing schemes generated from the array subscripts in a program, with added multiplicity information. Giving Approx information to the index schemes inferred from the program is not necessary, nor desirable– we have an exact set of schemes. We refer to the first component of the predicate as *spec model* and the second as *code model*.

Definition 25. Consistency is defined in Figure 9. We let M_s, M_c range over elements of $\mathbb{Z}_\infty{}^n$ where M_s comes from the spec model and M_c comes from *modelise* on the source code. Let N_c range over elements of $\mathsf{Mult}(\mathbb{Z}_\infty{}^n)$ coming from the code model, and A_s over elements of $\mathsf{Mult}(\mathsf{Approx}(\mathbb{Z}_\infty{}^n))$ from the spec model.

The first equation states that, if the multiplicity of the spec and code model are both once, then the problem is reduced to checking consistency of models in which non-unique indexing is allowed, labelled with mult.

The second equation considers the case where the specification is an `atMost` upper-bound approximation (labelled with up as shorthand for upper here). An upper bound specification means that for every scheme in the code model, modelling a source-level index, $\forall u \in M_c$ there exists a generalising scheme in the spec model $\exists v \in M_s$, *i.e.*, $u \preceq_n v$. That is, every source-level index is generalised by the model.

The third equation is the converse of an `atLeast`, lower approximation (labelled low as shorthand for lower). Consistency is dual to the above: for every scheme in the spec model $\forall v \in M_s$ there exists a scheme in the code model $\exists u \in M_c$ such that $u \preceq_n v$. That is, every model scheme is generalised by source indexing.

These two are combined for consistency on an exact specification (the fourth equation) such that the spec 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 spec model there is a scheme in the source model that specialises it.

Lastly, if the spec model has both an upper and lower bound consistency is defined inductively similar to the exact case.

Example 26. We give an extended example of verifying a stencil computation against its specification. We consider a more general, but still consistent, version of the *five-point* stencil specification inferred previously in Section 4.1:

```
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 the assignment on line three as $l3$ for brevity.

Verification of the code against the spec is then given by $check((\texttt{readOnce, fivepoint}), l3, \mathtt{b})$. We explain the steps.

1. Generate the model from the specification as in Section 3.
$$M_s = \{\begin{bmatrix}-1\\ \infty\end{bmatrix}, \begin{bmatrix}0\\ \infty\end{bmatrix}, \begin{bmatrix}1\\ \infty\end{bmatrix}, \begin{bmatrix}\infty\\ -1\end{bmatrix}, \begin{bmatrix}\infty\\ 0\end{bmatrix}, \begin{bmatrix}\infty\\ 1\end{bmatrix}\}$$
$$[\![\texttt{readOnce, fivepoint}]\!]_2 = \mathsf{once}(\mathsf{exact}(M_s))$$
2. Obtained the code model via *modelise*, giving:
$$M_c = \{\begin{bmatrix}0\\ -1\end{bmatrix}, \begin{bmatrix}0\\ 0\end{bmatrix}, \begin{bmatrix}0\\ 1\end{bmatrix}, \begin{bmatrix}-1\\ 0\end{bmatrix}, \begin{bmatrix}1\\ 0\end{bmatrix}\}$$
$$\mathit{modelise}(l3, \mathtt{b}) = \mathsf{once}(M_c)$$
3. Consistency $cons_2(\mathsf{once}(\mathsf{exact}(M_s)), \mathsf{once}(M_c))$ holds when $cons_2(\mathsf{mult}(\mathsf{exact}(M_s)), \mathsf{mult}(M_c))$ (Fig. 9) which holds when:
$$cons_2(\mathsf{mult}(\mathsf{up}(M_s)), \mathsf{mult}(M_c)) \wedge cons_2(\mathsf{mult}(\mathsf{low}(M_s), \mathsf{mult}(M_c))$$
4. This condition holds by the following sets of scheme generalisations: for the upper bound, $\forall u \in M_c, \exists v \in M_s . u \preceq_2 v$:
$$\begin{array}{lll} [0\ -1] \preceq_2 [0\ \infty] & [0\ 0] \preceq_2 [0\ \infty] & [0\ 1] \preceq_2 [0\ \infty] \\ [-1\ 0] \preceq_2 [\infty\ 0] & [1\ 0] \preceq_2 [\infty\ 0] & \end{array} \quad (1)$$
and for the lower bound: $\forall v \in M_s, \exists u \in M_c . u \preceq_n v$:
$$\begin{array}{lll} [-1\ 0] \preceq_2 [-1\ \infty] & [0\ 0] \preceq_2 [0\ \infty] & [1\ 0] \preceq_2 [1\ \infty] \\ [0\ -1] \preceq_2 [\infty\ -1] & [0\ 0] \preceq_2 [\infty\ 0] & [0\ 1] \preceq_2 [\infty\ 1] \end{array} \quad (2)$$
Since both 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 27. We consider now an erroneous user modification to $l3$ with an additional term `b(i+2,j+2)` such that $l3'$ is:

```
3  a(i,j) = b(i,j-1) + b(i,j) + b(i,j+1) + b(i-1,j) +
   ↪  b(i+1,j) + b(i+2,j+2)
```

The checking procedure proceeds in the same way as the first three steps of the previous example, except that the result of $\mathit{modelise}(l3', \mathtt{b})$ generates an additional scheme [2 2]. Since the spec model has not changed and the code model is has only the additional scheme [2 2], the generalisations for the lower bound (2) of step 4 above still holds since its consistency condition universally quantifies over the *spec model*. However, for the upper bound, the consistency condition universally quantifies over scheme u of the *code model* for which there exists a scheme $v \in M_s$ such that $u \preceq_2 v$. Thus, we require a $v \in M_s$ such that $[2\ 2] \preceq_2 v$ but no such generalising scheme existing in the spec model. Hence the model fails to hold as an upper bound.

Note that if the specification were as follows:

```
1  != stencil atLeast, readOnce, fivepoint :: b
```

then consistency holds as it no longer requires the failing upper-bound case $cons(\mathsf{mult}(\mathsf{up}(M_s)), \mathsf{mult}(M_c))$. This makes sense semantically since the region `fivepoint` is indeed contained in the regions touched by `b` in $l3'$.

4.3 Approximations and checking

As noted, the `fivepoint` region above is more accepting than that used in Section 2. This introduces a new notion of approximation beyond what is given by the syntactic theory of sub-specifications $<:$ in Section 2.2. Consistency leads to a notion of approximation that is a *semantic sub-specification* relation:

Definition 28 (Approximation)**.** A specification S' *approximates* a more specific specification S, written $S \sqsubseteq S'$, *iff*:

$$\forall M_c \in \mathsf{Mult}(\mathbb{Z}_\infty{}^n).\ cons_n([\![S]\!]_n, M_c) \implies cons([\![S']\!]_n, M_c)$$

For the syntactic sub-specification equations on regions $<:^r$, it follows that these imply semantic approximation:

Lemma 29 (Sub-specification $<:^r$ coincides with approx $\sqsubseteq$)**.**

$$\forall S, S'.\ S <:^r S' \Rightarrow S \sqsubseteq S'$$

However, the converse direction does not hold. The regions describing the more accepting five-point stencil above do not follow from the definition of $<:$ on the previous more specific specification.However, consistency (and thus $\sqsubseteq$) shows that we can check the five-point stencil code against the more general specification of `fivepoint`. Thus, we can always use an approximation in place of the more specific specification. Although this decreases the likelihood of catching stencil errors (as more access patterns conform), it reduces the space and effort it takes to write the specification.

5. Evaluation

We evaluated our specification language on a corpus of scientific code. We overview the corpus itself before presenting our results.

Software Corpus Table 1 shows summary statistics of the software packages used in our evaluation, all of which are written in Fortran 90 or Fortran 77. In total we analysed 1,165,390 lines of code from 7 packages, of which we successfully parsed 803,829 lines. The "Number of files" column shows how many files in each corpus that we were able to analyse with CamFort. The most common reasons for CamFort rejecting a file were either use of a C-preprocessor, or illegal use of language features from a modern Fortran variant.

The Unified Model (Wilson and Ballard 1999) is a weather forecasting and climate modelling tool developed by the Met Office in the United Kingdom. It is used by research organisations and meteorological services around the world. We use the development branch (trunk) of the model. The code base in closed source but institutional licenses are available for research purposes.

E3MG (An Energy-Environment-Economy (E3) Model at the Global Level) is a macroeconomic model used for assessment of environmental policy (Barker et al. 2006). This was developed by Cambridge Econometrics, an independent consultancy company.

BLAS (Blackford et al. 2002) (Basic Linear Algebra Subprograms) is a popular library providing efficient and portable routines for vector and matrix operations. These routines feature in many other libraries (including LAPACK). We used version 3.6.0. We chose to include this package for breadth, as it provides general numerical functions rather than a specialised scientific model.

Hybrid4 is a vegetation and biomass model for simulating carbon, water and nitrogen flows (Friend and White 2000).

GEOS-Chem (Bey et al. 2001) is a three-dimensional model of tropospheric chemistry developed at Harvard and used by ∼70 universities and research institutions world-wide. We use v.10-01.

Navier is a small numerical simulation, giving a discrete approximation to the two-dimensional Navier-Stokes fluid equations, based on the book of Griebel et al. (1997).

CP consists of the example code from the second edition of the book "Computational Physics" (Nicholas J. Giordano 2006)

	Number of files		Lines of code	
Package	Total	Parsed	Total	Parsed
UM	2,533	2,251	619,776	526,998
E3MG	139	120	52,296	43,336
BLAS	151	147	16,046	15,946
Hybrid4	29	20	4,831	2,765
GEOS-Chem	596	182	471,138	213,519
Navier	7	7	510	510
CP	16	15	793	755
Total	3,471	2,742	1,165,390	803,829

Table 1. Summary of software packages used for evaluation

	Potential		Actual		Exact	Inexact
Package	#	%	#	%		
UM	39,353	7.5 %	21,640	4.1 %	30,269	74
E3MG	4,483	10 %	2,458	5.7 %	3,223	5
BLAS	1,298	8.1 %	626	3.9 %	700	0
Hybrid4	34	1.2 %	18	0.65 %	20	0
GEOS-Chem	35,191	16 %	1,137	0.53 %	1,281	0
Navier	45	8.8 %	30	8.8 %	39	0
CP	97	13 %	40	5.3 %	62	0
Total	80,501	10 %	25,949	3.2 %	35,594	79

Table 2. Stencil specification assignment rates

introducing numerical techniques and their application to modern physics problems such as fields, waves, statistical mechanics and quantum mechanics.

5.1 Occurrences of stencils

We first examined how frequently stencil computations occur in our corpus. This was carried out by running our implementation in whole-code-base inference mode. We define a *potential* statement as one whose left-hand side is an array subscript on neighbourhood indices. These statements are the inputs to Step 2 of the Inference process described in Section 4.1. Each potential statement for which we eventually generate some specification is also counted as an *actual* statement.

The first two columns of Table 2 show the number of potential and actual stencils and their rate as a percentage of the total number of successfully parsed lines of code. The ratio of actual to potential stencils overall is thus 32%, *i.e.*, we give a specification to roughly a third of the potential stencil specifications.

There are a variety of reasons why we might not infer a actual specification from a potential stencil 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 (Defn. 8,p. 5). 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 the indexing variable (`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)`.

If the statement meets the stencil criteria but the inference process cannot determine an exact stencil for a statement it will instead produce an upper (`atMost`) and lower (`atLeast`) bound. The number of exact and inexact stencils are shown in columns three and four of Table 2. Note that the total for "Exact" is higher than "Actual" because for some lines we produce multiple specifications (perhaps for different array variables). We see that our specification language is capable of precisely specifying the overwhelming majority of stencils. Most of the inexact stencils apply to some form of data copying. For example (taken from **E3MG**):

```
!- stencil atLeast, readOnce, irreflexive(dims=2),
 ↪  (reflexive(dim=1)) :: sfda
!- stencil atMost, readOnce, (forward(depth=8,
 ↪  dim=2))*(reflexive(dim=1)) :: sfdt
ZZ1(I,J) = SFDT(I,J+8)
```

5.2 Stencil categories

We grouped our inferred specifications together into four classes based on similar levels of complexity. These four classes account for 99.4% of all the stencils we found in the corpus.

All-reflexive The vast majority (66%) of specification are reflexive in every dimension. These most often correspond to pointwise transformations on data such as scaling, *e.g.* from **E3MG**:

```
ZZ1(I,K) = .000041868*ZZ1(I,K)
```

Arrays with this class of stencil are indexed by all the induction variables used in the LHS without any offsets.

Some-reflexive specifications (24%) consist of only reflexive terms but leave some dimensions unspecified. These two examples are from **GEOS-Chem** and **E3MG**:

```
leafpools(:,age_class)=leafpool(:,1)
RVM(J,I2) = VRFO(ivar,J)
```

where for the first example `leafpools` contains an implicit loop due to the range term `:`, and for the second `J` is the induction variable. This stencil permits array indexing by a subset of the LHS induction variables but again without any offsets.

Single-action specifications (6%) consist of a single `backward`, `forward`, or `centered` region and some number of `reflexive` region constants. For example, taken from **GEOS-Chem**:

```
do j=2,jm-ig
   do i=1,im
      != stencil readOnce, (backward(depth=1, dim=2))
 ↪  *(reflexive(dim=1)) :: dm, q
       al(i,j) = 0.5e+0_fp*(q(i,j-1)+q(i,j)) +
 ↪  r3*(dm(i,j-1) - dm(i,j))
```

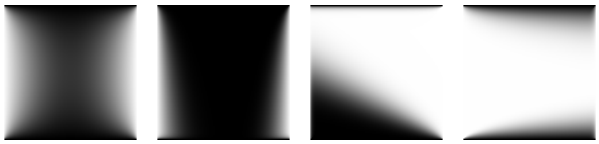
These kinds of specification require zero offsets in one dimension and and neighbourhood offsets (from 0 to -1 above) in the other. The remaining dimensions must either have no offset or not pertain to an induction variable.

Single-action-irreflexive specifications (3%) consist of a single `backward`, `forward`, or `centered` irreflexive region and some number of reflexive region constants, *e.g.*, from **GEOS-Chem**:

```
do lev = nlayers, 1, -1
    ! ...
    radld = radld - radld * (atrans(lev) + &
       efclfrac(lev,igc) * (1. - atrans(lev))) + &
       gassrc + cldfmc(igc,lev) * &
       (bbdtot * atot(lev) - gassrc)
    != stencil readOnce, forward(depth=1, dim=2,
 ↪  irreflexive) :: cldfmc
    drad(lev-1) = drad(lev-1) + radld
```

Note that the read from `cldfmc` on line 5 flows to line 8. Stencils of the above kind require a neighbourhood index without a zero-offset in one dimension only. The remaining dimensions must either have no offset or not pertain to an induction variable.

The remaining specification shapes (less than 1%) consisted primarily of multiple `forward`, `backward`, or `centered` dimensions. We note that only ∼0.4% of inferred stencil specifications required use of the `+` operator. Despite their low frequency these stencils are arguably the most important since they represent more complicated code and thus greater potential for programming errors.



(a) (b) (c) (d)

Figure 10. Perturbations of Jacobi code; (a) is reference output.

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

Figure 11. A complex specification inferred from **Unified Model**

5.3 Verification case study 1: 2-D Jacobi iteration

One relatively basic example of a stencil computation is 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 can be expressed in a line of Fortran code:

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

and CamFort is able to infer a precise specification for this:

```
 != stencil
↪ reflexive(dim=1)*centered(depth=1,dim=2,irreflexive)+⌋
↪ reflexive(dim=2)*centered(depth=1,dim=1,irreflexive)
```

assigned to `a`. We then tested the error-detecting ability of our approach by generating a family of 6,561 Jacobi iteration programs by perturbing the indices used in the following manner: each access of the array was templated as `a(i+ki,j+kj)` and then all possible combinations of `ki` and `kj` were drawn from the set $\{-1, 0, 1\}$. There are four uses of `ki` and four of `kj`, hence $3^8 = 6561$ possible perturbations of this kind. This particular setup tests the class of errors in which the programmer typed an incorrect number but at least got the ordering of the indices correct, although CamFort would also detect a mixup of variables in this case as well.

We assigned the specification inferred above to the generated Jacobi stencils programs. CamFort checked all 6,561 programs against this specification and found that only 24 of them were valid implementations of the stencil specification. This was the expected result as there are 4! possible correct permutations of the four terms in the kernel. All of the other programs were found to violate the specification in some way. Another way to test the programs is through visual examination of their output in graphical format. Figure 10 shows some of the results of running the Jacobi iteration programs, rendered into image files. All of the correct variants emitted images indistinguishable from Figure 10(a), while all of the incorrect variants had wildly different image outputs. Both the "visual test" and a byte-by-byte comparison showed results that completely concurred with the results of the CamFort test.

5.4 Verification case study 2: Turbulence calculation

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. We show one example in Figure 11. This specifies the access pattern to a 3-dimensional ar-

ray (which we have renamed to be called `x`). The array is accessed 8 times in the computation. We tried introducing indexing errors by changing the offset in one dimension of one of the references to `x`. For example changing `x(i+1,j,k)` to `x(i,j,k)` or `x(i-1,j,k)`.

There are 48 possible errors of this type of which only 6 were not detected by the checker. The reason is that the inferred specification describes a family of access patterns. Some offset errors produce a stencil computation that belongs to the same family and goes undetected. One of the reasons this specification can catch many offset errors is that it is a `readOnce` specification. This prevents offset errors leading to duplicate access patterns be caught even if the resulting set of accesses are part of the same family.

6. Related work and conclusions

Various prior approaches to specifying and verifying the behaviour of stencils have been based on fine-grained representations of the code. These essentially duplicate the indexing scheme of a stencil computation in their specifications.

Kamil et al. (2016) propose a technique they call *verified lifting* for mapping low-level, possibly greatly optimised, stencil computations into a high-level predicate language using inductive program synthesis. The aim is that the high-level abstract specifications can be automatically translated into high-performance DSLs. Compared to our approach, their specification language is more fine grained, capturing the exact indexing pattern of a stencil. For example, the following is a post-condition generated by their compiler STNG from a Fortran stencil computation (Kamil et al. 2016, p.3):

$$post(\mathtt{a},\mathtt{b}) \equiv \forall \mathtt{imin}+1 \leq \mathtt{i} \leq \mathtt{imax}, \mathtt{jmin} \leq \mathtt{j} \leq \mathtt{jmax}$$
$$\mathtt{a}(\mathtt{i},\mathtt{j}) = \mathtt{b}(\mathtt{i}-1,\mathtt{j}) + \mathtt{b}(\mathtt{i},\mathtt{j})$$

Notably, the bounds of the stencil and the exact indexing pattern are captured. Our approach aims to be much more abstract, to facilitate human reasoning, and avoid low-level lexical errors. However, STNG specifications are generated, *i.e.*, are never written by a programmer, avoiding this problem. Their inductive program synthesis approach allows specifications to extracted even from optimised and unrolled code. For example, the above post condition is generated from the code (Kamil et al. 2016, p.3):

```
1  do j=jmin, jmax
2      t = b(imin, j)
3      do i=imin+1,imax
4          q = b(i, j);  a(i, j) = q + t;  t = q
5      end do
6  end do
```

Thus, STNG can understand cross-loop data-dependencies in order to assign the above post-condition. CamFort is not currently able to ascribe a specification in the above case as the use of `t` on line 4 maps only to the definition on line 2, and not to that on line 5. Extending our approach to follow loop-carried dependencies is future work. An alternate approach for us would be to build on top of the work of Kamil *et al.* by inferring our specifications from STNG (as an intermediate step).

Our work has some similarities with efforts to verify kernels written for General-Purpose Graphics Processing Unit (GPGPU) programming, such as in Blom et al. (2014). Stencils are a form of kernel, and GPGPU programming can be viewed as a massively parallel method of transforming a large matrix. However, that work focuses mainly on the synchronisation of such kernels and the avoidance of data races, while we are interested in the correctness of stencils embedded within a more typical general-purpose programming language. Other work on GPGPU computation, such as Zhang and Mueller (2012), has focused primarily on generating optimised code based on relatively simple specifications: to provide performance while keeping the programmer's effort within reason.

Solar-Lezama et al. (2007) give specifications of stencils using unoptimised "reference" stencils, coupled with partial implementations of stencils, which are then completed by a code generation tool. These kinds of specifications are just simple implementations, so this tool is useful for hand-written, optimised stencils. The primary purpose of this tool is optimisation rather than correctness, and the language of specification is much different and more elaborate than we envision for our system.

Tang et al. (2011) define a specification language for writing stencils embedded in C++ (with Cilk extensions) that are then compiled into fast parallel programs based on trapezoidal decompositions with hyperspace cuts. The Pochoir specification language is used for describing the kernel, boundary conditions and shape of the stencil. Programs are first compiled in C++ with a template library that checks whether the annotations are used correctly, and then a Pochoir preprocessor may be applied to generate the high-performance parallel versions using Cilk. Pochoir is aimed primarily at assisting programmers who may be reluctant to implement the high-performance tricky cache-oblivious "hypertrapezoidal" algorithms for stencils. Like much of the related work, the main goal is optimisation rather than verifying program correctness.

By contrast, the work of Abe et al. (2013) has looked at the correctness problem explicitly by bringing a form of model-checking to verify certain stencil computations. This is in the context of decomposing stencils for parallel computation on multiple processors in partitioned global address space languages. The authors have designed a new language to be used for writing stencil computations. Much of the specification effort goes towards describing the nuts and bolts of distributing the computation over multiple processors. The code for the stencil kernel is generated from a relatively high-level specification. This approach differs from CamFort because we are interested in integrating directly into existing, legacy codebases and long-established languages such as older versions of Fortran. We infer stencil specifications from Fortran code and check annotations on stencil computations within Fortran code.

Conclusion and future work There is an increasing awareness of the need for verification techniques in science (Post and Votta 2005; Oberkampf and Roy 2010; Orchard and Rice 2014). Our specification language is an approach in this direction, providing a system of lightweight specification for numerical code. Compared to other specification approaches (discussed above), our language is novel in that it does not aim to reify indexing in the specification, but instead provides abstract spatial descriptions which capture a large number of common patterns.

We evaluated our approach on top of CamFort, an open-source analysis tool for Fortran programs, running the inference process over a corpus of 1.1 million lines of Fortran code. Our motivation for this work was to provide an intermediate specification language to allow scientists to verify numerical computations. Our evaluation showed that relatively simple stencil computations are the norm comprising mainly reflexive region constants which are less likely to give rise to programming errors. However, around 10% of stencils covered more complicated program structures for which indexing errors are much more plausible. In the more detailed cases, with larger specifications, the importance and impact of such specifications grows. We showed two examples of more complicated stencils, for which CamFort was able to detect a significant majority of possible indexing errors in our case studies.

Our definition of a stencil computation meant that we generated no specification for a large number of iterated array accesses in the corpus, such as stencils that incorporates reductions. For future work, we will be looking into expanding the flexibility of our inference approach to more of these examples. Furthermore, our specification language can be used more flexibly to give descriptions of data access patterns in any context, not just within a stencil.

References

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.

T. Barker, H. Pan, J. Kohler, R. Warren, and S. 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.

I. Bey, D. J. Jacob, R. M. Yantosca, J. A. Logan, B. D. Field, A. M. Fiore, Q. Li, H. Y. Liu, L. J. Mickley, and M. G. Schultz. Global modeling of tropospheric chemistry with assimilated meteorology: Model description and evaluation. *Journal of Geophysical Research: Atmospheres*, 106 (D19):23073–23095, 2001.

L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

S. Blom, M. Huisman, and M. 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} SACSVT 2013 + Bytecode 2013.

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

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.

A. D. Friend and A. 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.

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

S. Kamil, A. Cheung, S. Itzhaky, and A. 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.

R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

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

W. Oberkampf and C. Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

D. Orchard and A. Rice. A computational science agenda for programming language research. *Procedia Computer Science*, 29:713–727, 2014.

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

G. Recktenwald. Finite-difference approximations to the heat equation. *Class Notes*, 2004. http://www.f.kth.se/~jjalap/numme/FDheat.pdf.

A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. 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.

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. 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.

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

D. R. Wilson and S. 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.

Y. Zhang and F. Mueller. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164. ACM, 2012.