Verifying Spatial Properties of Array Computations

DOMINIC ORCHARD, University of Kent, UK
MISTRAL CONTRASTIN, University of Cambridge, UK
MATTHEW DANISH, University of Cambridge, UK
ANDREW RICE, University of Cambridge, UK

Arrays computations are at the core of numerical modelling and computational science applications. However, low-level manipulation of array indices is a source of program error. Many practitioners are aware of the need to ensure program correctness, yet very few of the techniques from the programming research community are applied by scientists. We aim to change that by providing targetted lightweight verification techniques for scientific code. We focus on the all too common mistake of array offset errors as a generalisation of off-by-one errors. Firstly, we report on a code analysis study on eleven real-world computational science code base, identifying common idioms of array usage and their spatial properties. This provides much needed data on array programming idioms common in scientific code. From this data, we designed a lightweight declarative specification language capturing the majority of array access patterns via a small set of combinators. We detail a semantic model, and the design and implementation of a verification tool for our specification language, which both checks and infers specifications. We evaluate our tool on our corpus of scientific code. Using the inference mode, we found roughly 87,000 targets for specification across roughly 1.1 million lines of code, showing that the vast majority of array computations read from arrays in a pattern with a simple, regular, static shape. We also studied the commit logs of one of our corpus packages, finding past bug fixes for which our specification system distinguishes the change and thus could have been applied to detect such bugs.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Domain specific languages*; *Software maintenance tools*;

Additional Key Words and Phrases: verification, static-analysis, scientific computing, arrays, stencils

ACM Reference Format:
Dominic Orchard, Mistral Contrastin, Matthew Danish, and Andrew Rice. 2017. Verifying Spatial Properties of Array Computations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 75 (October 2017), 30 pages. https://doi.org/10.1145/3133899

1 INTRODUCTION

In the sciences, complex models are now almost always expressed as computer programs. But how can a scientist have confidence that the implementation of their model is as they intended? There is an increasing awareness of the need for program verification in science and the possibility of using (semi-)automated tools [Oberkampf and Roy 2010; Orchard and Rice 2014; Post and Votta 2005]. However, whilst program verification approaches are slowly maturing in computer science they see little use in the natural and physical sciences. This is partly due to a lack of training and

Authors' addresses: Dominic Orchard, School of Computing, University of Kent, Canterbury, UK, CT2 7NF, d.a.orchard@kent.ac.uk; Mistral Contrastin, Computer Laboratory, University of Cambridge, Cambridge, UK, CB3 0FD, Mistral.Contrastin@cl.cam.ac.uk; Matthew Danish, Computer Laboratory, University of Cambridge, Cambridge, UK, CB3 0FD, Matthew.Danish@cl.cam.ac.uk; Andrew Rice, Computer Laboratory, University of Cambridge, Cambridge, UK, CB3 0FD, Andrew.Rice@cl.cam.ac.uk.

2475-1421/2017/10-ART75
https://doi.org/10.1145/3133899

awareness, but also a lack of tools targeted at the needs of scientists. We aim to change that. This paper is part of a line of research providing lightweight, easy-to-use verification tools targeted at common programming patterns in science, motivated by empirical analysis of real code.

We focus on one common concept: *arrays*, the core data structure in numerical modelling code, typically used to represent discrete approximations of physical space/time or to store data sets. A common programming pattern, sometimes referred to as the *structured grid* pattern [Asanovič et al. 2006], traverses the index space of one or more arrays via a loop, computing elements of another array (also called a *stencil computation*) or reducing the elements to a single value. For example, the following Fortran code computes the one-dimensional discrete Laplace operator for an array:

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

This is an example stencil computation, where elements of an array at each index i are computed from a *neighbourhood* of values around i in some input array(s). Stencils are common in scientific, graphical, and numerical code, *e.g.*, convolutions in image processing, approximations to differential equations in modelling, and cellular automata.

Such array computations are prone to programming errors in their indexing terms. For example, a logical off-by-one-error might manifest itself as writing `a(i)` instead of `a(i-1)` (we revisit examples we found of this in Section 7.3). Errors also arise by simple lexical mistakes when large amounts of fine-grained indexing are involved in a single expression. For example, the following snippet from a Navier-Stokes fluid model [Griebel et al. 1997] has two arrays (`u` and `v`) which are read with different data access patterns, across two dimensions, with dense index-manipulation:

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

This miasma of indexing expressions is hard to read and prone to simple textual input mistakes, *e.g.* swapping `-` and `+`, missing an indexing term, transforming the wrong variable, *e.g.* `u(i+1,j)` instead of `u(i,j+1)`, or reading from the wrong array, *e.g.* `u(i,j)` instead of `v(i,j)`.

In practice, the typical development procedure for complex stencil computations involves some *ad hoc* testing to ensure that no mistakes have been made *e.g.*, by visual inspections on data, or comparison against manufactured or analytical solutions [Farrell et al. 2010]. Such testing is often discarded once the code is shown correct. This is not the only information that is discarded. The shape of the indexing pattern is usually the result of choices made in the numerical-analysis procedure used to discretise some continuous equations. Rarely are these decisions captured in the source code, yet the derived shape is usually uniform with a clear and concise description *e.g.*, *centered space, of depth 1* referring to indexing terms `a(i)`, `a(i-1)` and `a(i+1)` [Recktenwald 2004].

To support correct array programming, we propose a simple, abstract specification language for the data access pattern of array-loop computations. This provides a way to prevent indexing errors and also to capture some of the original high-level intent of the algorithm. The language design is informed by an initial empirical study of array computations in a corpus of a real-world scientific code bases, totalling 1.4 million physical lines of code[1] (Section 2).

[1]Though due to strictness of the parser, we analysed 1.1 million physical lines of code from this corpus. We use Wheeler's *SLOCCount* to get a count of physical source lines, excluding comments and blank lines [Wheeler 2001].

We confirm our initial hypotheses of the ubiquity of looped array computations, but also that they have a common form, reading from arrays in a fixed neighbourhood of contiguous elements with simple static patterns. From this we designed a simple set of specification combinators to abstractly describe common array patterns (Section 3). As an example, the one-dimensional Laplace program is specified in our language by:

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

That is, an array named `a` is accessed with a symmetrical pattern in its first dimension ("centered") to a depth of one in each direction relative to some index (let's call it i) and its result contributes to an array write at index i; it is a stencil. The `readOnce` keyword means that each array subscript occurs exactly once. The Navier-Stokes example has two specifications:

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

These specifications record that `u` is accessed with a centered pattern to depth of 1 in both dimensions (this is known as the *five-point stencil*) and `v` is accessed in a neighbourhood bounded forwards to depth of 1 in the first dimension and backward to a depth of 1 in the second dimension. The specification is relatively small and abstract compared with the code, with a small number of combinators that *do not involve any indexing expressions*, *e.g.* `a(i+1,j-1)`. This contrasts with other specification approaches, *e.g.*, deductive verification tools such as ACSL [Baudin et al. 2008], where specifications about array computations must also be expressed in terms of array-indexing operations. Thus, any low-level mistakes that could be made whilst programming complex indexing code could also be made when writing its specification. Our specifications are abstract to avoid the error-prone nature of index manipulation and are lightweight to aid their adoption by scientists.

We implemented a verification tool for our specification language as an extension to CamFort, an open-source analysis tool for Fortran [Contrastin et al. 2016, 2017]. Specifications are associated with code via comments, against which the tool checks the code for conformance. This specify-and-check approach reduces testing effort and future-proofs against bugs introduced during refactoring and maintenance. A specification also concisely captures the array access pattern, providing documentation. We provide an inference procedure which efficiently generates specifications with no programmer intervention, automatically inserting specifications at appropriate places in the source code. This aids adoption of our approach to existing legacy code base.

The checking and inference algorithms (Section 6) work on a model of array access patterns in code (Section 5). The domain of this model is reused by a denotational model of our specification language (Section 5.2). Thus checking reduces to equality on models.

Using the inference procedure, we applied our tool to our original corpus (Section 7). Our tool identifies and infers specifications for 87,280 array computations in the corpus. Of those computations we found, 7,865 are non-trivial, corresponding to code that has a higher potential for errors. This validates the design of our language in its capability to capture many core patterns.

Our approach does not target a class of bugs that can be detected automatically (*push-button verification*). Instead, array indexing bugs must be identified relative to a specification of the intended access pattern. Nevertheless, we studied the commit logs of one package in our corpus and report on instances of code revisions where a correct specification would have spotted programming errors (which were later corrected by a code change) or would have assisted in refactoring array code (Section 7.3).

Terminology and notation. We use the following terminology for syntactic constructs in our target language.

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

Definition 2 (Array subscripts and indices)**.** An *array subscript*, denoted $a(\bar{e})$, is an expression which indicates the element of an N-dimensional array a at the N-dimensional *index* $\bar{e}$, specified by a comma-separated sequence of integer expressions $e_1, \ldots, e_N$. We also refer to each one of these e_i as an index. An index e_i is called *relative* if the expression involves an induction variable. An *absolute index* is an integer expression which is constant with respect to the enclosing loop, *i.e.*, it does not involve an induction variable.

Definition 3 (Origin)**.** For an array subscript $a(\bar{e})$, the index $\bar{e}$ is called an *origin index*, if all $e \in \bar{e}$ are naked induction variable expressions, *e.g.*, `a(i, j)`. We refer to this as the "origin" since the indices are offset by 0 in each dimension relative to the induction variables.

Definition 4 (Neighbourhood and affine index)**.** For an array subscript $a(\bar{e})$, an index $e \in \bar{e}$ is a *neighbourhood index* (or *neighbour*) if e is of the form $e \equiv$ `i`, $e \equiv$ `i + c`, or $e \equiv$ `i - c`, where `c` is an integer constant and `i` is an induction variable. That is, a neighbourhood index is a constant offset/translation of an induction variable. (The relation $\equiv$ here identifies terms up-to commutativity of + and the inverse relationship of + and − *e.g.*, `(-b) + i` $\equiv$ `i - b`).

An *affine index* is an index expression of the form `a*i + b`, where `a` and `b` are constants.

2 EMPIRICAL STUDY OF ARRAY COMPUTATIONS IN SCIENTIFIC FORTRAN CODE

We start with the following hypotheses about array programming idioms in scientific code, formed from our observations and conversations with scientists:

(1) Computations involving loops over arrays are common, where arrays are read according to a static pattern of neighbourhood or affine indexing expressions.
(2) Most looped-array computations of the previous form read from arrays with a *contiguous* pattern, *e.g.*:

```
1    x = a(i-1) + a(i) + a(i+1)
```

(3) Most loop-array computations of the previous form read from arrays with a pattern that includes the origin index or its immediate neighbours (offsets of 1 from the induction variables);
(4) Many array computations are *stencil computations*: an assignment whose right-hand side comprises array subscripts with neighbourhood indices and whose left-hand side is a neighbourhood-indexed array subscript (*e.g.*, the Laplace operator);
(5) Many array computations read from each particular array subscript just once per loop iteration.

The significance of the last hypothesis is that, if true, it would be useful to provide a specification of subscript uniqueness, so accidental lexical repetition of subscripts within a loop in complex code can be detected.

From these hypotheses, we conjecture that the spatial properties of the above programming idioms can be specified declaratively via a small set of combinators, capturing data access patterns. We performed a large scale source-code analysis to validate these hypotheses and guide the design of our language. These results are potentially of interest to others, such as designers of array/stencil DSLs, libraries, autotuners, or future verification tools.

2.1 Methodology

We constructed a corpus of eleven scientific computing packages written in Fortran ranging in size and scope:

Table 1. Summary of software packages used for evaluation

	Number of files		Physical lines of code		Lines of raw code	
Package	Total	Parsed	Total	Parsed	Total	Parsed
UM	2,540	2,272	635,525	543,000	1,010,936	868,761
E3ME	167	155	44,935	40,347	73,545	63,185
Hybrid4	29	29	4,831	4,831	8,361	8,361
GEOS-Chem	604	340	449,222	275,389	856,463	420,850
Navier	6	6	505	505	696	696
CP	52	48	2,334	2,121	3,978	3,632
BLAS	151	149	16,046	15,993	40,882	40,679
ARPACK-NG	312	290	50,208	47,453	144,081	139,290
SPECFEM3D	555	477	137,468	103,381	232,356	178,813
MUDPACK	88	88	54,753	54,753	78,652	78,652
Cliffs	30	30	2,424	2,424	3,149	3,149
Total	4,534	3,884	1,398,251	1,090,197	2,453,099	1,806,068

(1) The Unified Model (UM) developed by the UK Met Office for weather modelling [Wilson and Ballard 1999] (version 10.4);
(2) E3ME, a mixed economic/energy impact predication model [Barker et al. 2006];
(3) Hybrid4, a global scale ecosystem model [Friend and White 2000];
(4) GEOS-Chem, troposhperic chemistry model [Bey et al. 2001];
(5) Navier, a small-size Navier-Stokes fluid model [Griebel et al. 1997];
(6) Computational Physics (CP), programs from a popular textbook, *Introduction to Computational Physics* [Pang 1999];
(7) BLAS, a common linear-algebra library used in modelling [Blackford et al. 2002];
(8) ARPACK-NG, an open-source library for solving eigenvalue problems [Sorenson et al. 2017];
(9) SPECFEM3D, global seismic wave models [Komatitsch et al. 2016];
(10) MUDPACK, a general multi-grid solver for elliptical partial differentials [Adams 1991];
(11) Cliffs, a tsunami model [Tolkova 2014].

These cover approximately 1.4 million physical lines of Fortran code (2.4 million including comments and white space). The UM and GEOS-Chem packages are the largest at ≈635kloc and ≈450kloc respectively. Table 1 provides further detail on these packages and their sizes. We used Wheeler's *SLOCCount* to get a count of the physical source lines (excluding comments and blank lines) [Wheeler 2001]. A third of the packages come from active research teams who are our project partners (*i.e.*, they were not selected carefully to fit our hypotheses).

Analysis tool. We built a code analysis tool based on CamFort, an open-source Fortran analyser [Contrastin et al. 2017]. Fortran source files are parsed to an AST and standard control-flow and data-flow analyses are computed, including some of key importance for us: induction variable identification and reaching definitions. The resulting AST is traversed top-down and assignment statements inside loops are analysed and classified. CamFort implements standards compliant parsers, and so we were not able to parse all of the corpus as some packages contain non-standard code. Of the 1.4 million lines, just under 1.1 million physical lines of code were parsed.

Our analysis classifies assignment statements as array computations if they are contained within a loop and their right-hand side has relative array subscripts, *i.e.* their indices use induction variables. Assignments are classified based on all the expressions that may[2] flow to their right-hand side. For

[2]We use a "may" analysis, in the sense that it takes the union of information from merging dataflow paths.

example, the following code has an array computation with a one-dimensional three-point pattern split across multiple intermediate assignments:

```
1 do i = 1, n
2   x = a(i)
3   y = a(i+1)
4   b(i) = (a(i-1) + x + y)/3.0
5 end do
```

Our analysis recognises this as a single array computation (rather than three), starting at line 4, and reading from subscripts `a(i)`, `a(i+1)`, `a(i-1)`. We traverse loop bodies bottom-up, using reaching-definitions to calculate the array subscripts that may flow to an assignment. Since `x` and `y` flow to line 4, the intermediate statements on line 2 and line 3 are not classified separately, though several such intermediate assignments can flow to many classified statements.

Classifications. Left-hand sides and right-hand sides of assignments are classified based on their array subscripting expressions:

Classification (of subscripts)	**Applies** to	**Example**
Just a variable	LHS	`x = . . .`
Induction variables (IVs)	LHS	`a(i, j)`
Neighbourhood offsets (of the form $i \pm c$)	LHS/RHS	`a(i, j-1)`
Neighbourhood offsets and absolutes	LHS/RHS	`b(i, 0, j+1)`
Affine offsets (of the form $a * i \pm b$)	LHS/RHS	`a(2*i+1,j)`
Affine offsets and absolutes	LHS/RHS	`a(i+1, 0, 3*j+2)`
Subscript expression not included above	LHS/RHS	`x(f(i))`, `a(i*i)`, `a(0,1)`

All classifications are disjoint. For example, neighbourhood patterns, despite being affine, are not included in categories labelled as affine. Similarly, categories with absolute variants are separate rather than subcategories. Note however, that subscripts classified as affine can have a mixture of neighbourhood and affine indices, *i.e.*, they must have at least one affine indexing expression.

We further sub-classify sets of array subscripts on the right-hand side of an assignment based on their spatial properties:

Property	**Shorthand**	**Classifications (of RHS)**	**Example**
Contiguity	contig	Contiguous	`a(i) + a(i+1) + a(i+2)`
	disjoint	Non-contiguous	`a(i) + a(i+2)`
Reuse	readonce	Unique subscripts	`a(i) + a(i+1)`
	mult	Repeated subscripts	`a(i) + a(i)`
Positioning	origin	Includes origin	`a(i,j)`
	straddle	At distance 1 of origin	`a(i+1,j)`, `a(i-1,j+1)`
	away	Away from the origin	`a(i+2)`, `a(i+3)`

Finally, we classify the relationship between the LHS and RHS in terms of the use of induction variables. The two sides of an assignment are *directly consistent* if the same induction variables appear in each side, used for the same dimensions. This is weakened to a *permutation* if the roles of the induction variables changes. This is weakened further if the LHS induction variables are either a subset or superset of the induction variables on the RHS. Otherwise, the two sides are seen as inconsistent:

Classification	Sub Classification	Example
Consistent	Direct	`a(i, j) = b(i, j) + b(i+1,j+1)`
	Permutation	`a(i, j) = c(j, i)` or `a(i, 0) = b(0, i)`
	LHS subset	`a(i) = b(i, j) + b(i, j-1)`
	LHS superset	`a(i, j) = b(i)`
Inconsistent		`a(i) = b(j)`

2.2 Results

We revisit each hypothesis and show the related data and conclusions.

(1) Computations involving loops over arrays are common, where arrays are read according to a static pattern of neighbourhood or affine indexing expressions.

We identified 133,577 instances of assignments within loops in which an array subscript flows to the right-hand side. We refer to each one of these as an *array computation*. We performed the classification described above and grouped the data by the right-hand side classification only:

RHS classification	Number	%	*Grouping 1*
Affine	37	0.03%	
Neighbourhood	71,971	53.88%	76.94%
Neighbourhood + absolute	30,764	23.03%	
Other	30,805	23.06%	
Total	133,577	100.00%	

Since affine and neighbour patterns are static, we group them (*Grouping 1*) to see that 76.94% of the array computations (102,772 instances) have a static pattern comprising neighbourhood or affine subscript expressions (potentially with some absolute indices), confirming *Hypothesis 1*. Note that affine subscripts are rare, perhaps due to programmers relying on compilers for optimisation rather than hand optimising (which tends to lead to affine indices). Overwhelmingly, the main category is neighbourhood offsets (possibly mixed with absolute indices in some dimensions). The "affine + absolute" class is not represented at all.

(2) Most loop-array computations of the previous form read from arrays with a *contiguous* pattern.
(3) Most loop-array computations of the previous form read from arrays with a pattern that includes the immediate neighbours (offsets of 1 from the induction variables);

We grouped the data based on subclassifications of *Grouping 1* in terms of contiguity and positioning. From this point on we combine the classifications of "neighbourhood" and "neighbourhood with absolute", and refer to the class as "neighbour" (or "neigh" for short).

RHS classification	Number	%(of whole)	Position	Number	%(of whole)
Affine, contiguous	34	0.03%			
Affine, non-contiguous	3	0.00%			
Neighbour, contiguous	96,502	72.24%	away	166	0.12%
			origin	93,733	70.17%
			straddle	2,603	1.95%
Neighbour, non-contiguous	6,233	4.67%	away	2	0.00%
			origin	6,215	4.65%
			straddle	16	0.01%
Other	30,805	23.06%			
Total	133,577	100.00%			

Thus, 72.24% of array computations have only neighbour indices (which may include some absolute indices) in a contiguous pattern on the right hand side, with 72.12% either including the origin

(70.17%) or offset by a distance of 1 from the origin (1.95%). This confirms both Hypothesis 2 and Hypothesis 3. We found only 168 instances of neighbour index patterns (contiguous or not) that are not immediately next-to or over the origin. There is a non trivial amount of non-contiguous arrays access (4.67% of all array computations), but it is a much smaller proportion of the whole, and mostly contains the origin.

(4) Many array computations are *stencil computations*: an assignment whose right-hand side comprises array subscripts with neighbourhood indices and whose left-hand side is a neighbourhood array subscript.

We classified array computations by their RHS and LHS subscript classifications considered together, We additionally included the orthogonal classification of whether left- and right-hand sides are inconsistent or otherwise (have a common subset of induction variables).

LHS and RHS classification		**# inconsistent**	%(whole)	**# consistent**	%(whole)	*stencils*
LHS Vars	RHS neigh	0	0.00%	8,386	6.28%	
LHS Vars	RHS affines	0	0.00%	5	0.00%	
LHS IVs	RHS neigh	5,757	4.31%	54,626	40.89%	
LHS neigh	RHS neigh	2,422	1.81%	20,030	15.00%	55.86%
LHS IVs	RHS affines	6	0.00%	0	0.00%	
LHS affine	RHS affines	0	0.00%	6	0.00%	
LHS neigh	RHS affines	8	0.01%	0	0.00%	
LHS affine	RHS neigh	23	0.02%	0	0.00%	
other		42,308	31.67%			
Total		50,524	37.82%	83,053	62.17%	

The most common category is an LHS array subscript indexed by induction variables, and an RHS comprising neighbourhood offsets. Next most common is to have neighbourhood offsets on the left-hand and right-hand sides. Notably there are two categories always seen as inconsistent: "LHS affine, RHS neighbour" and "LHS neighbour, RHS affine" since they comprise different indexing schemes on each side of the assignment, e.g. $a(i+1) = b(2*i+1)$. These appear very rarely.

We found that 62.17% of array computations have a static pattern on the LHS and RHS that is either neighbourhood or affine and is consistent, and that 55.86% are *stencils*, writing to an array subscript on the left. Thus, the data supports our hypothesis that stencils are common.

For computations with consistent LHS and RHS, we considered two subclassifications in terms of contiguity (left table) and the different kinds of left-right-hand side relationship (right table):

Contiguity	#	%(whole)
Contiguous	79,024	59.16%
Non-contiguous	4,029	3.02%
Total	83,053	62.17%

Consistency	#	%(whole)	**#relativised**	%(whole)
Direct	55,443	41.51%	854	0.64%
LHS subset	10,461	7.83%	72	0.05%
LHS superset	12,005	8.99%	91	0.07%
Permutation	4,080	3.05%	47	0.04%
Total	81,989	61.38%	1,064	0.08%

The additional column in the right table (#relativised) counts the number of specifications where the left-hand side has neighbourhood offsets, *e.g.*, `a(i+1) = b(i)`. Most consistent array computations are contiguous, but there is some non-contiguity. Most consistent array computations have matching induction variables on the left and right-hand sides (directed consistency) but there are non-trivial amounts of the other kinds of relationship.

(5) Many array computations read from each particular subscript just once per loop iteration.

We classified right-hand sides comprising neighbourhood (or affine) subscripts and subclassified this by whether array subscript terms were all unique or whether there were some repeats.

RHS form and reuse	Number	%(whole)
RHS neighbour/affine unique subscripts	89,678	67.14%
RHS neighbour/affine repeated subscripts	13,094	9.80%
other	30,805	23.06%

Thus most array computations have access patterns with unique use of array subscripts, confirming Hypothesis 5.

2.3 Additional Properties of Array Computations

For each of the 133,577 array computations, we recorded three additional properties: the dimensionality of array subscripts, the number of subscript terms in the computation, and the length of the dataflow path (*i.e.*, how many intermediate assignments contribute to the array computation):

Dimensionality	1	2	3	4	5	6	7	8	9	10
Count	55,832	40,059	27,909	9,081	575	78	15	31	0	3

# subscripts:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16≤
Count	102,846	17,601	5,744	3,087	1,280	709	235	564	296	194	43	81	35	17	47	798

Dataflow length:	0	1	2	3	4	5	6	7	8	9	10	11	12	13≤
Count	70,251	7,814	9,445	8,186	7,730	6,073	5,244	3,887	3,162	2,422	2,075	1,628	1,496	13,257

The data for "number of subscripts" shows that most computations are relatively simple, with only 1-4 array subscripts in 97% of array computations. However, some are complex, with roughly 800 array computations using over 16 indexing terms. We had one example with 96 array subscripts. Thus, we see that there are some cases in practice with complex patterns. We aim to provide support for such code with our specification language since these are also the most error prone. The rest of the data is useful when considering the performance of our checking procedure (Section 6.2).

2.4 From Experiment to Language Design

We used these results to inform the design of our specification language. The objective was to capture the salient aspects of the programming patterns identified above whilst remaining independent of their implementation (and associated programming errors). We made the following design decisions:

- The core support should be for neighbourhood offsets on right-hand side of assignments in loops (76.94% of our array computations);
- Access patterns are mostly contiguous (72.24% of all array computations) and cross or straddle the origin (72.12%). We thus represent patterns by *finite intervals* that meet at, or are next to, the origin. These intervals will be given declaratively and abstractly, avoiding replicating indexing terms at the specification level.
- We will support only consistent left and right-hand sides (where there are common induction variables on each side) (62.17% of all array computations).
- Stencils will be the primary focus (55.86%) but we will also allow specifications on assignments where the LHS is a variable (*e.g.*, in a reduction) (6.28%).
- There is some non-contiguity for stencils (3.02% of total array computations) which we will specify by finite intervals given as approximations that cover a non-contiguous pattern.
- Affine indexing is very rare 0.03% so we do not consider it.

$$
\begin{aligned}
\textit{specification} &::= \textit{regionDec} \mid \textit{specDec} \\
\textit{specDec} &::= \texttt{stencil}\ \textit{spec}\ \texttt{::}\ v\ [\texttt{,}v]\ldots \mid \texttt{access}\ \textit{spec}\ \texttt{::}\ v\ [\texttt{,}v]\ldots \\
\textit{regionDec} &::= \texttt{region}\ \texttt{::}\ \textit{rvar}\ \texttt{=}\ \textit{region} \\
\textit{spec} &::= [\textit{mult},]\ [\textit{approx},]\ \textit{region} \\
\textit{mult} &::= \texttt{readOnce} \\
\textit{approx} &::= \texttt{atMost} \mid \texttt{atLeast} \\
\textit{region} &::= \textit{rvar} \mid \textit{rconst} \mid \textit{region} + \textit{region} \mid \textit{region} * \textit{region} \\
\textit{rconst} &::= \texttt{pointed(dim=}\mathbb{N}_{>0}\texttt{)} \\
&\quad \mid \texttt{forward(dim=}\mathbb{N}_{>0}\texttt{,depth=}\mathbb{N}_{>0}\texttt{, [, nonpointed])} \\
&\quad \mid \texttt{backward(dim=}\mathbb{N}_{>0}\texttt{,depth=}\mathbb{N}_{>0}\texttt{, [, nonpointed])} \\
&\quad \mid \texttt{centered(dim=}\mathbb{N}_{>0}\texttt{,depth=}\mathbb{N}_{>0}\texttt{, [, nonpointed])} \\
\textit{rvar} &::= \texttt{[a-z A-Z 0-9]+}
\end{aligned}
$$

Fig. 1. Specification syntax (EBNF grammar)

3 A SPECIFICATION LANGUAGE FOR THE SHAPE OF ARRAY PATTERNS

Figure 1 gives the syntax of specifications, which is detailed below. The entry point is the *specification* production which splits into either a *region declaration* (*regionDec*) or a *specification declaration* (*specDec*). Specifications are associated to one or more Fortran array variables (ranged over by v). Regions comprise *region constants* which are combined via region operators + and *.

Region constants (non-terminal *rconst*) specify a finite interval in a single dimension starting at the origin and are either `pointed`, `forward`, `backward`, or `centered`. The region names are inspired by numerical analysis terminology, *e.g.* the standard explicit method for approximating PDEs is known as the *Forward Time, Centered Space* (FTCS) scheme [Dawson et al. 1991]. Each region constant has a dimension identifier d (written `dim=`d) given as a positive natural number. Each constant except `pointed` has a depth parameter n (written `depth=`n) given as a positive natural number; `pointed` regions implicitly have a depth of 0.

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

```
1  != stencil forward(depth=2, dim=1) :: a
2  e(i, 0) = a(i, 0) + a(i+1, 0) + a(i+2, 0)
3
4  != stencil backward(depth=2, dim=1) :: b
5  f(i) = b(i) + b(i-1) + b(i-2)
6
7  != stencil centered(depth=1, dim=1) :: c
8  e(i, j) = (c(j-1) + c(j) + c(j+1))/3.0
9
10 != stencil pointed(dim=3) :: d
11 e(i, j) = d(0, 0, i)
```

If subscripts in an array computation contain only absolute index terms in one particular dimension, then that dimension is left unspecified. For example, in the computation on line 2, the second dimension is always absolute and so is not included in the specification on line 1.

The `forward`, `backward`, and `centered` regions may all have an additional attribute `nonpointed` which marks absence of the origin, corresponding to those array computations classified as straddle in the previous section. For example, the following is a `nonpointed` `backward` stencil:

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

Combining regions. The region operators + and * respectively combine regions by union and intersection. The intersection of two regions $r * s$ means that any indices in the specified code must be consistent with both r and s simultaneously. Dually, the union of two regions $r + s$ means that indices in the specified code must be consistent with one of r or s, or both. For example, the following *nine-point stencil* has a specification given by the product of two `centered` regions in each dimension:

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

The specification ranges over the values that flow to the array subscript on the left-hand side, and so ranges over the assignments to `x`, `y`, and `z`. Each index in the code is consistent with both parts of the specification, *e.g.*, `a(i-1, j+1)` is within the centered region in dimension 1 and the centered region in dimension 2.

The union of two regions $r + s$ means that any indices in the specified code must be consistent with either of r or s. For example, the following gives the specification of a five-point stencil which is the sum of two compound `pointed` and `centered` regions in each dimension:

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

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

Region declarations and variables. Region specifications can be assigned to region variables (*rvar*) via region declarations. For example, the shape of a "*Roberts cross*" edge-detection convolution [Davis 1975] can be stated:

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

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

Modifiers. Region specifications can be modified by *approximation* and *multiplicity* information (in *spec* in Figure 1). The `readOnce` modifier enforces that no subscript appears more than once in the concrete syntax (that is, its multiplicity is one). For example, all of the previous examples should have the `readOnce` modifier (though it was elided previously for clarity):

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

This specification would be invalid if any of the array subscripts were repeated lexically. This modifier provides a way to rule out any accidental repetition of array subscripts. The notion is inspired by linear types [Wadler 1990], where a value must be used exactly once. However, in this

case we look for syntactic occurrences rather than semantic occurrences, *e.g.* `x = a(i); y = x + x` is `readOnce` whereas `y = a(i) + a(i)` is not. We chose the informative and easily understood name `readOnce`, meaning that within the context of this loop, the particular array index must be read only once, whereas without `readOnce` the corresponding array index must be read at least once.

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

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

Relativisation. The data access pattern of a stencil is understood relative to the subscript on the left-hand side. If the left-hand side of a stencil has a neighbourhood subscript with non-zero offsets, then the right-hand side is understood relative to this shifted origin. For example:

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

Note that the specification for dimension 1 is to a depth of 2, since the left-hand side is shifted forward by 1.

Access specifications. Specifications can also be given to array computations which are not stencils (*i.e.*, do not have a left-hand side which is an array subscript). For example, for a reduction:

```
do i = 1, n
  != access pointed(dim=1) :: a
  r = max(a(i), r)
end do
```

The main difference between `stencil` and `access` specifications is that there is no left-hand side with which to check the consistency of induction variable use or to relativise against.

4 EQUATIONAL & APPROXIMATION THEORIES

Our region specifications are subject to an equational theory $\equiv$, which explains which region specifications are equivalent, and a mutually-defined approximation theory $\preceq$ for over- and under-approximation on regions.

4.1 Equivalences

We define an equivalence relation, $\equiv$. The purpose of this relation is to allow programmers to write specifications with greater flexibility. It allows specifications to be written in various levels of compactness allowing for space optimisation or greater clarity of documentation. The relation is defined on regions as follows:

Basic $\star$ and $+$ are both idempotent, commutative, and associative;

Subsumption If S and R are regions with $S \preceq R$, then $S{+}R \equiv R$ and $S{\star}R \equiv S$.

Distribution $\star$ distributes over $+$ and dually $+$ distributes over $\star$, meaning if $\mathbf{R}$, $\mathbf{S}$, and $\mathbf{T}$ are regions, then we have the following dual equivalences:

$$\mathbf{R}{\star}(\mathbf{S}{+}\mathbf{T}) \equiv (\mathbf{R}{\star}\mathbf{S}){+}(\mathbf{R}{\star}\mathbf{T}) \qquad\qquad \mathbf{R}{+}(\mathbf{S}{\star}\mathbf{T}) \equiv (\mathbf{R}{+}\mathbf{S}){\star}(\mathbf{R}{+}\mathbf{T})$$

Overlapping pointed If $\mathbf{R}$ is one of `forward`, `backward`, or `centered`, then we have the following:

$$\mathbf{R}(\texttt{dim=}n,\texttt{depth=}k,\texttt{nonpointed}) + \texttt{pointed(dim=}n\texttt{)} \equiv \mathbf{R}(\texttt{dim=}n,\texttt{depth=}k)$$

Centered The region constants `forward` and `backward` are two halves of `centered` specifications:

$$\texttt{centered(dim=}n\texttt{,depth=}k\texttt{,}\mathbf{p}\texttt{)} \equiv \texttt{forward(dim=}n\texttt{,depth=}k\texttt{,}\mathbf{p1}\texttt{)+backward(dim=}n\texttt{,depth=}k\texttt{,}\mathbf{p2}\texttt{)}$$

Here **p** is nonpointed if both **p1** and **p2** are nonpointed. Otherwise **p** is "pointed", meaning that the region is centered(dim=n,depth=k).

4.2 Approximations

We define a partial order of approximations, $\preceq$. This relation is used in the equational theory and provides a means of writing more compact lower and upper bound specifications. The relation is defined as follows:

Equivalence If S and R are regions and $S \equiv R$, then we have $S \preceq R$ and $R \preceq S$.
Combined If S and R are regions, then we have $S \preceq S{+}R$ and $S{*}R \preceq S$.
Depth Let k and l be in positive integers and $k \leq l$, n some fixed dimension, and **p** either pointed or nonpointed. Further, let **R** be one of centered, forward, and backward. We then have

$$\mathbf{R}(\text{dim}{=}n,\text{depth}{=}k,\mathbf{p}) \preceq \mathbf{R}(\text{dim}{=}n,\text{depth}{=}l,\mathbf{p})$$

We present some derivable inequalities that are useful when writing specifications:

Proposition 1 (Centered approximation). *For any dimension n, depth k, and pointed attribute* **p***:*

$$\begin{aligned}\text{forward}(\text{dim}{=}n,\text{depth}{=}k,\mathbf{p}) &\preceq \text{centered}(\text{dim}{=}n,\text{depth}{=}k,\mathbf{p})\\ \text{backward}(\text{dim}{=}n,\text{depth}{=}k,\mathbf{p}) &\preceq \text{centered}(\text{dim}{=}n,\text{depth}{=}k,\mathbf{p})\end{aligned}$$

Proposition 2 (Point approximation). *Let* **R** *be one of* forward, backward, *and* centered, *n a fixed dimension, and k a fixed depth, then we have*

$$\begin{aligned}\text{pointed}(\text{dim}{=}n) &\preceq \mathbf{R}(\text{dim}{=}n,\text{depth}{=}k)\\ \mathbf{R}(\text{dim}{=}n,\text{depth}{=}k,\text{nonpointed}) &\preceq \mathbf{R}(\text{dim}{=}n,\text{depth}{=}k)\end{aligned}$$

If an array computation conforms to a region R and there is a region S such that $R \preceq S$, then it conforms also to the upper bound specification atMost S.

5 SEMANTIC MODEL

We define a lattice model of array access patterns which serves as a semantic model of our specification language and a model of array access patterns in source code. This model is (1) used to explain the meaning of our specifications; (2) provides a basis for the inference and checking algorithms in Section 6; (3) justifies the equational theory for specifications; (4) is used to optimise specifications using lattice identities; and (5) can be used to guide correct implementations.

The model is defined over sets of integer vectors. As an initial informal example, consider the stencil kernel `y(i) = x(i, c) + x(i+1, c)` where `c` is a constant. The access pattern on array `x`, relative to induction variable `i`, is captured in our model by the Cartesian product of two integer sets: $\{0, 1\} \times \mathbb{Z}$. This describes that, in the first dimension, the array is read at offsets of 0 and 1 from an induction variable (`i` here). In the second dimension, the index is unconstrained as it is absolute (constant `c`). The intuition for mapping the absolute index `c` to $\mathbb{Z}$ is that it is at an *arbitrary* distance from the induction variable `i`.

Our model of specifications forms a lattice which provides a rich set of equations and properties, which we exploit. We first set up the domain of the model (§5.1), using it to define a semantics for our specification language (§5.2) and then as the target for an interpretation of the syntax of array subscripts in code (§5.3); thus we define two models: one for specifications and one for array terms in source code. We state various results in this section, for which the proofs are provided in the accompanying technical report [Orchard et al. 2017].

5.1 Lattice Model of Regions

The underlying domain of our semantics is the set $\mathcal{P}(\mathbb{Z}^N)$ that is, sets of integer vectors of length N. We characterise various subsets of this space used in our model, called: *index schemes*, *interval schemes*, *regions*, and *subscript schemes*.

Definition 5 (Index scheme)**.** An N-dimensional *index scheme* S is a set of integer vectors ($S \in \mathcal{P}(\mathbb{Z}^N)$) which can be written as the Cartesian product of N subsets of $\mathbb{Z}$, *i.e.*

$$\exists S_1 \subseteq \mathbb{Z}, \ldots, S_N \subseteq \mathbb{Z} \,.\; S = \prod_{i=1}^{N} S_i$$

with the additional condition that every S_i is either finite (*i.e.*, $\exists n.|S_i| = n$) or $S_i = \mathbb{Z}$.

We use S, T, U to denote index schemes throughout. Henceforth, we implicitly assume index schemes are N-dimensional[3] for some N when it is clear from the context.

Index schemes can be *projected* in the i^{th} dimension by $\pi_i : \mathcal{P}(\mathbb{Z}^N) \to \mathcal{P}(\mathbb{Z})$. For an index scheme S, we refer to $\pi_i(S)$ as the i^{th} *component* of S. We assume that i, when used for projection, always lies between 1 and N.

Lemma 1. *Intersection distributes over index schemes. That is, for index schemes S, T*

$$S \cap T = \prod_{i=1}^{N} \pi_i(S) \cap \pi_i(T)$$

Thus intersection is closed for index schemes: the intersection of two index schemes is an index scheme.

Union however is not closed over index schemes; the union of two index schemes is not itself an index scheme because union does not distribute over Cartesian product (recall, index schemes must be definable as a Cartesian product of integer sets). However, a more restricted property holds.

Lemma 2. *If S and T are index schemes where $\pi_i(S) = \pi_i(T)$ for all $1 \leq i \leq N$ apart from some dimension k, then:*

$$S \cup T = \pi_1(S) \times \cdots \times (\pi_k(S) \cup \pi_k(T)) \times \cdots \times \pi_N(S)$$

Definition 6 (Intervals with an optional hole)**.** We define an extended notion of closed interval on $\mathbb{Z}$ which may contain a *hole* at the origin, written $[a \ldots b]^c$ where a and b are drawn from $\mathbb{Z}$ with $a \leq 0 \leq b$ and c is drawn from $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. Intervals are interpreted as sets as follows:

$$[a \ldots b]^c \triangleq \{n \mid a \leq n \leq b \wedge (c \vee n \neq 0)\}$$

If the superscript to the interval is omitted it is treated as true (no hole). We also add the distinguished interval $[-\infty \ldots \infty]$, which is simply an alias for $\mathbb{Z}$, but this notation avoids separate handling of the infinite interval in the following definitions, lemmas, and their proofs.

We denote the set of all intervals with an optional hole as *Interval*.

Lemma 3. *Union and intersection are closed for Interval, where we have the following dual identities:*

$$[a \ldots b]^c \cap [d \ldots e]^f = [\max\{a, d\} \ldots \min\{b, e\}]^{c \wedge f}$$
$$[a \ldots b]^c \cup [d \ldots e]^f = [\min\{a, d\} \ldots \max\{b, e\}]^{c \vee f}$$

We define a subset of index schemes called *interval schemes*, where each component of an index scheme is a member of *Interval*.

[3]Whenever the model is used, the particular dimensionality N is derived from source code, from the static type of the array being analysed.

Definition 7 (Interval scheme)**.** An *interval scheme* is an N-dimensional index scheme such that each component of the scheme is a holed interval (Definition 6) (since intervals define either finite subsets of $\mathbb{Z}$ or $\mathbb{Z}$ in its entirety).

$Interval_N$ denotes the set of all N-dimensional interval schemes.

Interval schemes are composed by union and intersection to form a *region*, which describes the extent of a specification in our language.

Definition 8 (Region)**.** The set of all N-dimensional regions, denoted $Region_N$ is defined as the smallest set satisfying the following:

(1) If R is in $Interval_N$, then R is in $Region_N$.
(2) If R and S are in $Region_N$, then so are $R \cap S$ and $R \cup S$.

Proposition 3. *$(Region_N, \cup, \cap, \top, \bot)$ is a bounded distributive lattice with top element $\top = \mathbb{Z}^N$ and bottom element $\bot = \emptyset$.*

The set of regions $Region_N$ is the target of our specification language model. However, the model must also incorporate information about approximation and multiplicity specification modifiers. This information is provided by the following labelled variants.

Definition 9. Mult and Approx are parametric labelled variant types with injections given by:

$$\mathsf{Mult}\ \ a \triangleq \mathsf{mult}\ a \mid \mathsf{only}\ a \qquad \mathsf{Approx}\ \ a \triangleq \mathsf{exact}\ a \mid \mathsf{lower}\ a \mid \mathsf{upper}\ a$$

e.g., lower is an injection into Approx, of type $\mathsf{lower} : a \to \mathsf{Approx}\ a$.

These injection functions are used to mark the model of regions ($Region_N$) with *multiplicity* and *approximation* information.

During specification and consistency checking (Section 6.2), Mult is used to determine whether repeated indices should be allowed, while Approx is used to determine if the region in the specification should be treated as a lower bound, an upper bound, or in exact correspondence with the region defined by the stencil computation.
Finally, we define another subset of index schemes called *subscript schemes* where each component of an index scheme is either a singleton set or $\mathbb{Z}$. This is used to model array index terms from source code in Section 5.3.

Definition 10 (Subscript scheme)**.** A *subscript scheme* S is an index scheme where:

$$\forall i.\ 1 \le i \le N \implies \exists p.\ \pi_i(S) = \{p\} \ \vee\ \pi_i(S) = \mathbb{Z}$$

That is, the i^{th} component of the set is either a singleton in $\mathbb{Z}$ or all of $\mathbb{Z}$.

5.2 Denotational Semantics for Specifications

This section defines the model of top-level specifications (non-terminal *spec* in Figure 1) as members of $Region_N$ augmented with modifier information.

Definition 11 (Semantics of specifications)**.** An interpretation function $[\![-]\!]_N$ maps closed[4] specifications to an element of $\mathsf{Mult}(\mathsf{Approx}(Region_N))$ as follows:

$$[\![\texttt{mult, approx, region}]\!]_N = [\![\texttt{mult}]\!]^{\mathrm{m}}\ ([\![\texttt{approx}]\!]^{\mathrm{a}}\ [\![\texttt{region}]\!]_N)$$

[4]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*.

$$
\begin{array}{ll}
[\![-]\!]^{\mathsf{a}} : \mathit{approx} \to (A \to \mathsf{Approx}\, A) & \\
[\![\texttt{atLeast}]\!]^{\mathsf{a}} & = \mathsf{lower} \\
[\![\texttt{atMost}]\!]^{\mathsf{a}} & = \mathsf{upper} \\
[\![\epsilon]\!]^{\mathsf{a}} & = \mathsf{exact}
\end{array}
$$

$$
\begin{array}{ll}
[\![-]\!]^{\mathsf{m}} : \mathit{mult} \to (A \to \mathsf{Mult}\, A) & \\
[\![\texttt{readOnce}]\!]^{\mathsf{m}} & = \mathsf{once} \\
[\![\epsilon]\!]^{\mathsf{m}} & = \mathsf{mult} \\
 & \\
[\![\texttt{nonpointed}]\!] & = \mathsf{false} \\
[\![\epsilon]\!] & = \mathsf{true}
\end{array}
$$

(a) Interpretation of modifiers

$$
\begin{array}{ll}
[\![-]\!]_N : \mathit{region} \to \mathit{Region}_N & \\
[\![\mathbf{r}\ \texttt{+}\ \mathbf{s}]\!]_N & = [\![\mathbf{r}]\!]_N \cup [\![\mathbf{s}]\!]_N \\
[\![\mathbf{r}\ \texttt{*}\ \mathbf{s}]\!]_N & = [\![\mathbf{r}]\!]_N \cap [\![\mathbf{s}]\!]_N \\
[\![\mathbf{rconst}]\!]_N & = \mathit{promote}_N(\mathit{dim}(\mathbf{rconst}), [\![\mathbf{rconst}]\!])
\end{array}
$$

$$
\begin{array}{ll}
[\![-]\!] : \mathit{rconst} \to \mathit{Interval} & \\
[\![\texttt{pointed(dim=}i\texttt{)}]\!] & = [0 \ldots 0]^{\mathsf{true}} \\
[\![\texttt{centered(dim=}i\texttt{,depth=}k\texttt{,}\mathbf{p}\texttt{)}]\!] & = [-k \ldots k]^{[\![\mathbf{p}]\!]} \\
[\![\texttt{forward(dim=}i\texttt{,depth=}k\texttt{,}\mathbf{p}\texttt{)}]\!] & = [0 \ldots k]^{[\![\mathbf{p}]\!]} \\
[\![\texttt{backward(dim=}i\texttt{,depth=}k\texttt{,}\mathbf{p}\texttt{)}]\!] & = [-k \ldots 0]^{[\![\mathbf{p}]\!]}
\end{array}
$$

(b) Interpretation of regions and region constants

Fig. 2. Semantic model of specifications

The interpretation is built from three intermediate interpretations of multiplicity modifiers **mult**, approximation modifiers **approx**, and regions **region**.

Multiplicity modifiers are interpreted as injections into the Mult variant by $[\![-]\!]^{\mathsf{m}}$, corresponding to the presence or absence of the `readOnce` modifier as shown in Figure 2a. Similarly, approximation modifiers are interpreted as injections into the Approx variant by $[\![-]\!]^{\mathsf{a}}$, defined in Figure 2a. The Approx type corresponds to the presence or absence of a spatial approximation modifier **approx**, with exact when there is no such modifier and lower and upper for `atLeast` and `atMost` respectively.

Lastly, the interpretation $[\![-]\!]_N$ is overloaded on region syntax, defined in Figure 2b. Two helper operations are used.

Definition 12. Let $\mathit{promote}_N : \mathbb{N}_{>0} \times \mathit{Interval} \to \mathit{Interval}_N$ be a function generating an interval scheme such that if v is $\mathit{promote}_N(i, [a \ldots b]^c)$, then $\pi_i(v) = [a \ldots b]^c$ and $\pi_j(v) = \mathbb{Z}$ in all other dimensions j.

Definition 13. Given a region constant **rconst**, let $\mathit{dim}(\mathbf{rconst}) \in \mathbb{N}_{>0}$ be the dimension for which that region constant is defined, *e.g.*, $\mathit{dim}(\texttt{centered(dim=}d\texttt{,depth=}k\texttt{)}) = d$.

The intermediate interpretation $[\![-]\!]$ defined second in Figure 2b models region constants. This is lifted by $\mathit{promote}_N$ to interpret region constants in the last equation of $[\![-]\!]_N$. Note that `nonpointed` is modelled as false, and its absence is modelled by true (bottom of Figure 2a). The + and * operators are modelled in terms of the join (union) and meet (intersection) of Region_N. Thus the syntax of regions is modelled by members of Region_N.

Theorem 1 (Equational soundness)**.** *The lattice model is sound with respect to the equational theory. Let R and S be N-dimensional region terms, then we have*

$$\forall R, S, N. \quad R \equiv S \implies [\![R]\!]_N = [\![S]\!]_N$$

Theorem 2 (Approximation soundness)**.** *The lattice model is sound with respect to the theory of approximation. Let R and S be N-dimensional regions, then we have*

$$\forall R, S, N. \quad R \preceq S \implies [\![R]\!]_N \subseteq [\![S]\!]_N$$

Note that the model is not complete with respect to equations or approximations since the specification language has no model of the bottom element of the lattice. An empty specification corresponds to the top point: $\mathbb{Z}^N$ meaning "fully unconstrained".

5.3 Modelling Array Subscripts

Definition 14 (Individual array terms)**.** Recall array subscript terms of the form $a(\bar{e})$ from Definition 2. We interpret these terms with a partial interpretation to subscript schemes (a subset of index schemes) $[\![-]\!]^{aterm} : \mathit{array\text{-}term} \nrightarrow \mathcal{P}(\mathbb{Z}^N)$. The interpretation is defined when all indices are either absolute or neighbourhood indices (Definition 4).

$$[\![a(\bar{e})]\!]^{aterm} = \prod_{1 \le i \le N} \mathit{subscript}(\bar{e}_i) \qquad \mathit{subscript}(e) = \begin{cases} \{c\} & e \equiv \mathtt{j} \pm c \\ \mathbb{Z} & e \text{ is absolute} \end{cases}$$

where `j` is an induction variable. Note that this interpretation function produces subscript schemes, but these are not necessarily members of $\mathit{Interval}_N$ or Region_N.

Definition 15 (Collections of array terms)**.** Given an set of $\mathcal{A}$ of array terms flowing to a particular statement in code, we take the union of the interpretations of array terms to give a model of the array access in the computation by:

$$[\![\mathcal{A}]\!]^{aterm} = \bigcup_{a \in \mathcal{A}} [\![a]\!]^{aterm}$$

Thus, $[\![\mathcal{A}]\!]^{aterm} \in \mathcal{P}(\mathbb{Z}^N)$. In Section 6, we use this interpretation on array subscripts in the static analysis of code prior to checking and inference.

5.4 Union Normal Form

We lastly address the concrete representations of the sets output from our models as used in our implementation for checking and inference (Section 6).

When a particular dimension is left unconstrained by a specification, the resulting model is infinite in size since an unconstrained dimension results in an index scheme with component $\mathbb{Z}$. Similarly, when an array term with an absolute index is modelled, *e.g.*, `a(i, 1)`, the resulting model is infinite since its derivation involves a subscript scheme with $\mathbb{Z}$ as a component. Thus, it is not always possible to simply enumerate all elements of our models as they may be infinite. However, a finite representation is possible, which we call *union normal form*. This is used in our consistency checking procedure and for presenting inferred specifications in a more readable manner.

Definition 16 (Union Normal Form)**.** A set $V \in \mathcal{P}(\mathbb{Z}^N)$ is in union normal form (UNF) if it can be expressed as the union of a finite number of N-dimensional index schemes; that is, there is a finite set of index schemes $\mathcal{S}$ such that $V = \bigcup_{T \in \mathcal{S}} T$. We refer to $\mathcal{S}$ as the *support* of V.

Lemma 4. *A set in UNF has a finite representation.*

It remains to show how both the models of specifications and the models of source code array terms can be written in UNF.

Lemma 5. *The model of a region specification produces a Region_N which can be converted into UNF.*

Furthermore, since Region_N is composed of interval schemes, for which intersection is closed (Lemma 3), the resulting model is a union of interval schemes — a compact representation. Multiple interval schemes combined with intersection are turned into a single interval scheme that is concretely represented by a triple of a lower bound, an upper bound, and a flag for its hole.

Lemma 6. *The model of array subscripts in code (Definition 14) already produces a set in union normal form: a UNF of subscript schemes (by Definition 15).*

Thus, both the models of specifications and array indices can be represented finitely in UNF. This representation acts as an intermediate form that allows straightforward constraint generation during consistency checking (whose constraints are discharged by an SMT solver, see Section 6.2).

Furthermore, UNF mitigates the danger of compact yet complicated specifications that can hinder user's understanding. Consider the following specification:

```
1 != stencil ((forward(depth=1,dim=1) + backward(depth=1,dim=2)) * pointed(dim=3) +
  ↪  forward(depth=1,dim=2)) * pointed(dim=4) :: a
```

Although this example has a geometric interpretation, it is not immediately obvious. If, however, normalisation is applied to the model before transforming into region syntax, we would obtain:

```
1 != stencil forward(depth=1,dim=1)*pointed(dim=3)*pointed(dim=4) +
  ↪  backward(depth=1,dim=2)*pointed(dim=3)*pointed(dim=4) +
  ↪  forward(depth=1,dim=2)*pointed(dim=4) :: a
```

Despite the introduction of redundant region constants in the specification, the geometric interpretation is easier to understand as a straightforward superimposition of three regions whose extent is defined by a $*$ of region constants.

6 ANALYSIS, CHECKING, AND INFERENCE

We outline here the procedures for checking conformance of source code against specifications (Section 6.2) and for inferring specifications from code (Section 6.3). Both rely on a program analysis that converts array subscripts into sets of index schemes. We outline this analysis first (Section 6.1). Note that the analysis can be made more complex and wide-ranging independent of the checking and inference procedures. At the moment, the analysis is largely *syntactic*, with only a small amount of semantic interpretation of the code.

6.1 Static Analysis of Array Accesses

The analysis builds on standard program analyses: **(1)** basic blocks (CFG); **(2)** induction variables per basic block; **(3)** data-flow analysis, providing a "*flows to*" graph (reaching definitions); and **(4)** type information per variable.

The right-hand side of any assignment statement in a loop is classified based on whether it has array subscripts flowing to it which are neighbourhood offsets (or a combination of neighbourhood and absolute in some dimensions). Each index term is then converted into our model domain by the interpretation $[\![-]\!]^{aterm}$ (Definition 14) and grouped into a finite map from array variables to support sets $\mathcal{S}$, which are the set of subscript schemes that are unioned together to give the model M of an array term in UNF (recall Definition 16: $M = \bigcup_{T \in \mathcal{S}} T$).

This set of subscript schemes is then augmented with multiplicity information (only or mult) depending on whether subscripts are unique or not in the analysed statement. Thus, for each assignment, the analysis generates a map from array variables to values in $\mathsf{Mult}(\mathcal{P}(\mathbb{Z}^N))$.

Example 1. Consider the following five-point stencil example:

```
1 b(i, j) = (a(i, j) + a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1)) / 5.0
```

The support set of subscript schemes produced by the analysis for this code is:

$$\mathcal{S}_0 = \{\{0\} \times \{0\}, \{-1\} \times \{0\}, \{0\} \times \{-1\}, \{0\} \times \{1\}, \{1\} \times \{0\}\}$$

With the added multiplicity information, the resulting model is: $\mathsf{only}(\mathcal{S}_0)$ since no array subscript appears more than once statically.

During analysis, any assignment statement from which array subscripts flow to the current assignment is marked as visited such that the main analysis does not classify it as the root of

an array computation. In the inference procedure, we assign specifications only to these array computation roots.

6.2 Checking Code against Specifications

A checking procedure verifies the access pattern of an array computation in the source language against associated specifications. It proceeds by generating a model from a specification and generating a model from the source code, and comparing them for consistency. Note that both the model of the specification and the model derived from code are sets of integer vectors in UNF (union normal form, see Section 5.4). The notion of consistency is then intuitively and primarily whether these two sets are equal. This leads to a simple function $\mathit{consistent}(M_C, M_S)$ which tests the consistency of a model M_C of the source code against a model M_S of its specification, where $\mathit{consistent} : \mathsf{Mult}(\mathcal{P}(\mathbb{Z}^N)) \times \mathsf{Mult}(\mathsf{Approx}(\mathit{Region}_N)) \to \mathbb{B}$ is defined:

$$\mathit{consistent}(M_C, M_S) = \begin{cases} \mathsf{false} & M_C = \mathsf{once}(x) \wedge M_S = \mathsf{mult}(y) \\ \mathsf{false} & M_C = \mathsf{mult}(x) \wedge M_S = \mathsf{once}(y) \\ \mathit{consistent}'(\mathit{peel}(M_C), \mathit{peel}(M_S)) & \mathit{otherwise} \end{cases}$$

where the helper function $\mathit{peel} : \mathsf{Approx}\ a \to a$ removes the approximation label. The intermediate function $\mathit{consistent}' : \mathcal{P}(\mathbb{Z}^N) \times \mathsf{Approx}(\mathit{Region}_N) \to \mathbb{B}$ is defined:

$$\mathit{consistent}'(M'_C, M'_S) = \begin{cases} m = M'_C & M'_S = \mathsf{exact}(m) \\ m \supseteq M'_C & M'_S = \mathsf{upper}(m) \\ m \subseteq M'_C & M'_S = \mathsf{lower}(m) \end{cases}$$

The top-level *consistent* function checks whether multiplicity information in the specification matches that of the indices, *i.e.* if the specification allows indices to be repeated or not. The consistency function then delegates to *consistent′* to check if the points observed in the array terms match the space defined by the specification.

Lower bounds, marked `atLeast`, require the space defined by the specification to remain inside the set of indices, while an upper bound, marked by `atMost`, requires the opposite: enclosure. In the absence of such modifiers, the two sets are compared for equality since the region defined by the specification must correspond exactly to the space covered by the array subscripts.

Example 2. For example, consider the following specification and associated code:

```
1 != stencil pointed(dim=1) + forward(dim=1,depth=1,nonpointed)*forward(dim=2,depth=2) :: b
2 a(i,j) = b(i,42) + b(i+1, j) + b(i+1,j+1) + b(i+1,j+2) + b(i+1,j+2)
```

A model of the specification (Section 5.2) and a model of the code following static analysis (Section 6.1) are then computed as M_S and M_C respectively in UNF:

$$M_C = \mathsf{mult}((\{0\} \times \mathbb{Z}) \cup (\{1\} \times \{0\}) \cup (\{1\} \times \{1\}) \cup (\{1\} \times \{2\}) \cup (\{1\} \times \{2\}))$$

$$M_S = \mathsf{mult}(\mathsf{exact}(([0 \ldots 0]^{\mathsf{true}} \times \mathbb{Z}) \cup ([0 \ldots 1]^{\mathsf{false}} \times [0 \ldots 2]^{\mathsf{true}})))$$

The top-level function *consistent* finds that both M_C and M_S have matching multiplicity (the mult injection function). Then *consistent′* compares the unwrapped models. Since M_S here is wrapped by exact, the sets of integer vectors defined in M_C and M_S are compared for equality, which holds.

6.2.1 Deciding Set Equality. As explained in Section 5.4, the sets being compared are potentially infinite thus equality cannot be computed by exhaustively comparing elements in each set. Instead, we compile the UNF models of specifications into interval constraints and the UNF models of code into equality constraints, then invoke the Z3 SMT solver [De Moura and Bjørner 2008] to see if the

two sets of constraints represent the same region. The translation from UNF into constraints is straightforward, unfolding definitions of set operations and intervals.

Let $\mathcal{S}$ be the support of a UNF model M_S of an N-dimensional specification. The support contains interval schemes since it is the support of a value in $Region_N$. Further, let FV be a vector of free integer variables with dimension N. We give a boolean expression that represents the region inside M_S as follows:

$$\bigvee_{s \in \mathcal{S}} \bigwedge_{i=1}^{N} \begin{cases} a \leq FV_i \leq b & \pi_i(s) = [a \ldots b]^{\text{true}} \\ a \leq FV_i \leq b \wedge FV_i \neq 0 & \pi_i(s) = [a \ldots b]^{\text{false}} \\ \text{true} & \pi_i(s) = [-\infty \ldots \infty] \end{cases} \tag{1}$$

Similarly, let $\mathcal{R}$ be the support of the set of all subscript schemes defining the UNF model M_C of array reads in an N-dimensional array computation. The following boolean expression represents the region inside M_C, where FV is the same vector of integer variables as above:

$$\bigvee_{r \in \mathcal{R}} \bigwedge_{i=1}^{N} \begin{cases} FV_i = a & \pi_i(r) = \{a\} \\ \text{true} & \pi_i(r) = \mathbb{Z} \end{cases} \tag{2}$$

These two expressions (1) and (2) should be satisfied by exactly the same assignments to the free variables if the models of code and specification are equal.

Going back to Example 2, after stripping M_C and M_S of their multiplicity and approximation injections, we are left with the sets M'_C and M''_S defining the regions in UNF:

$$M''_S = ([0 \ldots 0]^{\text{true}} \times \mathbb{Z}) \cup ([0 \ldots 1]^{\text{false}} \times [0 \ldots 2]^{\text{true}})$$
$$M'_C = (\{0\} \times \mathbb{Z}) \cup (\{1\} \times \{0\}) \cup (\{1\} \times \{1\}) \cup (\{1\} \times \{2\}) \cup (\{1\} \times \{2\})$$

We then translate the support of M'_C and M''_S to the following respective boolean expressions using the above procedure. For the sake of presentation, let x and y stand for FV_1 and FV_2 respectively.

$$\begin{array}{lll} M''_S & \rightsquigarrow & (0 \leq x \leq 0 \wedge \text{true}) \vee ((0 \leq x \leq 1 \wedge x \neq 0) \wedge (0 \leq y \leq 2)) \\ M'_C & \rightsquigarrow & (x = 0 \wedge \text{true}) \vee (x = 1 \wedge y = 0) \vee (x = 1 \wedge y = 1) \vee (x = 1 \wedge y = 2) \vee (x = 1 \wedge y = 2) \end{array}$$

We need to check whether these Boolean expressions are satisfied by exactly the same set of assignments to the free variables. One way to encode this as a satisfaction problem in Z3 is to compare them using $\neq$ and look for a satisfying assignment. If Z3 fails to produce one, since the procedure is complete, we can conclude that they are satisfied by the same set of assignments. In this example, indeed, that is the case. Thus, the specification matches the code.

The query is expressed in the decidable theory of quantifier-free integer linear arithmetic. This may suggest this procedure would be potentially expensive with modest increase in the number of dimensions. However, as the translation above shows, all the constraints generated are merely *simple integer bounds* or equalities. So in this case, the decision procedure for linear arithmetic itself is cheap and Z3 is primarily used as a SAT solver on short formulae. In Section 2.3, we established that 99.5% of array computations have dimensionality at most four. This shows the expected number of free variables in the formulae is low (just four) in the vast majority of cases. To estimate the size of the propositional template of the formulae, we found 96.8% of array computations involve at most 4 array subscript terms, which bounds the size of the subformulae generated from the code. Section 7 shows that all specifications in the corpus comprises no more than two + operations, which is indicative of the size of the subformula generated by the specification. Thus, our approach using Z3 is practical and efficient in all but corner cases.

6.3 Inferring Specifications Automatically

We provide an inference procedure for generating specifications from code. Inferred specifications are then inserted automatically as comments for the root array computations in a loop. This supports maintenance of a legacy code base and aids adoption of the specification language.

The program analysis converts the concrete syntax of array subscripts into a set of subscript schemes $\mathcal{S}_0$. Inference then has two parts. First, "adjacent" index schemes are coalesced into a smaller sets of index schemes which remain in union normal form. Secondly, the resulting union normal form is converted to the specification syntax.

6.3.1 Covering. A covering of (possibly overlapping) index schemes represented via closed intervals is calculated by coalescing *adjacent* index schemes until a fixed-point is reached.

Definition 17 (Adjacent)**.** Two index schemes S and T are *adjacent* written $adjacent(S, T)$, iff $\pi_i(S) = \pi_i(T)$ for all $1 \leq i \leq N$ apart from some dimension k such that $\pi_k(S) = [a, b]$ and $\pi_k(T) = [b + 1, c]$ (these are standard closed intervals, rather than holed intervals of *Interval*).

Given a set of indexing schemes $\mathcal{S}$ and a particular index scheme S we generate a set from coalescing S with adjacent index schemes:

$$coalesce(S, \mathcal{S}) = \{\, S \cup T \mid T \in \mathcal{S} \ \wedge \ adjacent(S, T)\}$$

This is then used by the following recursive procedure:

$$coalesceStep(\mathcal{S}) = \exists T \in \mathcal{S}. \begin{cases} \{T\} \cup coalesceStep(\mathcal{S} - \{T\}) & coalesce(T, \mathcal{S}) = \emptyset \\ coalesceStep(coalesce(T, \mathcal{S}) \cup \mathcal{S} - \{T\}) & otherwise \end{cases}$$

This gives a specification rather than an implementation. Our implementation represents sets by a list, and so $\exists T \in \mathcal{S}$ above corresponds to deconstructing the list into its head element T (corresponding to picking an element from the set). If T has no adjacent index schemes, then coalescing is attempted on the rest of the set, and T is returned in the result. Otherwise, the set of coalesced index scheme is computed and this is passed to a recursive procedure called *coalesceStep*, along with the rest of the elements).

The fixed-point of $coalesceStep(\mathcal{S}_0)$ is computed, giving a covering over the initial subscript space.

Lemma 7. *For a set $\mathcal{S}$ of index schemes, $coalesceStep(\mathcal{S})$ is a set of index schemes. Furthermore, if every index scheme in $\mathcal{S}$ has components which are all either finite closed intervals or $\mathbb{Z}$, then every index scheme in $coalesceStep(\mathcal{S})$ similarly has all components as finite closed intervals or $\mathbb{Z}$.*

Corollary 1. *A subscript scheme has components which are either $\mathbb{Z}$ or a singleton set (which is a closed interval* e.g.*, $\{n\} = [n, n]$) thus the fixed-point of coalesceStep on $\mathcal{S}_0$ (the initial set of subscript schemes from the analysis) yields a set of index schemes whose components are finite intervals or $\mathbb{Z}$.*

Example 3. Recall the five-point stencil example:

```
1  b(i, j) = (a(i, j) + a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1)) / 5.0
```

which has a model with support $\mathcal{S}_0 = \{\{0\} \times \{0\}, \{-1\} \times \{0\}, \{0\} \times \{-1\}, \{0\} \times \{1\}, \{1\} \times \{0\}\}$. The fixed-pointed of *coalesceStep* applied to $\mathcal{S}_0$ is reached within two steps (note each component of each index scheme in $\mathcal{S}_0$ can be represented as a finite closed interval):

$$coalesceStep(\mathcal{S}_0) = \{[-1, 0] \times [0, 0],\ [0, 0] \times [-1, 0],\ [0, 0] \times [0, 1],\ [0, 1] \times [0, 0]\}$$

$$coalesceStep^2(\mathcal{S}_0) = \{[-1, 1] \times [0, 0],\ [0, 0] \times [-1, 1]\} = coalesceStep^3(\mathcal{S}_0)$$

6.3.2 From Index Schemes to Syntax. Let the fixed point of *coalesceStep* on input $\mathcal{S}_0$ be written as $\mathcal{S}_\omega$, the covering set. Next, $\mathcal{S}_\omega$ is translated into specification syntax in four stages.

(1) We check whether $\mathcal{S}_\omega$ is the top indexing scheme $\mathbb{Z}^N$. This occurs if absolute indices appear in each of the dimension across the array subscripts, leading to an unconstrained access pattern, which is not represented in our syntax, ending inference.
(2) Otherwise, we determine if the indexing schemes are interval schemes (Definition 7) so that they can be represented exactly by our syntax. It not, they are altered into interval schemes to be represented as approximations.
(3) At this point we might have redundant regions in our model. We apply basic optimisations to reduce the eventual number of region constants.
(4) Interval schemes are mapped into joins (+) of meets (*) of region constants, where some intervals are split into multiple separate region constants.

An index scheme is an interval scheme (Definition 7) if each component can be represented as a holed interval $[a \ldots b]^c$. Since index schemes of the previous stage are all formed from closed intervals or $\mathbb{Z}$ (Corollary 1) then these index schemes are interval schemes if their interval components have lower bounds ≤ 0 and upper bounds ≥ 0. Otherwise, an approximate specification is generated.

An upper bound is established by *elongating* any index schemes in multiple dimensions such that they become interval schemes. The elongation function is given as:

$$elongate([a, b]) = \begin{cases} [0 \ldots b]^{\text{false}} & a > 1 \\ [a \ldots 0]^{\text{false}} & b < -1 \end{cases}$$

If any component of an index scheme is elongated, then the whole index scheme is elongated. In $\mathcal{S}_\omega$, all index schemes that are representable as interval schemes form a lower bound, whilst those that are not interval schemes form an upper bound by their elongation. A lower bound may not be generated if none of the indexing schemes are interval schemes. In the case of our example, the resulting $\mathcal{S}_\omega$ comprises a set of interval schemes, thus we can generate an exact specification.

Optimising regions. The inference algorithm makes no guarantees regarding minimality of inference results. Indeed, not all representable regions can be broken into non-overlapping regions. However, there may also be superimposed regions that are strictly enclosed by another or a more general sense of adjacency between regions than accounted by coalescing in the previous step. To improve on inference output, we do a simple optimisation pass on regions before translating back to specification syntax.

The union lemma (Lemma 2) and holed interval identities (Lemma 3) together mean if two interval schemes differ only in one of their dimensions, we can produce a single interval scheme that agrees with the original in invariant dimensions and is the union of intervals in the differing dimension. This in effect means that some adjacent regions combined through union can be expressed as one, resulting in fewer region constants. Coalescing prior to the optimisation may not account for this for two reasons. Firstly, adjacency as defined in the previous step does not consider two intervals that are not right next to each other around the origin to be adjacent *e.g.* $[-3 \ldots 0]^{\text{false}}$ and $[0 \ldots 3]^{\text{false}}$ are not adjacent because neither of them include 0. This optimisation handles this case. Secondly, since elongation occurs after coalescing, this optimisation is needed to see whether the elongated region is adjacent to other regions (elongated or otherwise).

The equational theory (Section 4.1) contains a subsumption rule that allows two regions superimposed with each other where one of which is entirely contained within the other to be removed, leading to fewer regions. Since the model is sound with respect to the equational theory, the subsumption rule extends to interval schemes. Thus, we can remove some of the redundant regions.

These two optimisations interact. For example, the combination of two adjacent regions might reveal that a region that was not contained by either of the original regions is contained by the new combined region. To increase the effectiveness of both optimisations, we apply them in succession and repeatedly until a fixed point of regions is reached, *i.e.* the optimisations no longer reduce regions. Such a fixed point exists for arbitrary regions because both optimisations can only reduce the number of regions, *i.e.* they are strictly decreasing functions on the number of regions.

Translation from interval schemes. Recall the model of region constants in Figure 2b where $[\![-]\!]$ maps to holed intervals which have either 0 as the lower-bound or upper-bound for `forward` and `backward` respectively, or both 0 for `pointed`, or $-k$ and k lower and upper bounds for `centered`. We can invert this map to generate region constants from holed intervals. However, some intervals computed from the above steps may not match the constraints of this interpretation, *e.g.*, $[-2 \ldots 1] \times [-1 \ldots 1]$. By Lemma 2, we can split interval schemes into two, for example, producing $[-2 \ldots 0] \times [-1 \ldots 1]$ and $[0 \ldots 1] \times [-1 \ldots 1]$. The following function *split* iterates over the components of an index scheme, splitting apart interval schemes when necessary:

$$split([a \ldots b]^c \times S) = \begin{cases} \{[a \ldots 0]^c \times T \mid T \in split(S)\} \cup \{[0 \ldots b]^c \times T \mid T \in split(S)\} & a \neq 0 \wedge b \neq 0 \wedge |a| \neq b \\ \{[a \ldots b]^c \times T \mid T \in split(S)\} & otherwise \end{cases}$$

This forms a set of interval schemes to which $[\![-]\!]^{-1}$ (Figure 2b) is well-defined for each component.

The final stage of inference maps the set of interval schemes to a region specifications.

Definition 18. (Convert) We map a component of an interval scheme to a region constant using $[\![-]\!]^{-1}$, combining each component with $*$ and combining the resulting regions by $+$:

$$convert_N(\mathcal{S}) = \sum_{S \in \mathcal{S}} \prod_{\pi_i(S) \neq \mathbb{Z}} [\![\pi_i(S)]\!]^{-1} \tag{3}$$

Where summation is by the syntactic $+$ and product is by the syntactic $*$.

In the example, $[-1 \ldots 1] \times [0 \ldots 0]$ is mapped to `centered(dim=1,depth=1) * pointed(dim=2)`, and $[0 \ldots 0] \times [-1 \ldots 1]$ is similar. When combined by $+$ we obtain the following specification:

```
!= stencil readOnce, centered(dim=1,depth=1)*pointed(dim=2) + pointed(dim=1)*centered(dim=2,depth=1)
```

Theorem 3 (Inference soundness). *For all region specs R, then $infer([\![R]\!]_N)_N \equiv R$*

7 EVALUATION

To study the effectiveness of our approach in terms of its applicability to real code, we applied our tool to the corpus described in Section 2. We used the inference procedure of the previous section to generate array specifications for the corpus to assess the design of the language, *i.e.*, the extent to which our specification language can describe real code. We did not expect to infer specifications for every array computation identified in the code base because our analysis restricts the array-subscript-statements, along the lines of Section 2.

Overall we ended up inferring 87,280 specifications of which 73,372 are `stencil` specifications and 13,908 are `access` specifications (an array-computation whose left-hand side is a variable rather than a subscript). This shows that we can express a large number of array access patterns within our high-level abstractions and it validates our initial design choices informed by our data (Section 2.4).

The majority of specifications generated were relatively simple but we found significant numbers of more complex shapes. We grouped common patterns into categories, and studied the frequency

of different syntactic constructs of our specification language:

All pointed specifications comprise only `pointed` regions (with no approximation modifiers). We inferred 79,415 such specifications. Common examples of this were pointwise transformations on data (such as scaling). These represent fairly mundane array computations which are unlikely to be a source of indexing error. The remaining 7,865 specification we consider to be non-trivial, *i.e.* with a higher potential for programming error, since they involve more complex indexing patterns.

Single-action specifications comprise one `forward`, `backward`, or `centered` region constant combined via `+` or `*` with any number of `pointed` regions. We identified 6,879 single-action specifications, of which 5,282 were single-action with a `nonpointed` modifier.

Multi-action specifications comprise at least two of `forward`, `backward`, or `centered` regions, combined with any number of `pointed` regions. We identified 971 multi-action specifications[5] out of which 403 had regions combined only with `*` and 568 combined with a mix of `*` and `+`. The multi-action classes represent more complex array access patterns.

We measured the frequencies of occurrences of `*` and `+` within the inferred specifications:

Occurrences:	0	1	2	3	4	5	6	7	8	9
*	43,044	22,112	19,390	2,259	325		148			2
+	86,472	541	267							

Thus we see that specifications tend to use `+` rarely, *i.e.*, they do not involve the union of two regions. Whilst the proportion of more complex specifications is low, the absolute number of 568 specifications with multiple region types composed via a mix of `*` and `+` presents a significant amount of use cases for our tool given that we are looking at 11 software packages.

Bounded specifications are approximations, either over-approximations with `atMost` or under-approximations with `atLeast`. We inferred 327 `atMost` specifications and 21 `atLeast` specifications, the latter of which were always paired with an accompanying upper-bound `atMost` specification. Thus, we see that almost all of our specifications are exact, rather than approximate. This helps demonstrate that our language is expressive enough to capture common behaviour since more than 99% of our specifications are precise. The inclusion of bounded specifications allows us to say something approximate about the few remaining patterns, without complicating the core language.

ReadOnce specifications have the modifier `readOnce`, which occurred 77,178 times (88% of specifications) which correlates with the high proportion of unique-subscript expressions in the initial study where 87% of all RHS neighbour/affine array computations had unique subscripts.[6]

7.1 Limitations

There were various reasons why we did not infer specifications on every looped array computation:

- **Computed induction variables** where an index `x` is computed from an induction variable via an intermediate definition, *e.g.*, `x = i+1`, or using functions that we cannot analyse;
- **Inconsistent LHS/RHS** 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)`.

[5]The reader may notice that there are 7,421 single action and 972 multi-action specifications, giving a total of 8,393 specifications. There are 8,409 non-all-pointed specifications. The remaining 16 were all-pointed specifications with an `atLeast` modifier, which fall between the three categories here.

[6]The discrepancy is accounted for by the fact that the initial study ignored consistency when considering uniqueness of subscripts, hence the data set of the study will include more array computations (in Hypothesis 5). Nonetheless, we see roughly the same proportion in the evaluation here as in the initial study.

- **Fortran overloading and index-range expressions** Fortran supports array ranges of the form `a(n:m)` meaning a subarray of `a` between indices `n` and `m`. This can be combined with overloaded operations on arrays to write array computations. We have only minimal support for this when a wildcard range is given, of the form, `a(:)`.

7.2 Detecting Errors in the 2D Jacobi Iteration

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

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

We infer a precise specification of its shape as:

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

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

7.3 Verification Case Study

Our approach does not target a class of bugs that can be detected automatically (push-button verification). Instead, array indexing bugs are identified relative to a specification of the intended access pattern. We sought to extract examples of program code from our corpus where array specifications could have prevented or identified an error. Classifying errors is difficult, though, because it requires knowledge of the programmer's intent, and a means to convert that intent into a specification. However, version-control commit messages accompanying a code change often indicate their intent informally, at least. We developed a partially-automated set of scripts to comb through the logs of certain version-control repositories and hunt down changesets that looked like they might provide promising examples of array computation-related bugs. We then examined each of those changesets and its accompanying log message to track down the meaning of the commit and decide whether it might qualify as a case where our specification tool could have helped.

Not all codebases have the same quality of code history. We focussed on the largest and most well-logged package: the Unified Model (UM), and some of its associated packages. On the UM, our script applied our specification inference tool to successive code revisions[7] and filtered out those that did not induce specification changes. We then manually assessed the remaining commit messages and looked up their associated bug reports on the UM bug-tracking system. We discovered relevant past bug fixes which fell into a few categories in the UM, for which we report a representative example in Section 7.3.1.

In general, we expect relevant bugs to appear more frequently earlier in a project, when it is less mature, but archived revisions of the UM prior to 2008 were not available to us (the UM is a much older project). Furthermore, commit logs are in general a conservative view of the real number of fixed bugs; in practice, bugs are caught between revisions by testing and are therefore not necessarily reflected in commit logs [Negara et al. 2012].

7.3.1 Catching Bugs. In the Unified Model we identified several previously fixed off-by-1 errors that could have been avoided using our specifications. For example, this array computation was

[7] This was applied to revisions between versions 8.6 and 10.7 of the UM.

identified as having a bug which was given a specification `pointed` in all dimensions by our inference:

```
!= stencil readOnce, pointed(dim=1)*pointed(dim=2)*pointed(dim=3)
```

The programmer made it clear in the comments after the bug fix to this code that the intention had been `forward` in the third dimension. The patched code has the inferred specification:

```
!= stencil readOnce, forward(depth=1, dim=3, nonpointed)*pointed(dim=1)*pointed(dim=2)
```

Thus, the initial bug could have been detected given the above specification.

7.3.2 Aiding Refactoring. One of the goals of CamFort is to provide tools that enable refactoring without changing behaviour, and the array specification feature was designed with that in mind. But we realised that there were also many cases where CamFort could help with refactorings precisely because they required changes in behaviour. For example, a relatively common change observed in the version control logs for the UM is the refactoring of array dimensions: either re-ordering, adding or deleting dimensions. Any of these, if not perfectly propagated throughout all uses, could result in unexpected outcomes. However, CamFort will pick up the difference during specification checking. Ideally, an experienced programmer might take advantage of the region variable feature of CamFort in order to minimise the number of specifications that need changing and gain the full time-saving advantage. But even if not, a quick search for specifications can be used to update them quickly and then any code that is not yet refactored will trigger an error.

The following shows two example specifications before and after a refactoring which adds a single dimension to an array, adapted from an refactoring found in the UM:

```
!= region :: r = pointed(dim=1)
! becomes
!= region :: r = pointed(dim=1)*pointed(dim=2)
```

Thus, our tool can help to ensure the correctness of array refactorings.

7.4 Threats to Validity

The composition of the software corpus has an effect on the number and kinds of specifications that we infer but we have found the general conclusions remain the same. The biggest discrepancy we found was for Hypothesis 5 in Section 2 where we found 67% of array computations read once-only from a particular subscript. However, if one considers the freely distributable models from our corpus (excluding MUDPACK, Computational Physics, UM, E3ME and Hybrid4) then this number drops to 53%, though this still supports the hypothesis. Nevertheless, we encourage interested parties to make use of our implementation to check how their software packages compare.

We generated the numbers in our empirical study of array computations (Section 2) and our evaluation (Section 7) with different tools. This has given rise to some inconsistencies in their results. For example we found that 55.86% of the 133,577 identified array assignments corresponded to consistent stencils. This makes a total of 74,616 stencils compared with the 73,372 we found in the evaluation. This is in part due to the tool of Section 2 being more conservative in some of its classifications, though there may also be bugs in either. Despite this we still find that both studies produce consistent results in terms of the relative rates of different stencil shapes.

8 RELATED WORK AND CONCLUSIONS

Various deductive verification tools can express array indexing in their specifications, *e.g.*, ACSL for C [Baudin et al. 2008]. A specification can be given for an array computation but must use fine-grained indexing similar to the code itself [Burghardt et al. 2010, Example 3.4.1] and therefore is prone to indexing errors. Our approach is more abstract—it does not reify indexing in the specification, but provides simple spatial descriptions which capture many common patterns.

Kamil et al. [2016] propose *verified lifting* to extract a functionally-complete mathematical representation of low-level, and potentially optimised, stencils in Fortran code. This extracted predicate representation of a stencil is used to generate code for the Halide high-performance compiler [Ragan-Kelley et al. 2013]. Thus they must capture the full meaning of a stencil computation which requires significant program analysis. For example, they report that some degenerate stencil kernels take up to 17 hours to analyse and others require programmer intervention for correct invariants to be inferred. By contrast, it takes roughly 1.5 hours on commodity hardware (3.2 GHz Intel Core i5, 16 GB of RAM) to infer and generate stencil specifications for our entire corpus.

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

Our work has similarities with kernel verification for General-Purpose GPU programming, such as in Blom et al. [2014]. However, their focus is on the synchronisation of kernels and the avoidance of data races while we are interested in correctness. Solar-Lezama et al. [2007] give specifications of stencils using unoptimised "reference" stencils, coupled with partial implementations which are completed by code generation. Their primary purpose is optimisation rather than correctness, and so require a more elaborate language.

Tang et al. [2011] give a stencil compiler called Pochoir and an accompanying language for writing stencils embedded in C++ (with Cilk extensions [Blumofe et al. 1996]) that are compiled into parallel programs based on trapezoidal decompositions with hyperspace cuts. Pochoir specifications are used for describing the kernel, boundary conditions, and shape of the stencil. Their language, like ours, makes use of spatial properties (*dimension* and *depth*). They use their shape declarations for finding out-of-bounds violations in the kernels. One significant difference is that relative offsets from induction variables are written explicitly meaning there are as many offsets in the shape declaration as there are offsets in the original kernel. The premise of CamFort is that such offset-heavy kernels lead to programming errors, hence we consider Pochoir shape declarations a poor fit for catching offset-related bugs. Tang et al. explain that the reason shape declarations are given in addition to the computation kernels as opposed to inference from the kernel is because the computation kernels might have arbitrary function calls that hinder shape inference. We provide a dataflow aware *syntactic* shape inference procedure which works well for catching shape-related bugs. This approach might not apply for Tang et al. since their compilation requires a *semantically* precise shape. Pochoir is aimed at programmers reluctant to implement high-performance algorithms. Like much of the related work, the focus is on optimisation rather than correctness.

Abe et al. [2013] introduce a form of model-checking to partitioned global address space (PGAS) languages that are often used to write paralellised stencil computations. PGAS languages can parallelise computation on large data arrays over many processors. They divide up the array into subarrays, each one local to a processor, but allowing remote access to other processors' arrays. Abe *et al.* analyse stencil computations written in the XscalableMP (XMP) directive-based language extension for Fortran and C. XMP is a PGAS language and is intended to be a simpler replacement for MPI. The XMP directives are written into comments that annotate Fortran or C programs, and then are translated by a compiler pre-processor into efficient code that divides up the work onto

multiple processors. Their analysis of stencil computations is dependent upon the implementation details of XMP. For example, the descriptions of stencil computations are abstracted to describe only their boundary elements, not the entire computation. This can be accomplished since in the XMP implementation the access to boundary elements requires a different set of underlying instructions than the non-boundary elements do. The XMP specifications tell the pre-processor how to translate the code into a parallelised form that properly meshes together. The model-checking procedure used by the XMP-SPIN checker is mainly intended to consider the correctness of the inter-processor communication. It is used to synchronise the portions of one processor's address space overlapping with that of the other processors.

In contrast, we integrate into existing codebases and do not affect the behaviour of the programs we annotate, so that they may be processed by ordinary compilers and tools. Our specification language describes the shape of array computations and the purpose of the checking procedure is to ensure that code correctly meets a specification, not to optimise or generate code.

8.1 Concluding Remarks

We have proposed a novel solution to a common programming task in scientific computing. Our approach to language design is also relatively uncommon — informed by a quantitative study of existing code. The resulting language is flexible and expressive, capturing roughly 90,000 array computations across 1.1 million lines of Fortran with simple, short, abstract spatial specifications (taking inspiration from the numerical literature).

We are working with our project partners to integrate our approach into their workflows. We will then be able to assess over months (or years) of program evolution how our specifications can aid correct program design. In lieu of this kind of data (which will take time to gather), we provided two kinds of evidence of the value of our approach: (1) applicability to real code; and (2) existence of real bugs targetable by our tool. The evidence of applicability is *a priori*: our specification design is based on the empirical study of common patterns. Applicability was also shown by running the inference procedure over the corpus yielding 87,280 specifications of which 7,865 are non-trivial (not just pointwise array operations). We gave evidence of targetable bugs via our heuristic analysis of the UM repositories (Section 7.3). Our tool would have detected these bugs given a prior correct specification, or would at least have helped the programmer solidify their thinking.

We are positive about usability of the approach given its relative simplicity and because our language is directly related to the style in which these algorithms are taught and described in textbooks. The inference tool also helps users, generating specifications for inspection and integration.

We provide an implementation[8] of this approach as part of the CamFort project. A VirtualBox appliance reproducing this paper is also available.[9]

ACKNOWLEDGEMENTS

This work was supported by the EPSRC [grant number EP/M026124/1]. We would like to thank Alan Mycroft for his general feedback on the approach and suggestions regarding evaluation and Ben Moon for work on the CamFort implementation. We also thank the anonymous reviewers for their thorough comments which helped to improve the paper.

Thank you to our collaborators for allowing us to analysis their code, especially Andrew Friend and the developers of E3ME.

Material produced using the Met Office Software. We thank the Met Office for granting us access to their code base under an agreed Non-Commercial Research Use collaboration license.

[8] https://github.com/camfort/camfort
[9] https://doi.org/10.17863/CAM.12627

REFERENCES

T. Abe, T. Maeda, and M. Sato. 2013. 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*. 365–374. DOI: http://dx.doi.org/10.1109/IPDPSW.2013.90

J. Adams. 1991. *MUDPACK: multigrid software for linear elliptic partial differential equations, version 3.0.* National Center for Atmospheric Research, Boulder, Colorado. Scientific Computing Division User Doc.

Krste Asanovìc, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

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

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2008. ACSL: ANSI C Specification Language. (2008).

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

L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, and others. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

Stefan Blom, Marieke Huisman, and Matej MihelÄŤiÄĞ. 2014. Specification and verification of GPGPU programs. *Science of Computer Programming* 95, Part 3 (2014), 376 – 388. DOI: http://dx.doi.org/10.1016/j.scico.2014.03.013 Special Section: {ACM} SAC-SVT 2013 + Bytecode 2013.

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

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

Mistral Contrastin, Matthew Danish, Dominic Orchard, and Andrew Rice. 2016. Lightning Talk: Supporting Software Sustainability with Lightweight Specifications. In *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4), University of Manchester, Manchester, UK, September 12–14*, Vol. 1686. CEUR Workshop Proceedings.

Mistral Contrastin, Matthew Danish, Dominic Orchard, and Andrew Rice. 2017. CamFort - refactoring, analysis, and verification tool for scientific Fortran programs. https://camfort.github.com. (2017). Accessed: 23rd August 2017.

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

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

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

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

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

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

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

Dimitri Komatitsch, Jeroen Tromp, and others. 2016. SPECFEM3D. https://github.com/geodynamics/specfem3d. (2016). Accessed: 15 November 2016.

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. 2012. Is it dangerous to use version control histories to study source code evolution?. In *ECOOP*, Vol. 12. Springer, 79–103.

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

Dominic Orchard, Mistral Contrastin, Matthew Danish, and Andrew Rice. 2017. *Proofs for 'Verifying Spatial Properties of Array Computations'*. Technical Report UCAM-CL-TR-911. University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom.

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

Tao Pang. 1999. An introduction to computational physics. (1999). 1st Edition.

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

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

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

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. *SIGPLAN Not.* 42, 6 (June 2007), 167–178. DOI: http://dx.doi.org/10.1145/1273442.1250754

David Sorenson, Richard Lehoucq, Chao Yang, Kristi Maschhoff, Sylvestre Ledru, and Allan Cornet. 2017. ARPACK-NG. https://github.com/opencollab/arpack-ng. (2017).

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

Elena Tolkova. 2014. Land–Water Boundary Treatment for a Tsunami Model With Dimensional Splitting. *Pure and Applied Geophysics* 171, 9 (2014), 2289–2314.

Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. Citeseer, 347–359.

David A Wheeler. 2001. SLOCCount. (2001).

Damian R Wilson and Susan P Ballard. 1999. A microphysically based precipitation scheme for the UK Meteorological Office Unified Model. *Quarterly Journal of the Royal Meteorological Society* 125, 557 (1999), 1607–1636.