Verifying Spatial Properties of Array Computations

ANONYMOUS AUTHOR(S)

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 and give verification case studies of bug fixes that are detected by our approach. We found roughly 80,000 targets for specification across roughly 1.4 million lines of code, showing that the vast majority of array computations read from arrays in a pattern with a simple, regular, static shape.

ACM Reference format:
Anonymous Author(s). 2016. Verifying Spatial Properties of Array Computations. 1, 1, Article 1 (January 2016), 26 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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 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 analysis of real code.

We focus on one common concept: *arrays*, the core data structure in numerical modelling code, typically representing discrete approximations of physical space or data sets. A common programming pattern, sometimes referred to as the *structured grid* pattern (Asanovic̀ et al. 2006), traverses the index space of one or more arrays via a loop, computing elements of another array or reducing the elements to a single value. For example, the following Fortran code computes the one-dimensional discrete Laplace transform approximating a derivative:

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


DOI: 10.1145/nnnnnnn.nnnnnnn

This is a *stencil* computation, an idiom 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, cellular automata.

Such array computations are prone to error 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 which are read with different data access patterns, across two dimensions, with dense index-manipulation:

```
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))- &
  (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))) /(4.0*delx)
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))- &
  (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))) / (4.0*dely)

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
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, or transforming the wrong variable *e.g.* `(i+1,j)` instead of `(i,j+1)`.

In practice, the typical development procedure for complex stencil computations involves some ad hoc testing to ensure that no 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 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 base, totaling 1.4 million lines of code (Section 2). 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 combinators to express common array patterns (Section 3). As an example, the one-dimensional Laplace program is specified in our language by:

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

That is, `a` is accessed with a symmetrical pattern in its first dimension ("centered") to a depth of one in each direction and its result contributes to an array write. The Navier-Stokes example has two specifications:

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

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 expressions. 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 (Constrastin et al. 2017), an open-source program analysis tool for Fortran. 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 and provides documentation. We also provide an inference procedure which efficiently produces 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) are derived from a model of array access patterns, which is also used to give a denotational model of our specification language (Section 5.2).

Using the inference-mode of the tool, we applied our tool to our original corpus (Section 7). Our tool identifies and infers specifications for 80,731 array computations in the corpus. Of those computations we found, 7,391 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 report on instances of code revisions from the commit histories of our corpus where a correct specification would have spotted an error (which was latter corrected) or would have assisted in refactoring (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 a *base 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, i, j, k range over induction variables.

A *derived induction variable* is an expression of the form $a * i + b$, where a and b are constant expressions, *i.e.*, an affine expression on an induction variable i.

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 *index* $\bar{e}$, specified by a comma-separated sequence of integer expressions, in expanded form as $(e_1, \ldots, e_N)$. An index e_i is called *relative* if the expression involves an induction variable. An *absolute index* is a integer expression which is constant with respect to the enclosing loop.

Definition 3 (Origin)**.** An array subscript $a(\bar{e})$ has an *origin index* if all $e \in \bar{e}$ are induction variables, *e.g.*, `a(i, j)`.

Definition 4 (Neighbourhood 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. That is, a neighbourhood index is a constant translation of an induction variable. (The relation $\equiv$ identifies terms up-to commutativity of $+$ and the inverse relationship of $+$ and $-$ *e.g.*, $(-b) + i \equiv i - b$).

2 AN 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 based on constant offsets from (base or derived) induction variables;
(2) Most loop-array computations of the previous form read from arrays with a *contiguous* pattern, *e.g.*:

```
1    b(n) = 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 index just once. The significance of this hypothesis is that, if true, it would be useful to provide a specification of index uniqueness, so accidental repetition of subscripts 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 or autotuners.

2.1 Methodology

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

(1) The Unified Model (UM) developed by the UK Met Office for weather modelling (Wilson and Ballard 1999);
(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 chemstry model (Bey et al. 2001);
(5) Navier, a small-size Navier-Stokes fluid model, based on (Griebel et al. 1997);
(6) Computational Physics (CP), programs from a popular textbook (Giordano and Nakanishi 1997);
(7) BLAS, a common linear-algebra library used in modelling (Blackford et al. 2002);
(8) ARPACK-NG, an open-source library for solving large-scale 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 lines of Fortran code (2.4 million including comments and white space). The UM and GEOS-Chem packages are the largest at ≈630kloc and ≈450kloc respectively. Appendix A 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).

We built a code analysis tool based on CamFort, an open-source Fortran analyser (Constrastin 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 is not able to parse all of the corpus, some of which contains non-standard code. Of the 1.4 million lines, over 1 million was parsed (see Appendix A for exact numbers).

Classifications are computed from all the information that flows into an assignment. For example, the following decomposes a one-dimensional three-point pattern 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 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 multiple assignments can flow to multiple classified statements.

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	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)`

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

Property	Shorthand	Classifications (of RHS pattern)	Example
Contiguity	(contig)	Contiguous	`a(i) + a(i+1) + a(i+2)`
	(disjoint)	Non-contiguous	`a(i) + a(i+2)`
Reuse	(readonce)	Unique subscripts	`b(i) = a(i) + a(i+1)`
	(mult)	Repeated subscripts	`b(i) = a(i) + a(i)`
Positioning	(origin)	Includes origin	`a(i)` or `a(i+1, j)` (in 2^{nd} dimension)
	(straddle)	Within distance 1 of origin	`a(i+1,j) + a(i-1,j)`
	(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 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 an arrays are read from with a static pattern based on constant offsets from (base or derived) induction variables;

We identified 108,773 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	34	0.03%	
Neighbourhood	65,334	60.06%	81.43%
Neighbourhood + absolute	23,209	21.34%	
Other	20,196	18.57%	
Total	108,773	100.00%	

Since, affine and neighbour patterns are static, we group them (*Grouping 1*) to see that 81.43% of the array computations (88,577) 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.

RHS classification	**Number**	%(of whole)	**Position**	**Number**	%(of whole)
Affine, contiguous	31	0.03%			
Affine, non-contiguous	3	0.00%			
Neighbour, contiguous	83,689	76.94%	away	152	0.14%
			origin	81,309	74.75%
			straddle	2,228	2.05%
Neighbour, non-contiguous	4,854	4.46%	away	2	0.00%
			origin	4,836	4.45%
			straddle	16	0.01%
Other	20,196	18.57%			
Total	108,773	100.00%			

Thus, 76.94% of array computations have only neighbour indices (which may include some absolute indices) in a contiguous pattern on the right hand side, with 76.80% either including the origin (74.75%) or offset by a distance of 1 from the origin (2.05%). This confirms both Hypothesis 2 and Hypothesis 3. Only 152 instances of index patterns not immediately next-to or over the origin were found. There is a non trivial amount of non-contiguous arrays access, 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 subclassified our data by RHS and LHS categories being either a variable, or induction-variable indexed, neighbour-indexed or affine-indexed array. 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%	4,591	4.22%	
LHS Vars	RHS affines	0	0.00%	4	0.00%	
LHS IVs	RHS neigh	4,560	4.19%	50,861	46.76%	
LHS neigh	RHS neigh	2,358	2.17%	16,336	15.02%	61.79%
LHS IVs	RHS affines	7	0.01%	0	0.00%	
LHS affine	RHS affines	0	0.00%	6	0.01%	
LHS neigh	RHS affines	8	0.01%	0	0.00%	
LHS affine	RHS neigh	23	0.02%	0	0.00%	
other		30,019	27.60%			
Total		36,975	34.00%	71,798	66.01%	

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. Thus, we find that 66.01% of array computations have a static pattern on the LHS and RHS that is either a neighbourhood or affine, and 61.79% are *stencils*, writing to an array subscript on the left. Thus, the data supports our hypothesis that stencils are common. Based on this grouping, we considered two subclassifications of different kinds of left-right-hand side relationship and contiguity:

Contiguity	**Number**	%(whole)
Contiguous stencil	68,246	62.74%
Non-contiguous stencil	3,552	3.27%
other	36,975	33.99%

Consistency	**Number**	%	**relativised #**	%
Direct	47,448	43.62%	636	0.58%
LHSsubset	9,812	9.02%	65	0.06%
LHSsuperset	10,110	9.29%	84	0.08%
Permutation	3,604	3.31%	39	0.04%
other	36,975	33.99%		

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 stencils are contiguous, but there is some non-contiguity. Most stencils have matching induction variables on the left and right-hand sides but again there are non-trivial amounts of the other kinds of relationship.

(5) Many array computations read from each particular index just once.

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**	%
RHS neighbour/affine unique subscripts	78,349	72.03%
RHS neighbour/affine repeated subscripts	10,228	9.40%
other	20,196	18.57%

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 108,773 array computations, we recorded three additional properties: the dimensionality of array subscripts involved, the number of subscript terms contributing to the computation, and the length of the dataflow path (*i.e.*, how many intermediate assignments contribute to the array computation):

Dimensionality	0	1	2	3	4	5	6	7	8	9	10
Count	0	43,227	32,830	24,621	7,608	467	13	5	0	0	3

# subscripts:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20≤
Count	0	84,690	14,377	4,234	2,367	881	607	192	457	284	207	45	72	33	16	41	40	27	15	4	184

Dataflow length:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15≤
Count	106,712	13,323	7,091	3,898	3,031	1,695	1,333	965	792	704	482	430	339	310	270	2544

The data for "number of subscripts" shows that most computations are relatively simple, with only 1-3 array subscripts. However, some are complex, with 184 with over 20 indexing terms. We had one example with 96 contributing array subscripts. Thus, we see that there are many cases in practice with complex patterns. We aim to provide support for such code with our specification language.

The rest of the data here is of use later when considering the performance of our checking procedure (Section 6.2).

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

Fig. 1. Specification syntax (EBNF grammar)

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 we 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 (81.43% of our array computations);
- Access patterns are mostly contiguous (76.94% of all array computations) and cross or straddle the origin (75.75%). 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) (66.01% of all array computations).
- Stencils will be the primary focus (61.79%) but we will also allow specifications on assignments where the LHS is a variable (*e.g.*, in a reduction) (4.22%).
- There is some non-contiguity (4.46%) 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.

3 A SPECIFICATION LANGUAGE FOR THE SHAPE OF ARRAY PATTERNS

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

Region constants (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 given as a positive natural number. Each constant except `pointed` has a depth parameter n given as a positive natural number; `pointed` regions implicitly have a depth of 0.

A `forward` region of depth n specifies a contiguous region in dimension d starting at the origin. This corresponds to specifying neighbourhood indices in dimension d ranging from i to $i+n$ for some induction variable i. Similarly, a `backward` region of depth n corresponds to contiguous indices from i to $i-n$ and `centered` of depth n from $i-n$

to $i + n$. A `pointed` stencil specifies a neighbour index i. For example, the following shows four specifications with four consistent stencil kernels reading from arrays `a`, `b`, `c` and `d`:

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

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

!= stencil centered(depth=1, dim=1) :: c
e(i, j) = (c(j-1) + c(j) + c(j+1))/3.0

!= stencil pointed(dim=3) :: d
e(i, j) = d(0, 0, i)
```

If a dimension contains only absolute index terms then it is left unspecified.

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

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

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 (that is, its multiplicity is one). For example, all of the previous examples could have `readOnce` added:

```
!= 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. This modifier provides a way to rule out any accidental repetition of array subscripts. The notion is similar to that of linear types (Wadler 1990), where a value must be used exactly once. We opt for a more informative and easily understood name `readOnce`.

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:

```
do i = 1, n
   != 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` is that there is no left-hand side with which to check consistency or 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 $*$ and $+$ are both idempotent, commutative, and associative;
Subsumption If S and R are regions with $S \preceq R$, then $S{+}R \equiv R$ and $S{*}R \equiv S$.
Distribution $*$ distributes over $+$ and dually $+$ distributes over $*$, meaning if $\mathbf{R}$, $\mathbf{S}$, and $\mathbf{T}$ are regions, then we have the following dual equivalences:

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

Overlapping pointed If **R** is one of `forward`, `backward`, or `centered`, then we have the following:

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

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

$$\texttt{ctd}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p1}) \equiv \texttt{fwd}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p2})+\texttt{bwd}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p3})$$

Here **p1** is `nonpointed` if both **p2** and **p3** are `nonpointed`, otherwise **p1** is `pointed`.

4.2 Approximations

We define a partial of 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 $S \preceq R$.
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 $k \preceq l$, n some fixed dimension, and **p** either `pointed` or `nonpointed`. Further, let **R** be one of `centered`, `forward`, and `backward`. We then have

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

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, we have*

$$\texttt{forward}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p}) \preceq \texttt{centered}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p})$$
$$\texttt{backward}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p}) \preceq \texttt{centered}(\texttt{dim}{=}n,\texttt{depth}{=}k,\mathbf{p})$$

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*

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

If an array computation conforms to a specification R and $R \preceq S$ then it conforms also to `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 (1) is 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 vectors of sets of integers which we call *index schemes*. As an initial informal example, consider the stencil kernel `y(i) = x(i,0) + x(i+1,0)`. The access pattern on array `x`, relative to induction variable `i`, is captured by a vector containing two integer sets: $\langle\{0,1\},\mathbb{Z}\}\rangle$. This describes that, in the first dimension, the array is read at offsets of 0 and 1 from an induction variable. In the second dimension, the index is unconstrained as they are constants.

Index schemes form 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 sytnax of array subscripts in code (§5.3). We state various results in this section, for which the proofs are provided in Appendix B.

5.1 Lattice model of regions

Definition 5 (Index scheme)**.** An N-dimensional *index scheme* is a vector containing N integer sets, *i.e.*, a member of $\mathcal{P}(\mathbb{Z})^N$. An equivalent view of these vectors is as an N-times Cartesian product on subsets of $\mathbb{Z}$. We use S, T, U to denote index schemes. Henceforth, we assume index schemes are all N-dimensional[1] for some N.

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* $S, T \in \mathcal{P}(\mathbb{Z})^N$

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

Union does not distribute over index schemes, however, a more restricted property holds.

Lemma 2. *If* S *and* T *are index schemes where* $\pi_i(S) = \pi_i(T)$ *for all* $1 \le i \le N$ *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 $a \le 0 \le b$ and c is drawn from $\mathbb{B} = \{\text{true}, \text{false}\}$. Intervals are interpreted as sets as follows:

$$[a \ldots b]^c \triangleq \{n \mid a \le n \le b \wedge (\neg c \implies n \neq 0)\}$$

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 proofs. Here, $-\infty$ and ∞ behave as top and bottom elements to $\mathbb{Z}$ respectively.

We denote the set of all such intervals as *Interval*.

Lemma 3. *We have the following dual identities for* $\mathbb{Z}$ *intervals:*

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

We define two specialisations of index schemes: the *subscript scheme* and the *interval scheme*.

Definition 7 (Interval scheme)**.** An interval scheme is an N-length vector (N-ways Cartesian product) of intervals on $\mathbb{Z}$ (as in Definition 6), denoted by the set $Interval^N$ for a product of N intervals.

Definition 8 (Subscript scheme)**.** A *subscript scheme* is an index scheme scheme where:

$$\forall i.\ 1 \le i \le N \implies \pi_i(S) = \{p\} \ \vee \ \pi_i(S) = [-\infty \ldots \infty]$$

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

Definition 9 (Region)**.** A region is an index scheme and $Region_N$ is the set of all regions (*i.e.*, $Region_N \subseteq \mathbb{Z}^N$). The set of all regions 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. $(Region_N, \cup, \cap, \subseteq)$ *is a bounded distributive lattice with top* $\top = \mathbb{Z}^N$ *and bottom* $\bot = \emptyset$.

[1] The particular dimensionality is derived from array types when the model is used.

$$[\![-]\!]_N : \mathit{region} \to \mathit{Region}_N$$
$$[\![\texttt{r + s}]\!]_N = [\![\texttt{r}]\!]_N \cup [\![\texttt{s}]\!]_N$$
$$[\![\texttt{r * s}]\!]_N = [\![\texttt{r}]\!]_N \cap [\![\texttt{s}]\!]_N$$
$$[\![\texttt{rconst}]\!]_N = \mathit{promote}_N(i, [\![\texttt{rconst}]\!])$$

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

(a) Interpretation of regions

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

$$[\![-]\!]^{\mathsf{m}} : \mathit{mult} \to (A \to \mathsf{Mult}\, A)$$
$$[\![\texttt{readOnce}]\!]^{\mathsf{m}} = \mathsf{once}$$
$$[\![\epsilon]\!]^{\mathsf{m}} = \mathsf{mult}$$

(b) Interpretation of modifiers

Fig. 2. Semantic model of specifications

The set of regions Region_N is the target of our specification language model, and of the code analysis. We further wrap this in a labelled variant which provides information on multiplicity and approximation.

Definition 10. Mult and Approx are parametric labelled variant types with injections given by their definition:

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

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

5.2 Denotational semantics for specifications

An interpretation function $[\![-]\!]_N$ maps closed[2] specifications to sets of N-dimensional index schemes with modifier information, *i.e.* specifications are mapped to $\mathsf{Mult}(\mathsf{Approx}(\mathit{Region}_N))$.

The interpretation is overloaded on *regions* in Figure 2a. Various intermediate notions are used.

Definition 11. Let $\mathit{promote}_N : \mathbb{N}^+ \times \mathit{Interval} \to \mathit{Interval}^N$ 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.

The intermediate interpretation $[\![-]\!]$ of Figure 2a models region constants. This is lifted by $\mathit{promote}_N$ to interpret region constants in the last equation of $[\![-]\!]_N$. The + and * operators are modelled in terms of the join (union) and meet (intersection) of regions. Thus the syntax of regions is modelled by members of Region_N.

We mark in our model the presence of modifiers such as `readOnce` and `atMost`. Approximation modifiers are interpreted as injections into the Approx variant by $[\![-]\!]^{\mathsf{a}}$ in Figure 2b. The Approx type corresponds to the presence or absence of the spatial approximation modifier, with exact when there is no such modifier and lower and upper for `atLeast` and `atMost` respectively. In a similar way, multiplicity modifiers are interpreted as injections in the Mult variant by $[\![-]\!]^{\mathsf{m}}$, corresponding to the presence or absence of the `readOnce` modifier as shown in Figure 2b.

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

Definition 12 (Semantics of specifications)**.** The intermediate interpretations of Section 5.2 are composed to give a model for the top-level specification syntax as:

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

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*

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

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*

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

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.

5.3 Interpreting array subscripts

Definition 13. Recall array subscript terms of the form $a(\bar{e})$ from Definition 2. We interpret these terms with the partial interpretation $[\![-]\!]^{aterm} : \mathit{array\text{-}term} \nrightarrow \mathcal{P}(\mathbb{Z})^N$. The interpretation is defined when all indices are either constant or neighbourhood indices as defined in Definition 4.

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

Note that this produces subscript schemes (a form of index scheme), but these are not members of *Interval* or Region_N. In the next section, we use this interpretation on array subscripts in the static analysis preceding checking and inference.

5.4 Union Normal Form

We lastly address a problem of *representation* for index schemes in the model. Index schemes may incorporate the infinite set $\mathbb{Z}$ in their components, *e.g.* when a dimension is unconstrained or when an absolute index of array term is interpreted. In order to provide a finite checking procedure, we require a finite and compact representation for indices which accounts for this.

Recall that Lemma 1 states that the intersection of index schemes is an index scheme. When combined with Lemma 3 (union and intersection on holed intervals), we can extend this closure to *interval* schemes. Subscripts schemes do not enjoy the same closure property as the intersection of two singleton sets may be empty, disqualifying the result from being a subscript index. In the following sections, we do not take intersection of subscript indices.

Union is not as flexible as intersection; it is not closed over indexing schemes, only on indexing schemes which are equal in all components but one (Lemma 2). For this reason, we represent sets of indexing schemes as unevaluated union terms, or *union normal form*. Since the model forms a distributive lattice as established in Proposition 3, intersections can be pushed inwards using the distributive law leaving only unions at the outer-level. We further exploit associativity of union to put unions into a cons-tree effectively representing union normal form as a non-empty list. We do not, however, attempt to canonicalise the representation. However, at the end of the inference procedure, Lemma 2 and index scheme subset are used to reduce the size of union normal form as a way of simplifying specifications.

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 arbitrarily more complex and wide-ranging independent of the checking and inference procedures. At the moment, the analysis is largely *syntactic*, with only a small amount of semantic interpretation of the code.

Example 6.1. We demonstrate analysis, checking, and inference on 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
```

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)** (interprocedural) data-flow analysis, providing a *flows to* graph (reaching definitions); and **(4)** type information per variable. The analysis traverses the control-flow graph of a program top-down and traverses statements inside of loops bottom-up. 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). These are then converted into our model domain by the interpretation $[\![-]\!]^{aterm}$ (Definition 13) and grouped into a finite map from array variables to set of subscript schemes. We denote sets of index/subscripts schemes by $\mathcal{S}$.

For our example, the set of subscript schemes from the analysis is:

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

This set of index 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 $\mathsf{Mult}(\mathcal{P}(\mathbb{Z})^N)$.

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

Checking verifies the access pattern of an array computation in the source language against any associated specifications. Checking proceeds by generating a model from a specification and generating a model from the source code (above), and comparing them for consistency. Since the model of Section 5 interprets both array indices and specifications as sets of points in N-dimensional space, the notion of consistency is then intuitively whether these two (potentially infinite) sets of points are equal. This leads to a simple notion of consistency $\mathit{consistent}(M_C, M_S)$ which tests the consistency of a model M_C of source code against a model M_S of a specification, where $\mathit{consistent} : \mathsf{Mult}(\mathcal{P}(\mathbb{Z}^N)) \times \mathsf{Mult}(\mathsf{Approx}(\mathit{Region}_N)) \to \mathbb{B}$ is defined:

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

with the intermediate $\mathit{consistent}' : \mathcal{P}(\mathbb{Z}^N) \times \mathsf{Approx}(\mathit{Region}_N) \to \mathbb{B}$ defined as:

$$consistent'(ixs, model) = \begin{cases} m = ixs & model = \text{exact}(m) \\ m \supseteq ixs & model = \text{upper}(m) \\ m \subseteq ixs & model = \text{lower}(m) \end{cases}$$

The *consistent* function checks whether linearity of the specification matches that of the indices, *i.e.* if the specification allows indices to be repeated or not. It 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, we expect the space defined by observed indices corresponds exactly with those defined by the region in the specification, hence requiring 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 region into interval constraints and subscript schemes into membership constraints and pass these to the Z3 SMT solver (De Moura and Bjørner 2008) to see if they are equal. The query is expressed in quantifier-free linear arithmetic, which is decidable.

Although satisfiability is super-exponential in the length of the formula in the worst case (Fischer et al. 1974), the performance of consistency checking using satisfiability is fast in practice as the length of the formula is linearly related with two factors: (1) dimensionality times the number of regions composed with + and (2) dimensionality times the number of array terms flowing into an assignment. In Section 2.3, we established that 99.6% of array computations have dimensionality at most four and 97% of array computations involve at most 4 array subscript terms. In Section 7, we show that all specifications in the corpus comprises no more than two + operations. Thus, our approach using Z3 is practical and efficient in all but corner cases.

6.3 Inferring specification automatically

We provide an inference procedure for generating specifications from code which are 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 sets of subscript schemes. Inference then has two parts. First, "adjacent" index schemes are coalesced into a smaller sets of index schemes, remaining 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) intervals is calculated by coalescing *adjacent* index schemes until a fixed-point is reached.

Definition 14 (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 $\pi_k(T) = [b + 1, c]$.

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 S (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 $coalesceStep(\mathcal{S})$ is computed to give a covering over the initial subscript space.

Lemma 4. *For a set of index schemes $\mathcal{S}$, then $coalesceSet(\mathcal{S})$ is a set of index schemes in union normal form.*

For our example, the fixed-pointed of *coalesceStep* is reached within two steps:

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

6.3.2 Index schemes to syntax. Let the covering indexing scheme from the fixed point of *coalesceStep* on input $\mathcal{S}_0$ be written as $\mathcal{S}_\omega$. Next, $\mathcal{S}_\omega$ is translated into specification syntax. This happens in three 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) Interval schemes are mapped into a joins of meets of region constants, where some intervals split into multiple separate region constants.

An index scheme is an interval scheme (Definition 7) if each vector can be represented as a vector of holed intervals $[a \ldots b]^c$. Since index schemes of the previous stage are all adjacent and form closed intervals (in the usual mathematical sense) then an interval with a lower bound ≤ 0 and an upper bound ≥ 0 is sufficient to form an interval scheme. 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 dimension 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 do not produce 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.

Translation from interval schemes. Recall the model of region constants in Figure 2a 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 into the relevant form, 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:

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

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

The final stage of inference thus maps the component of every interval scheme to a region constant via $[\![-]\!]^{-1}$, combining each component with * can combining the resulting regions by +, that is:

$$convert_N(\mathcal{S}) = \sum_{S \in \mathcal{S}} \prod_{\pi_i(S) \neq [-\infty \ldots \infty]} [\![\pi_i(S)]\!]^{-1} \quad (1)$$

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, ctd(dim=1,depth=1)*ptd(dim=2) + ptd(dim=1)*ctd(dim=2,depth=1)
```

Lemma 5 (Inference soundness). *For all region specs R, then infer*$([\![R]\!]_N)_N \equiv R$

7 EVALUATION

To study the effectiveness of our approach, we applied our tool back onto the corpus from 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. We did not expect to infer specifications for each array computation we found because our analysis restricts the array-subscript-statements, along the lines of Section 2. Overall we ended up with 80,731 specifications of which 3,613 were with a left-hand side variable rather than a subscript (an `access` specification), leaving 77,118 `stencil` specifications. 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:

All pointed 73,340 of the specifications we generated involved only `pointed` regions. Common examples of this were pointwise transformations on data (such as scaling). The remaining 7,391 we consider to be non-trivial, *i.e.* with a higher potential for programming error.

Single-action specifications comprise one forward, backward, or centered region constant combined via + or * with any number of `pointed` regions. We identified 6,442 single-action specifications, of which 4,732 were single-action with a `nonpointed` modifier.

Multi-action specifications comprise at least two forward, backward, or centered regions, combined with any number of `pointed` regions. We identified 885 multi-action specifications out of which 327 had regions combined only with * and 558 combined with a mix of * and +.

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

Occurrences:	0	1	2	3	4	5	6
*	37,714	22,100	18,385	2,073	357		100
+	80,143	468	120				

Bounded specifications occured with 238 `atMost` bounds and 22 `atLeast`, the latter of which were always also paired with an upper bound.

ReadOnce specification occurred 71,726 times, which correlates with the high proportion of unique-subscript expressions in the initial study.

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 could not analyse or classify;

- **Inconsistent LHS/RHS** occur when 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)`.

7.2 Detecting errors in the 2-D Jacobi iteration

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

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

We infer a precise specification of its shape as:

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

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

7.3 Verification case studies

7.3.1 Catching bugs. Detecting whether an error could have been prevented by a stencil specification is difficult because it requires knowledge of the programmer's intent. However, this is occasionally captured in commit messages accompanying a code change. For example in the Unified Model we identified an example of an off-by-1 error that could have been avoided through the use of stencil specification. The buggy code was identified as `pointed` in all dimensions by our inference:

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

But the programmer made it clear in the comments after the bug fix that the intention had been `forward` in the third dimension, *i.e.*, a specification:

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

We found a similar bug in Cliffs[3] where a subscript `u(i)` is corrected to `u(i-1)` which could have been detected by an initial specification of `!= stencil backward(depth=1, dim=1, nonpointed)`.

7.3.2 Aiding refactoring. One of the initial goals of CamFort was to provide tools that enable optimisation and refactoring without changing behaviour, and the array specification feature was designed with that in mind. But while we were perusing real-world source bases, 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 logs of revision control for our corpus 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 its stencil 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 the specifications can have them updated quickly and then any code that is not updated will trigger an error when checked.

An example of a refactored region adding a single dimension to an array, adapted from the UM:

```
! before
!= region :: r = readOnce, pointed(dim=1)
```

[3]https://github.com/Delta-function/cliffs-src/commit/e2ca312cfc5398287c7ca8594e977658617c2540

```
! after
!= region :: r = readOnce, pointed(dim=1)*pointed(dim=2)
```

8 RELATED WORK AND CONCLUSIONS

Various deductive verification tools can express array indexing in their specifications, *e.g.*, ACSL of Baudin et al. (2008) for C (*e.g.* Burghardt et al. (2010, Example 3.4.1)). A specification can be given for a stencil computation but must use fine-grained indexing as in code and therefore is similarly prone to indexing errors. Our approach is much more abstract—it does not aim to reify indexing in the specification, but provides simple spatial descriptions which capture a large number of common patterns.

Kamil et al. (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 (3.2Ghz 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 arbitrarily more sophisticated independent of the rest of the work. Furthermore, Kamil *et al.* do not provide a user-visible syntactic representation of their specifications, and nor do they provide verification from specifications *e.g.*, to future-proof the code against later changes. Even if they were to provide a syntactic representation, for complex stencils such as Navier-Stokes from 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 efforts to verify kernels written for General-Purpose GPU programming, such as in Blom et al. (2014). However, their focus is mainly on the synchronisation of kernels and the avoidance of data races., while we are interested in correctness Solar-Lezama et al. (2007) give specifications of stencils using unoptimised "reference" stencils, coupled with partial implementations which are completed by a code generation tool. The primary purpose of this tool is optimisation rather than correctness, and the language of specification is more elaborate than ours.

Tang et al. (2011) define a specification language for writing stencils embedded in C++ (with Cilk (Blumofe et al. 1996) extensions) that are then compiled into parallel programs based on trapezoidal decompositions with hyperspace cuts. Pochoir specifications are used for describing the kernel, boundary conditions, and shape of the stencil. Pochoir is aimed at programmers reluctant to implement the high-performance cache-oblivious "hypertrapezoidal" algorithms. Like much of the related work, the goal is optimisation rather than correctness.

By contrast, the work of Abe et al. (2013) studies correctness bringing a form of model-checking to verify certain stencil computations in the context of parallelism in partitioned global address space languages. Abe *et al.* provide a new language for writing stencil computations. Much of the specification effort goes towards describing the distribution of the computation over multiple processors. The code for the stencil kernel is generated from a relatively high-level specification. In contrast, we integrate directly into existing, legacy codebases and established languages, bringing the benefits of verification more easily to scientific computing.

Concluding remarks We proposed a novel solution to a common programming task in numerical and 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 80,000 array computations across a million lines of Fortran with simple, short, abstract specifications. We are now in the process of applying

our tool in collaboration with the authors' of some of our corpus packages. Future work is to expand the range of our program analysis, including options for further semantic analysis and capturing a wider set of patterns.

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 Mihelffhifj. 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 Constrastin, Matthew Danish, Dominic Orchard, and Andrew Rice. 2017. CamFort - refactoring, analysis, and verification tool for scientific Fortran programs. https://github.com/camfort/camfort. (2017). Accessed: 15 February 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.

Michael Jo Fischer, Michael J Fischer, and Michael O Rabin. 1974. Super-exponential complexity of Presburger arithmetic. (1974).

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

Nicholas J Giordano and Hisao Nakanishi. 1997. *Computational physics.* Prentice Hall Upper Saddle River.

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.

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

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

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

Package	Number of files Total	Parsed	Lines of physical code Total	Parsed	Lines of raw code Total	Parsed
ARPACK-NG	312	290	50,208	47,453	144,081	139,290
BLAS	151	149	16,046	15,993	40,882	40,679
Cliffs	30	30	2,424	2,424	3,149	3,149
CP	52	48	2,334	2,121	3,978	3,632
E3ME	167	154	44,935	39,700	73,545	62,238
GEOS-Chem	604	336	449,222	269,757	856,463	410,654
Hybrid4	29	29	4,831	4,831	8,361	8,361
mudpack	88	88	54,753	54,753	78,652	78,652
Navier	6	6	505	505	696	696
Specfem3D	555	475	137,468	103,328	232,356	178,317
UM	2,540	2,269	635,525	541,540	1,010,936	866,406
Total	4,534	3,874	1,398,251	1,082,405	2,453,099	1,792,074

Table 1. Summary of software packages used for evaluation

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.

A DETAILS OF THE CORPUS DATA SET

Software Corpus. Table 1 shows summary statistics of the software packages used in our evaluation, all of which are written in Fortran 90 or Fortran 77. In total we analysed 1,386,758 lines of code from 11 packages, of which we successfully parsed 1,086,321 lines. The "Number of files" column shows how many files in each corpus that we were able to analyse with CamFort. The most common reasons for CamFort rejecting a file were either use of a C-preprocessor, or illegal use of language features from a modern Fortran variant.

(1) **The Unified Model** (Wilson and Ballard 1999) is a weather forecasting and climate modelling tool developed by the Met Office in the United Kingdom. It is used by research organisations and meteorological services around the world. We use the development branch (trunk) of the model. The code base in closed source but institutional licenses are available for research purposes. The Met Office runs a comprehensive code quality system incorporating dedicated committers (we counted 11) for particular parts of the model. We counted 120 additional contributors whose submissions are reviewed and tested before being accepted into the code base.

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

(3) **BLAS** (Blackford et al. 2002) (Basic Linear Algebra Subprograms) is a popular library providing efficient and portable routines for vector and matrix operations. These routines feature in many other libraries (including LAPACK). We used version 3.6.0. We chose to include this package for breadth, as it provides general numerical functions rather than a specialised scientific model.
(4) **Hybrid4** is a vegetation and biomass model for simulating carbon, water and nitrogen flows (Friend and White 2000).
(5) **GEOS-Chem** (Bey et al. 2001) is a three-dimensional model of tropospheric chemistry developed at Harvard and used by ~70 universities and research institutions world-wide. We use v.10-01.
(6) **Navier** is a small numerical simulation, giving a discrete approximation to the two-dimensional Navier-Stokes fluid equations, based on the book of Griebel et al. (1997).
(7) **CP** consists of the example code from the second edition of the book "Computational Physics" (Giordano and Nakanishi 1997) introducing numerical techniques and their application to modern physics problems such as fields, waves, statistical mechanics and quantum mechanics.
(8) **ARPACK-NG** is an open-source library for solving large-scale eigenvalue problems (Sorenson et al. 2017).
(9) **SPECFEM3D** is a global seismic wave models (Komatitsch et al. 2016).
(10) **MUDPACK** is a general multi-grid solver for elliptical partial differentials (Adams 1991);
(11) **Cliffs** is a parallelised tsunami model (Tolkova 2014)

B PROOFS AND EXTENDED DEFINITIONS FOR THE MODEL

Lemma 1. *Intersection distributes over index schemes. That is, for index schemes* $S, T \in \mathcal{P}(\mathbb{Z})^N$

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

Proof.

$$\begin{aligned} S \cap T &= \left\{ x \;\middle|\; \bigwedge_{1 \le i \le N} x_i \in \pi_i(S) \right\} \bigcap \left\{ x \;\middle|\; \bigwedge_{1 \le i \le N} x_i \in \pi_i(T) \right\} \\ &= \left\{ x \;\middle|\; \bigwedge_{1 \le i \le N} (x_i \in \pi_i(S) \wedge x_i \in \pi_i(T)) \right\} \\ &= \prod_{i=1}^{N} \pi_i(S) \cap \pi_i(T) \end{aligned}$$

□

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

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

Proof.

$$
\begin{aligned}
S \cup T &= \left\{ x \;\middle|\; \bigwedge_{1 \le i \le N} x_i \in \pi_i(S) \right\} \bigcup \left\{ x \;\middle|\; \bigwedge_{1 \le i \le N} x_i \in \pi_i(T) \right\} \\
&= \left\{ x \;\middle|\; \bigwedge_{\substack{1 \le i \le N \\ i \neq k}} x_i \in \pi_i(S) \wedge x_k \in \pi_k(S) \vee \bigwedge_{\substack{1 \le i \le N \\ i \neq k}} x_i \in \pi_i(T) \wedge x_k \in \pi_k(T) \right\} \\
&= \left\{ x \;\middle|\; \bigwedge_{\substack{1 \le i \le N \\ i \neq k}} x_i \in \pi_i(S) \wedge x_k \in \pi_k(S) \cup \pi_k(T) \right\} \\
&= \pi_1(S) \times \cdots \times (\pi_k(S) \cup \pi_k(T)) \times \cdots \times \pi_N(S)
\end{aligned}
$$

□

Lemma 3. *We have the following dual identities for $\mathbb{Z}$ intervals:*

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

Proof. We give the proof of the first identity and the second one is similar.

$$
\begin{aligned}
[a \ldots b]^c \cap [d \ldots e]^f &= \left\{ n \;\middle|\; a \le n \le b \wedge (\neg c \implies n \neq 0) \right\} \cap \\
&\quad \left\{ n \;\middle|\; d \le n \le e \wedge (\neg f \implies n \neq 0) \right\} \\
&= \left\{ n \;\middle|\; \max\{a, d\} \le n \le \min\{b, e\} \wedge (\neg c \vee \neg f \implies n \neq 0) \right\} \\
&= [\max\{a, d\} \ldots \min\{b, e\}]^{c \wedge f}
\end{aligned}
$$

□

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

Proof. Straightforward, the join and meet are mapped to $\cup$ and $\cap$, the set is inductively designed to be closed under these operations. Union and intersection are associative, commutative, and absorptive under closed sets and then they are also for $Region_N$. This is enough to show that, it is a lattice.

Further, we have $\mathbb{Z}^N \cap R = R$ and $\emptyset \cup R = R$ with $\mathbb{Z}^N$ and $\emptyset$ belonging to $Region_N$. This makes the lattice a bounded one.

Finally, the lattice is distributive since union distributes over intersection and vice versa when the set is closed under these operations. □

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

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

Proof. Proof of the first inequality:

$$
\begin{aligned}
\texttt{fwd(dim=}n\texttt{,depth=}k\texttt{,p)} &\preceq \texttt{fwd(dim=}n\texttt{,depth=}k\texttt{,p)} + \texttt{bwd(dim=}n\texttt{,depth=}k\texttt{,p)} && \textbf{(Combined)} \\
&\equiv \texttt{ctd(dim=}n\texttt{,depth=}k\texttt{,p)} && \textbf{(Centered)}
\end{aligned}
$$

Proof of the second inequality:

$$\begin{aligned} \texttt{bwd(dim=}n\texttt{,depth=}k\texttt{,p)} &\preceq \texttt{fwd(dim=}n\texttt{,depth=}k\texttt{,p) + bwd(dim=}n\texttt{,depth=}k\texttt{,p)} && \textbf{(Combined)} \\ &\equiv \texttt{ctd(dim=}n\texttt{,depth=}k\texttt{,p)} && \textbf{(Centered)} \end{aligned}$$

□

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*

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

Proof. Proof of the first inequality:

$$\begin{aligned} \texttt{ptd(dim=}n\texttt{)} &\preceq \textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{,nonpointed) + ptd(dim=}n\texttt{)} && \textbf{(Combined)} \\ &\equiv \textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{)} && \textbf{(Overlapping pointed)} \end{aligned}$$

Proof of the second inequality:

$$\begin{aligned} \textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{,nonpointed)} &\preceq \textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{,nonpointed) + ptd(dim=}n\texttt{)} && \textbf{(Combined)} \\ &\equiv \textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{)} && \textbf{(Overlapping pointed)} \end{aligned}$$

□

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*

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

Proof. Both $\equiv$ and $=$ are equivalence relations. We only need to show that each property associated with specifications equivalence holds as an equality in the model.

Basic & Subsumption These follow from basic properties of lattices.

Dist This is immediate from the definition of distributive lattice which the model is one.

Overlapping pointed Counterparts of individual regions for some Boolean p are given below:

$$\begin{aligned} [\![\textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{,nonpointed)}]\!]_N &= \mathit{promote}_N(n, [-k \ldots k]^{\mathsf{false}}) \\ [\![\texttt{pointed(dim=}n\texttt{)}]\!]_N &= \mathit{promote}_N(n, [0 \ldots 0]^{\mathsf{true}}) \\ [\![\textbf{R}\texttt{(dim=}n\texttt{,depth=}k\texttt{)}]\!]_N &= \mathit{promote}_N(n, [-k \ldots k]^{\mathsf{true}}) \end{aligned}$$

Using Lemma 2, it is sufficient to show $[-k \ldots k]^p \cup [0 \ldots 0]^{\mathsf{true}} = [-k \ldots k]^{\mathsf{true}}$ holds. This immediately follows from Lemma 3.

Centered Let $[\![\textbf{p1}]\!]$, $[\![\textbf{p2}]\!]$, and $[\![\textbf{p3}]\!]$ be $p1$, $p2$, and $p3$ respectively. We know that $p1$ is $p2 \vee p3$ from (**Centered**) definition. Then we get the following interpretation for region constants involved in the equivalence:

$$\begin{aligned} [\![\texttt{centered(dim=}k\texttt{,depth=}n\texttt{,}p1\texttt{)}]\!]_N &= \mathit{promote}_N(n, [-k \ldots k]^{p2 \vee p3}) \\ [\![\texttt{forward(dim=}k\texttt{,depth=}n\texttt{,}p2\texttt{)}]\!]_N &= \mathit{promote}_N(n, [0 \ldots k]^{p2}) \\ [\![\texttt{backward(dim=}k\texttt{,depth=}n\texttt{,}p3\texttt{)}]\!]_N &= \mathit{promote}_N(n, [0 \ldots k]^{p3}) \end{aligned}$$

Using Lemma 2, it is sufficient to show $[-k \ldots k]^{p2 \vee p3} = [0 \ldots k]^{p2} \cup [-k \ldots 0]^{p3}$ holds. This immediately follows from Lemma 3.

□

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*

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

Proof. Both $\preceq$ and $\subseteq$ are partial orders on sets. We only need to show that each inequality holds in the model.

Equivalence Region equivalence is modeled with set equality and the partial order on regions is modeled with subset relation. Partial orders are reflexive and the model of equivalence agrees with that of the partial order, so this rule holds.

Combined `*` and `+` maps to meet and join operations in the lattice. All meets and joins in a lattice satisfy these inequalities.

Depth When m is k and l in `R(dim=`n`,depth=`m`,p)`, we obtain the interpretations $promote_N(n, int_k)$ and $promote_N(n, int_l)$ for some N as interpretations of regions. Call these *ks* and *ls* respectively. One way to show $ks \subseteq ls$ is to show that the intersection of *ks* and *ls* is *ks*.

To show that the intersection is *ks*, we use Lemma 1 which reduces the problem to showing that the intersection of int_k and int_l is the former since at every point except n the intervals agree, hence the intersection remains the same at these dimensions. Then we use Lemma 3 to calculate the intersection. By assumption we have $k \leq l$, so the maximum of the lower bounds is $-k$ and the minimum of lower bounds is k this is same as interval int_k. Note that pointedness is the same for both intervals and depths are positive integers, so the pointedness is preserved under intersection. This shows the intersection of *ls* and *ks* is indeed *ks* and concludes the proof.

□

Lemma 4. *For a set of index schemes $\mathcal{S}$, then coalesceSet($\mathcal{S}$) is a set of index schemes in union normal form.*

Proof. By Lemma 2, the union of two index schemes is an index scheme if all components are equal except at most one component. Contiguity (Definition 14) matches this precondition, thus the first part of the union in *coalesce* produces a set of index schemes. The second part comprises a subset of the original index schemes. □

Lemma 5 (Inference soundness)**.** *For all region specs R, then infer($[\![R]\!]_N$)$_N \equiv R$*

Proof. This follows by induction on the definition of regions syntax *R*. and by definitions of inference and the semantics sharing the same underlying model and using inverse mappings between each other.

For constants, we map to $promote_N(i, [\![\texttt{rconst}]\!])$ which by *convert*, in (1) is mapped back to `rconst` since *convert* is defined in terms of the inverse mapping $[\![-]\!]^{-1}$.

For $[\![r+s]\!]_N = [\![r]\!]_N \vee [\![s]\!]$ we map to a Union Normal Form of interval spaces which is convert back directly by *convert*. Similarly for `*`. □