Verifying Properties of Index Arrays in a Purely-Functional Data-Parallel Language

NIKOLAJ HEY HINNERSKOV, University of Copenhagen, Denmark
ROBERT SCHENCK, Vrije Universiteit Amsterdam, Netherlands
COSMIN E. OANCEA, University of Copenhagen, Denmark

This paper presents a novel approach to automatically verify properties of pure data-parallel programs with non-linear indexing—expressed as pre- and post-conditions on functions. Programs consist of nests of second-order array combinators (e.g., map, scan, and scatter) and loops. The key idea is to represent arrays as index functions: programs are index function transformations over which properties are propagated and inferred. Our framework proves properties on index functions by distilling them into algebraic (in)equalities and discharging them to a Fourier-Motzkin-based solver. The framework is practical and accessible: properties are not restricted to a decidable logic, but instead are carefully selected to express practically useful guarantees that can be automatically reasoned about and inferred. These guarantees extend beyond program correctness and can be exploited by the entire compiler pipeline for optimization. We implement our system in the pure data-parallel language Futhark and demonstrate its practicality on seven applications, reporting an average verification time of 1 second. Two case studies show how eliminating dynamic verifications in GPU programs results in significant speedups.

ACM Reference Format:
Nikolaj Hey Hinnerskov, Robert Schenck, and Cosmin E. Oancea. 2025. Verifying Properties of Index Arrays in a Purely-Functional Data-Parallel Language. 1, 1 (July 2025), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A rich body of work is dedicated to static verification of program properties. This includes (1) theorem provers such as Rocq [5] and Agda [9], which facilitate principled reasoning through their dependent type systems, (2) the systems in the F* family [58, 59], which are commonly aimed at verifying low-level code by, for example, combining dependent type, effect-based reasoning with separation logic, (3) work on Liquid Haskell [49, 63], which is facilitated by reasoning on recursive data types, such as lists, and allows specification of arbitrary user-defined properties, while enabling automated SMT-based reasoning, and (4) systems that implement decidable subsets of array logic [60, 69], but which are restricted to linear indexing. Such systems are commonly aimed at sequential code and either restrict the domain of supported computations (4), or require expert knowledge (e.g., the programmer must explicitly compose the proofs checked by the system).

This paper presents compiler analyses for inferring and verifying properties of *integral arrays*—e.g., ranges, monotonicity, injectivity, bijectivity, filtering, partitioning—applicable to *functional data-parallel languages*, such as Futhark [31, 44, 53], Accelerate [12, 19, 61], Lift [22, 55, 56], DaCe [3, 4, 70].

Authors' addresses: Nikolaj Hey Hinnerskov, nhin@912134.xyz, University of Copenhagen, Denmark; Robert Schenck, r@bert.lol, Vrije Universiteit Amsterdam, Netherlands; Cosmin E. Oancea, cosmin.oancea@di.ku.dk, University of Copenhagen, Denmark.

XXXX-XXXX/2025/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Specific to this domain, the computation is *separated* (fissed) into bulk-parallel array operations—such as map, scan (prefix sum), and scatter (irregular write)—called *second-order array combinators*, together with loops. Eventually, index arrays computed in this way are serve as the indirect indices of irregular read (gather) or write (scatter) operations. Index arrays are analytically convenient because (1) they are typically manipulated in simpler ways than general computation, and (2) they directly inform properties of general arrays, such as injectivity, filtering, and partitioning.

The programming style is both a challenge and an opportunity: On the one hand, the natural expression of sequential computation is in fused form, (e.g., folds), which facilitates easier tracking of the target property at each step of the way. On the other hand, language purity and the semantics of second-order combinators allow to lift the level of abstraction at which the compiler reasons.

Our solution is motivated by the evolution of scheduling languages [23, 48], which demonstrates that the coupling of language specalization (and its compiler repertoire) with human expertise has been essential to unlocking high performance. In the same spirit, our system supports (automates) a predefined, small but powerful set of properties, which are easy to understand and use by the *domain expert* (non-expert programmer). This allows the compiler to exploit the algebra of properties to derive new properties at a high level, and also to use the proven properties to further optimize the program. Possibilities here include static verification of bounds checking and safety of scatter, which we demonstrate (Section 6) to have high impact on GPU execution. On the other hand, our system does not allow the user to specify new properties, and is intended to be neither satisfiable nor decidable—we aim at practical compilation time without restricting the language.

Our framework is aimed at verifying Futhark programs and is implemented as a compiler pass that is structured into three logical components: (1) an analysis InfIxf, presented in Section 4, that infers an index function representation of arrays, which uses guarded expressions (polynomials defined by cases) to represent the values at each array index and supports jagged arrays whose segments may be empty, and (2) a property manager PM (Section 3), that verifies properties of index functions by breaking them into a sufficient set of low-level queries, which are sent to (3) the query solver QS (Section 5), which uses a Fourier-Motzkin [21, 66] adaptation algorithm that relies on an algebra of simplifications aimed at sums of array slices.

These components are connected by means of the supported array properties, e.g., the inference rule of scatter uses monotonicity and bijectivity properties to produce meaningful index functions that enable expression of jagged arrays and derivation of filtering/partitioning properties. The property manager can derive properties at a high level, in the absence of an index function, e.g., a filtering of a monotonic or injective array remains monotonic or injective. Finally, the query solver uses range, injectivity and monotonicity properties, and answers the queries of InfIxf and PM.

Section 6 presents an evaluation of verifying properties on seven challenging data-parallel applications that use non-linear indexing, including the maximal matching graph algorithm, three-way partitioning, segmented filtering and segmented two-way partitioning.[1]

The principle contributions of this work are:

(1) To our knowledge, this is the first solution addressing verification of array properties—including monotonicity, bijectivity, filtering, partitioning—in a purely-functional data-parallel context with non-linear indexing (produced by scatter, gather, scan).
(2) An architectural design that supports verification of a predefined set of properties that are accessible to non-expert programmers, and reveal to the compiler a compositional algebra that facilitates high-level reasoning, automation of inference and further code optimizations.
(3) An evaluation that reports successful verification of seven challenging benchmarks that include graph algorithms and flattened irregular parallelism (segmented filter/partitioning).

[1]Segmented two-way partitioning is the flat-parallel code [7] that filters/partitions each subarray of an (irregular) jagged array in flat form, according to predicates that depend on the segment number.

x, F	::=	$\ldots$	Variables	e^0	::=	v	Var. / Const.
τ^b	::=	`i64` | `f64` | `bool` | $\ldots$	Base Types		|	$v_1\ op\ v_2$	Operation
τ^a	::=	$\tau^b \mid [e^*]\tau^b$	Array Type		|	$x_{array}[x_{index}]$	Array index
τ	::=	$(.., \tau^a, ..)$	Tuple Type		|	`iota` x	$[0, \ldots, x-1]$
Pre	::=	e^*	Pre Cond.		|	replicate $x_n\ x$	$[x, \ldots, x]$
Pst	::=	$\lambda x.\ e^*$	Post Cond.		|	`if` x `then` e^* `else` e^*	Conditional
Fun	::=	`def` $F\ \overline{(x : \tau\ [\mid\mid Pre])}$ $: \tau\ [\mid\mid Pst] = e^*$	Function Def.		|	`loop` $(\overline{x_{var}\ [: \tau \mid Pre]}^q = \overline{x_{init}}^q)$ `while` x_c `do` e^*_{body}	While Loop
					|	`map` $(\lambda\overline{x_{elm}}^q.\ e^*)\ \overline{x_{array}}^q$	Map
					|	`scan` $(\lambda\overline{x_1}^q\ \overline{x_2}^q.\ e^*)\ \overline{k}^q\ \overline{x_{array}}^q$	Scan
op	::=	$+ \mid - \mid *$	Binary Op.		|	`scatter` $x_{dest}\ x_{inds}\ x_{vals}$	Scatter
	|	$> \mid \geq \mid < \mid \leq \mid = \mid \neq$			|	$F\ \overline{x}$	Apply Fun.
v	::=	$x \mid k$	Var./Const.	e^*	::=	`let` $(.., x, ..) = e^0$ `in` e^* | $(.., x, ..)$	Bindings

Notation: $\overline{o}^q$ denotes a sequence of q objects of some kind, separated by white space or by comma, as dictated by the context.

Fig. 1. Grammar for source language expressions (e^*) and function declarations (Fun).

2 LANGUAGES, ARRAY PROPERTIES AND BIRD'S EYE VIEW OF MAIN COMPONENTS

We describe our system as a static analysis on user-written source code. Accordingly, we begin by introducing the source language and two running examples in Section 2.1. Section 2.2 presents an internal language used by our analysis and the bird's eye view of the architecture. Finally, Section 2.3 specifies the array properties supported by our system.

2.1 The Source Language of the Analysis and Running Examples

The source language (e^*), with syntax shown in Fig. 1, is a standard purely functional, first-order expression language, which has been augmented with second-order array combinators, such as `map` and `scan`, and enforces a structure-of-arrays (SoA) layout. For presentation purposes, the language is in A-normal form [51], meaning (1) let bindings can be seen as a block of statements followed by a sequence of result variables, and (2) if conditions, loop initializers and function operands are variables. The types conform with the SoA layout, for example, $([n]\texttt{i64},\ [n]\texttt{bool})$ is a valid tuple type of integer and boolean arrays, both of length n, but the type $[n](\texttt{i64},\ \texttt{bool})$ is invalid, because it uses an array-of-structures (AoS) layout.

Function declaration allows annotating argument types with preconditions (boolean expressions) and the result type with a postcondition (lambda function from the result type to booleans).

The language has array constructors `iota` n, which produces the array $[0, \ldots, n-1]$, and `replicate` $n\ x$, which produces a length-n array containing x at each index. As well as if and while loops, which are equivalent to tail-recursive functions: the q loop parameters $\overline{x_{var}}^q$ are initialized with the values of variables $\overline{x_{init}}^q$ and are bound to the result of the loop-body expression e^*_{body} for the remaining iterations. Map and inclusive scan (prefix sum) have standard types and semantics:

$$\texttt{map} : (\alpha \to \beta) \to [n]\alpha \to [n]\beta \qquad \texttt{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$
$$\texttt{map}\ f\ [x_0, \ldots, x_{n-1}] = [f\ x_0, \ldots, f\ x_{n-1}] \qquad \texttt{scan}\ \odot\ ne_\odot\ [x_0, \ldots, x_{n-1}] = [x_0, \ldots, x_0 \odot \ldots \odot x_{n-1}]$$

But they adhere to the SoA layout: their lambda functions determine an arbitrary number of results, and they accept a matching number of arguments; $\odot$ must be associative with neutral element $ne_\odot$.

Finally, `scatter` is a bulk-write operator of type $[n]\alpha \to [m]i64 \to [m]\alpha \to [n]\alpha$ and imperative semantics: $\texttt{scatter}\ dst\ is\ vs\ \equiv\ \texttt{for}\ i = 0 \ldots m-1\ \{\ \texttt{if}\ (0 \leq is[i] < m)\ dst[is[i]] := vs[i]\ \}$. That is, `scatter` updates dst in place at indices is with the corresponding values from vs, but ignores updates to indices that are out of bounds in dst. Its pure semantics and $O(m)$ work asymptotic are ensured by a type checking technique that builds on uniqueness types [31]. To match the imperative, deterministic semantics, `scatter` must be idempotent. We can ensure this by requiring any duplicate indices in is to correspond to equal values in vs:

```
1  let sum [n] (xs: [n]i64) =
2    if n > 0 then (scan (+) 0 xs)[n-1] else 0
3
4  def partition2 [n] (p: f64 -> bool) (xs: [n]f64)
5      : (i64, [n]f64) | \ (m,ys) ->
6          m == sum ( map i64.bool (map p xs) )
7          && FiltPart ys xs (\_->true) (\i-> p xs[i]) =
8    let cs = map (\x -> p x) xs
9    let flagsT = map (\c -> if c then 1 else 0) cs
10   let flagsF = map (\b -> 1 - b) flagsT
11   let indicesT = scan (+) 0 flagsT
12   let num_true = if n > 0 then indicesT[n-1] else 0
13   let tmp = scan (+) 0 flagsF
14   let indicesF = map (\t -> t + num_true) tmp
15   let indices = map3 (\c t f -> if c then t-1 else f-1)
16                      cs indicesT indicesF
17   let zeros = replicate n 0
18   let ys = scatter zeros indices xs
19   in (num_true, ys)
```

```
⟸ Demo
p = \ x -> x < 5,
xs= [5,4,2,8,7,3]
n = 6
[F, T, T, F, F,T]
[0, 1, 1, 0, 0, 1]
[1, 0, 0, 1, 1, 0]
[0, 1, 2, 2, 2, 3]
3
[1, 1, 1, 2, 3, 3]
[4, 4, 4, 5, 6, 6]
[3, 0, 1, 4, 5, 2]

[0, 0, 0, 0, 0, 0]
[4, 2, 3, 5, 8, 7]
```

```
1  def sgmSum [n] (flags: [n]bool) (xs: [n]i64) : [n]i64 =
2    let (_, ys) = scan (\ f1 v1 f2 v2 -> let f = fx || fy
3                                         let v = if fy then vy else vy + vx
4                                         in (f, v) ) false 0i64 flags xs in ys
5  def mkSgmDescr [m] (shape: [m]i64 | Range shape (0,∞))
6      (xs: [m]i64) : []t | (\flags -> length flags == sum shape) =
7    let rot = map (\i -> if i==0 then 0 else shape[i-1]) (iota m)
8    let scn = scan (+) 0i64 shp_rot
9    let ind = map2 (\s i -> if s<=0 then -1 else i) shape shp_scn
10   let len = if m > 0 then shp_scn[m-1] + shape[m-1] else 0
11   let res = scatter (replicate len 0) ind xs in res
12   -- Example: shape=[0,2,1,0,3] & xs=[1,2,3,4,5] ⇒ res=[2,0,3,5,0,0]
13 def mkII [m] (shape: [m]i64 | Range shape (0,∞))
14       : []i64 | (\II -> length II == sum shape) =
15   let beg_vs = map (\i -> i + 1) (iota m)        -- [1,2,3,4,5]
16   let sct_vs1= mkSgmDescr shape beg_vs           -- [2,0, 3, 5,0,0]
17   let sct_vs = map (\v -> if v == 0 then 0 else v-1) scat_vs1
18   let flags = map (\f -> f > 0) scat_vs          -- [T,F, T, T,F,F]
19   let II = sgmSum flags scat_vs in II            -- [1,1, 2, 4,4,4]
```

Fig. 2. Running Examples: two-way partitioning (left) with demo (center) & building flag and II arrays (right).

$$0 \leq i_1 < m \wedge 0 \leq i_2 < m \wedge 0 \leq is[i_1] < n \wedge 0 \leq is[i_2] < n \wedge is[i_1] = is[i_2] \Rightarrow vs[i_1] = vs[i_2].$$

However, this is not verified in Futhark, neither statically nor dynamically; our work enables static verification of scatter and of array indexing (mostly verified by dynamic assertions), among others.

Our implementation uses Futhark's source language [31, 32], which is neither in A-normal form nor uses an SoA layout. For clarity, we elide the details of the transformation into this form, which is automatically performed in later compilation stages [26]. In particular, we support passing predicates as arguments to functions to enable general expression of filtering/partitioning properties.

Futhark has support for size-dependent types [2, 28], but size equality is purely syntactical: $[n+m]\tau \neq [m+n]\tau$. Our work naturally extends the expressible size dependencies using pre- and postconditions. The modifications needed to accommodate pre- and postconditions are standard: preconditions are verified and postconditions are assumed at call sites, and vice versa for function declarations. A property assigned to a loop parameter is verified on the loop initializer; then the property is assumed on the parameter and is verified on the corresponding loop-body result.

Running Examples. Figure 2 shows two code examples that are illustrative for the functional data-parallel programming style, where the computation is separated (fissed) into a sequence of bulk-parallel operations. Many such operations manipulate integral arrays that are eventually used for indirect indexing in gather and scatter operations. For example, the code on the left implements a two-way partitioning of an array xs based on a predicate p: the predicate is first mapped across the elements of xs and the integral result ($flagsT$) is scanned, resulting in array $indicesT$ that holds the indices at which the elements that succeed should be scattered plus one. The failing indices are treated similarly, resulting in $indicesF$. The final indices are put together by the `map` operation at line 15, and, finally, the partitioned arrays is computed by the scatter at line 15. This makes it more challenging to verify the partitioning property than in sequential languages, where the computation is, e.g., aggressively fused into a fold, whose accumulator maintains separate lists of succeeding and failing elements, thus allowing to track the property across each statement. The post conditions of `partition2` are that the split point m is equal to the number of elements that succeed under p and that the result is a partitioning of xs (as detailed in Section 2.3).

The right hand side of Fig. 2 shows helper functions that are used in flattened/segmented computations: *mkSgmDescr* takes as arguments the shape of a jagged array—which is allowed to have empty segments (the precondition only requires non-negative elements)—and an array

c	::=	x	Variable	t	::=	$c \mid c \cdot t$	Term	**Notation:**
	|	n	Integer	e	::=	$t \mid t + e$	Polynomial	i, j, k integral iterators
	|	$x[e]$	Indexing	g	::=	$c \Rightarrow e \mid g \bigwedge g$	Guarded expr.	v, w enumeration bounds
	|	$\sum_{x=e}^{e}(c)$	Sum	D	::=	$\text{for } x < e \mid \bigcup_{k=0}^{e} .x \geq e$	Lin/Sgm dom.	h, l ct-range iterators
	|	x^{-1}	$x[x^{-1}[i]] = i$					x, y, z array variables
	|	∞	Special: $\infty \notin \mathbb{Z}$	$ixfn$	::=	$D.\, g$	Index function	$e\{x \mapsto e_x\}$ substitutes x for e_x in e
	|	$\circlearrowleft$	Recurrence		|	$(\text{for } x < e.g) \cup (\bigcup_{k=1}^{e} .x \geq e \propto e.\, g)$		$\bigwedge_{h=1}^{v}(c_h \Rightarrow e_h)$ denotes
	|	$e \odot e$	Comparison	$\odot$	::=	$< \mid \leq \mid > \mid \geq \mid = \mid \neq$		$c_1 \Rightarrow e_1 \bigwedge \ldots \bigwedge c_v \Rightarrow e_v$
	|	$\neg c$	Logicals					
	|	$c \wedge c \mid c \vee c$						

Fig. 3. Grammar for symbols (c), polynomials (e), guarded expressions (g), and index functions ($ixfn$).

xs of matching length. It returns a flat array of length equal to the sum of the shapes (see the postcondition), such that each non-empty segment starts with the corresponding element of xs and the rest of the segment is zeroed. For example, if $shape = [0, 2, 1, 0, 3]$ and $xs = [1, 2, 3, 4, 5]$ then $mkSgmDescr\ shape\ xs = [2, 0, 3, 5, 0, 0]$. Note that empty segments pass negative indices (line 9) to scatter, which are ignored (otherwise WAW races would violate its deterministic semantics).

Similarly, *mkII* in Fig. 2 takes as argument a shape array and produces a flat array of that shape in which each element is assigned the index of the segment in which it resides. For example, if $shape = [0, 2, 1, 0, 3]$ then $mkII\ shape = [1, 1, 2, 4, 4, 4]$. We will name this array II henceforth. The implementation uses *mkSgmDescr* (line 16) to inscribe the index of each non-empty segment at the start of the segment (line 17), i.e., $[1, 0, 2, 4, 0, 0]$, and then propagates the start element throughout the segment by using a segmented scan [6] (line 19), which is implemented as a scan with a lifted operator (line 1). Note that there are no standard properties that accurately summarize the results of *mkII* and *mkSgmDescr*, albeit their content can be easily described as index functions.

2.2 Index-Function Representation and Bird's Eye View of Architecture

Figure 3 presents the grammar for the internal languages. A symbol (c) can be a variable name, an integer constant, array indexing, a sum of symbols, an inverse array (i.e., $x^{-1}[x[i]] = x[x^{-1}[i]] = i$; see Section 4), a special ∞ symbol used for pattern matching, a recurrence (needed transiently to represent scans—see Section 4), a comparison, or a Boolean operation. All constants are integers; in Boolean operations 0 is considered false, and 1 is considered true.[2] We normalize addition and multiplication on symbols into multivariate polynomials (e)—for instance, $7+2\cdot x^2\cdot y^3+3\cdot x\cdot y\cdot z^2$—in which the order of symbols in a term and the order of terms in a polynomial are syntactically and semantically irrelevant.[3] If symbols are restricted to variable names, the representation is strongly normalizing: two expressions are semantically equivalent *iff* they are syntactically equal. As it is, it is not (e.g., $\sum_{i=0}^{5}(x[i])$ is equivalent to $\sum_{i=0}^{4}(x[i]) + x[5]$), albeit our simplification engine attempts to keep them as normalized as possible (see Section 5.3).

Index functions are represented by an iteration domain (D) followed by a finite set of guarded expressions (g) delimited by $\bigwedge$. The guards partition the domain, such that the expression whose guard is true (non-zero) provides the value at each index of the domain. For example, variable `rot` at line 7 in Fig. 2 has index function for $i < m.\ (i = 0 \Rightarrow 0) \bigwedge (i \neq 0 \Rightarrow \text{shape}[i-1])$. Consequently, we maintain that guards are mutually exclusive and their disjunction is a tautology. We treat scalars as arrays of length one and in guards we may write `true` and `false` for 1 and 0, for clarity.

Domains. We define three kinds of domains. A *linear* domain has form $for\ i < e^n$ and denotes that iterator i takes values in $0 \ldots e^n - 1$. A *segmented* domain $\bigcup_{k=0}^{e^m} j \geq e_k$ requires that e_k is

[2]This treatment lets us sum over boolean symbols, for example, a sum over the number of true guards in an index function.
[3]A term is implemented as a multiset and a polynomial as a mapping that binds each term to its integer coefficient.

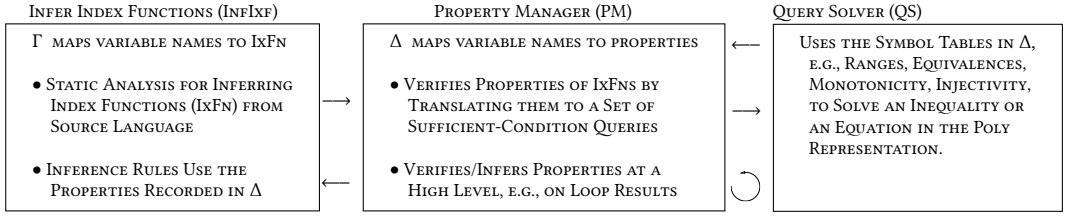

Fig. 4. Bird's Eye View of the Three Logical Components of the Framework, which are deeply connected.

(non-strictly) monotonic in k and $e_k\{k \mapsto 0\} = 0$, and denotes the union of integral intervals $[e_k, e_{k+1})$, where $k = 0 \ldots e^m$ and $e_{k+1} = e_k\{k \mapsto k+1\}$. The representation has ***two key properties***:

1 Permitting empty intervals is essential to expressing jagged arrays such as the II result of $mkII$ in Fig. 2 as: $\bigcup_{k=0}^{m-1}\ j \geq \sum_{k'=0}^{k-1}(shape[k'])$. **true** $\Rightarrow k$ or the result of $mkSgmDescr$ as $\bigcup_{k=0}^{m-1}\ j \geq \sum_{k'=0}^{k-1} \text{shape}[k']$. $(j = \sum_{k'=0}^{k-1} \text{shape}[k'] \Rightarrow xs[k])\ \bigwedge\ (j > \sum_{k'=0}^{k-1} \text{shape}[k'] \Rightarrow 0)$

2 A segmented index function $\bigcup_{k=0}^{e^m} i \geq e_k.\ g$ can always be translated to a linear domain by using the II array, which records at some index i the segment in which i resides, i.e., $k = II[i]$. The linear translation is thus $for\ i < e_k\{k \mapsto e^m + 1\}.\ g\{k \mapsto II[i]\}$.

The third kind models interval $[0, e^n)$ as the union of a linear domain with a segmented one: $(\text{for } i < e_0.\ g^{lin}) \cup (\bigcup_{k=1}^{e^m}.\ j \geq e_k \propto e^n.\ g^{sgm})$ denotes a first interval $[0, e_0)$ in linear form and e^m intervals $[e_k, e_{k+1})$ in segmented form. Denoting by $e_m = e_k\{k \mapsto e^m\}$, the last interval is $[e_m,\ e^n)$, and it must hold that $e_m \leq e^n$ and $e_0 = e_k\{k \mapsto 0\}$. This kind is motivated by the case of scattering at the positions of e^m positive and monotonic indices, which naturally produces $e^m + 1$ intervals.

Extensions. Potential useful extensions of index functions include allowing arbitrary union of linear and segmented domains that can be achieved with the grammar extension:

$$D ::= \text{for} < e \mid \cup_{k=e}^{e} \geq e \qquad S ::= D.g \mid S \cup S \qquad ixfn = \text{Dim}\ x < e.\ S$$

which also allows a general representation of multi-dimensional array by nesting domains; our implementation supports the simple case of 2D arrays having linear domains on each dimension. Finally, the index function may be lifted to support guards that are invariant to the domain iterators, i.e., $ixfn^{gen} ::= (c \Rightarrow ixfn) \mid ixfn^{gen} \wedge ixfn^{gen}$, which would improve the treatment of if expressions. The treatment of these extensions is tedious, but relatively straightforward, i.e., they can be reasoned by combining the treatments of linear and segmented domains.

Architectural Components. The verification analysis is performed during one traversal of the source program, but is split into three connected logical components, summarized in Fig. 4.

InfIxf (left) corresponds to static analysis of function declarations in AoS, A-Normal form that infers index-function representations for the scalar and array variables of integral base types (and for other types in special cases). These are recorded in the symbol table Γ. InfIxf's inference rules commonly require verification of simple (in)equalities, which are delegated to the query solver (QS), while challenging constructs such as `scatter`, also require monotonicity or bijectivity.

The second component, the property manager (PM), handles the recording and verification of properties, such as ranges, equivalences, monotonicity, injectivity, bijectivity, filtering/partitioning. Properties are recorded into corresponding symbol tables, aggregated in Δ. Properties are verified by translating them into a set of simple queries, which form a sufficient condition for the target property to hold and which are sent to the query solver. Importantly, PM attempts to infer certain properties at a high level, in the absence of an index function, e.g., filtering/partitioning an injective array results in an injective array. This enables scaling the analysis, e.g., across loops, which, unless

A segmented shape (e^m, k, e_k) assumes that $0 \le k \le e^m$, e_k is monotonically increasing in k, and $e_0 = e_k\{k \mapsto 0\} = 0$.

Notation: $e_{k+1} = \begin{cases} e^n & \text{if } e^m = 0 \quad \text{(only one segment)} \\ e_k\{k \mapsto k+1\} & \text{if } e^m > 0 \quad \text{(multiple segments)} \end{cases}$ Rcd, Img $= \begin{cases} \text{denotes an integral interval} \\ \text{can be generalized, e.g, union of slices} \end{cases}$

Property	Known	Semantics
$Monotonic^{\odot}\ X$ $\odot \in \{\le, <, \ge, >\}$	$X : [e^n]int$	$0 \le i < j < e^n - 1 \Rightarrow X[i] \odot X[j]$
$Range\ X\ (e^{lb}, e^{ub})$	$X : [e^n]int$	$0 \le i < e^n \Rightarrow e^{lb} \le X[i] \le e^{ub}$
$Equiv\ X\ e$		X has an index function equivalent with the one derived from e
$OrthogPreds$ $(e^m, k, e_k)\ \overline{p_h}$	$h = 1 \ldots v$	$\forall (h_1 \ne h_2) : 0 \le k \le e^m \ \wedge\ e_k \le i < e_{k+1} \Rightarrow \neg(p_{h_1}(i) \wedge p_{h_2}(i))$
$Inj\ X$ Rcd	$X : [e^n]int$	$(0 \le j < i < e^n \ \wedge\ X[i] \in \text{Rcd} \ \wedge\ X[j] \in \text{Rcd}) \Rightarrow X[i] \ne X[j]$
$Bij\ X$ Rcd $(e^m, k, e_k, \text{Img}_k)$	$X : [e^n]int$	$Inj\ X$ Rcd $\wedge$ $\text{Img}_k \overset{\text{set}}{=} \{X[i] \mid e_k \le i < e_{k+1} \ \wedge\ X[i] \in \text{Rcd}\}$
$FiltPart\ Y\ X$ $(e^m, k, e_k, p^f, \overline{p_h^p})$	$X : [e^n]\tau$ $Y : [e^y]\tau$ $h = 1 \ldots v$	$OrthogPreds\ (e^m, k, e_k)\ (p_1^p, \ldots, p_v^p)\ \wedge$ $Y \equiv \texttt{map}\ (\lambda k \rightarrow \texttt{let}\ \sigma = \texttt{filter}\ p^f\ [e_k, .., e_{k+1} - 1]$ $\texttt{let}\ \sigma' = \texttt{partition}_v\ (\overline{p_h^p})\ \sigma$ $\texttt{in map}\ (\lambda i \rightarrow X[i])\ \sigma'\ )\ [0, .., e^m - 1]\ \texttt{|>}\ \texttt{flatten}$
$InvFiltPart\ X$ Img $(e^m, k, e_k, p^f, \overline{p_{h'}^p})$	$X : [e^n]int$ $h' = 1..v\text{-}1$ $p_v = \neg(p_1 \vee \ldots \vee p_{v-1})$ $h = 1 \ldots v$	$OrthogPreds\ (e^m, k, e_k)\ (p_1^p, \ldots, p_v^p)\ \wedge$ $Bij\ X\ \text{Img}\ ((0, k, 0), \text{Img}) \ \wedge\ \lvert\text{Img}\rvert = \sum_{k=0}^{e^m} (\sum_{i=e_k}^{e_{k+1}-1} (p^f(i))) \wedge$ $(\ 0 \le k_1 < k_2 \le e^m \wedge e_{k_1} \le j_1 < e_{k_1+1} \le e_{k_2} \le j_2 < e_{k_2+1} \wedge p^f(j_{1,2}) \Rightarrow Q) \wedge$ $(\forall h : 0 \le k \le e^m \wedge e_k \le j_1 < j_2 < e_{k+1} \wedge p_h^p(j_{1,2}) \wedge p^f j_{1,2} \Rightarrow Q) \wedge$ $(\forall (h_1 < h_2) : 0 \le k \le e^m \wedge e_k \le j_{1,2} < e_{k+1} \wedge p_{h_1}^p(j_1) \wedge p_{h_2}^p(j_2) \wedge p^f(j_{1,2}) \Rightarrow Q)$ **where** Q denotes the query $X[j_1] < X[j_2]$

Fig. 5. Array Properties. X, Y, σ are array variables; p denotes a predicate in lambda form of type $int \rightarrow bool$.

trivial, cannot produce useful index functions for their results. *Finally*, the query solver (QS) uses the properties in Δ to solve (in)equalities by extending Fourier-Motzkin elimination to work in the presence of symbols such as array indexing and sums of array slices.

2.3 Array Properties

Figure 5 presents the array properties supported by our system. We ignore, for the moment, the gray text, e^m, k, e_k, which will be explained later. Increasing and decreasing (strict) monotonicity is standard, as is the range of an array element, except that the bounds are in polynomial representation. *Equiv* expresses equivalences, and it is mostly used for scalars, e.g., the post-condition `m == sum ( map i64.bool (map p xs) )` at line 6 in Fig. 2. *OrthogPreds* requires that the argument predicates are pairwise mutually exlcusive. The implementation supports simple predicates by solving queries as directed by their index functions.

Inj X Rcd says that array X is injective, if it is restricted to the indices that map to values within the *restricted co-domain* Rcd (specified as a range). This is motivated by `scatter dst is _` which ignores the values in *is* that are outside the bounds of `dst`; if all out-of-bounds values in `is` are ∞, and the remaining values are unique, then *Inj is* $[-\infty, \infty)$ holds and `scatter` is safe. *Bij X* Rcd Img similarly says that restricting X to co-domain Rcd results in a subarray that is bijective in Img. It follows that Img $\subseteq$ Rcd and the bijective subarray has no values in Rcd $-$ Img.

$FiltPart\ Y\ X\ p^f\ \overline{p^p}^{v-1}$ declares that the array Y is equivalent to filtering (the indices of) X with $p^f : int \rightarrow bool$ and then performing a v-way partitioning with the pairwise mutually exclusive predicates $\overline{p^p}$, i.e., the elements that succeed under p_1^p come before the ones that succeed under p_2^p and so on. By convention, if no filtering is performed, then $p^f\ \_ = \textbf{true}$ and an unknown filtering is represented by $p^f\ \_ = \textbf{false}$. $\overline{p^p}$ is similar, except that an empty sequence also means unknown.

Verification of the previous property is enabled by $InvFiltPart\ Z\ [0, e^n)\ p^f\ \overline{p^p}^{v-1}$, which essentially declares that Z is an array of indices such that `let Y = scatter Y0 Z X` (where $Y0 : [e^n]\tau$),

results in a filter-partitioning of X with predicates p^f and $\overline{p^p}$, i.e., $FiltPart\ Y\ X\ p^f\ \overline{p^p}^{v-1}$. The semantics is that (1) Z restricted to co-domain $[0, e^n)$ is bijective in interval image $[0, e^n)$, (2) the number of indices that succeed under p^f equals e^n, (3) for any $h = 1 \ldots v$, the indices that succeed under both p^f and p^p_h have monotonically increasing values, and (4) for all pairs $(h_1 < h_2)$ the values of the indices succeeding under p^f and $p^p_{h_1}$ are smaller than the ones that succeed under p^f and p^h_2. For example, in function `partition2` of Fig. 2, variable `indices` at line 15 has index function:

$$\text{for } i < n \;.\; (cs[i] \Rightarrow -1 + \textstyle\sum_{j=0}^{i}(cs[j])) \;\wedge\; (\neg cs[i] \Rightarrow i + \textstyle\sum_{j=1+i}^{n-1}(cs[j])) \quad \text{where } cs[i] = p\ (x[i])$$

which can be shown to satisfy $InvFiltPart$ indices $[0, n)$ $(\backslash\_ \rightarrow \textbf{true})$ $(\backslash i \rightarrow p\ x[i])$, i.e., the partitioning predicate is identified from the guards. The next line performs the scatter and yields the result of `partition2` on which the filtering-partitioning post-condition holds.

The gray text e^m, k, e_k in Fig. 5 extends the properties $OrthogPreds$, Bij, $FiltPart$ and $InvFiltPart$ to cover the segmented case, i.e., where k is a bound variable taking values in $0 \ldots e^m$, and e_k is a monotonically increasing sequence in k that denotes the union of segments. For example, $Bij\ X$ Rcd $(e^m, k, e_k, \text{Img}_k)$ denotes that restricting X to co-domain Rcd results in a per-segment bijective image Img, where Img may depend on k, but Rcd does not. In the source language, the post/preconditions referring to segmented arrays require to bind k as a lambda argument, e.g, a bijective precondition on argument X is expressed as: $Bij\ X\ (e^{lb}_{\text{Rcd}}, e^{ub}_{\text{Rcd}})\ (e^m, \backslash k \rightarrow (e_k, e^{lb}_{\text{Img}}, e^{ub}_{\text{Img}}))$.

Possible generalizations include, for example, extending the interval co-domains to a finite union of slices or LMADs [37, 47] or allowing the range of an element to depend on its index.

3 VERIFYING ARRAY PROPERTIES FROM THEIR INDEX FUNCTION

Section 3.1 introduces notation and the rationale of the design, Section 3.2 presents the verification of injective, bijective and filtering-partitioning properties (simpler properties such as ranges and monotonicity are omitted for brevity), and Section 3.3 presents further automation for inferring properties at a high level, in the absence of an index function.

3.1 Rationale of the Design and Notation

The rationale of the design is to define a small set of properties that (1) are accessible to the non-expert user under a gentle learning curve, (2) are known to the compiler, and (3) expose a compositional algebra that allows the compiler to scale/automate the analysis as much as possible without the user's intervention. However, the user's involvement is key to verification, not only in specifying the properties of the code, but also in performing strategic modifications/annotations that are rooted in the observation that it is much easier to verify a property than to infer it: e.g., it is easier to verify that array elements are within a given range than to infer the range.

As such, the user can guide the analysis by breaking a program into multiple functions that are annotated with pre- and postconditions—including equivalences on array sizes that enable unification of properties across if-expressions. This is facilitated by an architectural design that provides facilities to inspect the relevant index functions and environment in order to reason about whether a given property is actually provable or additional properties need to be specified.

The paper uses the following notation:

- $\mathcal{FV}(o)/\mathcal{BV}(o)$ denote the free/bound variables of an object o. We write "o unifies with o'" if o and o' are syntactically identical up to the names of bound variables [54]. Bound variables appear in sums and segmented domains, for example, x_1 is bound in $\sum_{x_1=e_1}^{e_2}(s_1)$.
- We use $\Gamma; \Delta \vdash f \rightsquigarrow f'$ and $\Gamma; \Delta \vdash e \rightsquigarrow e'$ to denote simplification of index functions and expressions. We assume that both are already in simplified form (see Section 5), and operations and substitutions such as $e_1 - e_2$ and $e\{k \mapsto e_k\}$ also simplify the result.

- The algebraic environment Δ is seen as a record whose fields are symbol tables, named after the corresponding properties, e.g., $\Delta.Inj$ denotes injectivity and the new environment created by adding binding $x \mapsto \text{Rcd}$ to $\Delta.Inj$ is denoted by: Δ with $Inj = \Delta.Inj \cup \{x \mapsto \text{Rcd}\}$.
- In other places we are less explicit: $\Delta \wedge (e^x = e^y) \wedge (e_1 \leq e_2 < e_3)$ denotes extending the equivalence symbol table $\Delta.Equiv$ with bindings derived from $e^x = e^y$ and the ranges table $\Delta.Range$ with the bindings derived from inequalities $e_1 \leq e_2$, $e_2 < e_3$ and $e_1 < e_3$, as explained in Section 5. The last inequality $e_1 < e_3$ is not necessarily subsumed by the other two, e.g., assuming a positive shape and k, $\sum_{k'=0}^{k-1}(\text{shape}[k']) \leq j < \sum_{k'=0}^{k}(\text{shape}[k'])$ would result in the last inequality being simplified to $\text{shape}[k] > 0$, which would improve the lower bound of symbol $\text{shape}[k]$ to be 1 rather than 0.
- For convenience, we also define the shorthand notation below for extending Δ with the inequalities assumed by $ixfn$'s domains, where $e_{k+1} = e_k\{k \mapsto k+1\}$ and $e_{sz} = e_{k+1} - e_k$:
$$\Delta^{+\text{for } i<e_n} = \Delta \wedge 0 \leq i < e_n \quad \text{and} \quad \Delta^{+\cup_{k=0}^{e^m} j \geq e_k} = \Delta \wedge 0 \leq k \leq e_m \wedge e_k \leq j < e_{k+1} \wedge 0 \leq e_{sz}$$
- Queries are marked in gray, use the notation $\Delta \vdash c^1 \stackrel{?}{\Rightarrow} c^2$ and succeed when it can be shown that c^1 implies c^2 in context Δ. Queries proceed by converting c^1 to DNF (i.e., $c^1 = c^1_1 \vee \ldots \vee c^1_q$), and then asking the solver to prove c^2 in contexts $\Delta \wedge c^1_i$ where $i = 1 \ldots q$. The query succeeds if all q sub-queries succeed.

3.2 Verifying Injectivity and Bijectivity of Index Functions

Figure 6 presents the inference rules that verify injective and bijective properties. Rules BIJV1 and BIJV2 take the form $\Gamma; \Delta \vdash (x, \text{Rcd}, (k, \text{Img})) \stackrel{Bij}{\rightarrow} (\textbf{bool}, \Delta')$ that answers whether the array denoted by variable x, when viewed as an index function restricted to the pre-image of Rcd, has bijective image Img; k enables the treatment of (jagged) segmented arrays, i.e., when Img depends on k then each segment (of index k) of the array has bijective image Img. The result is a boolean (**true** means success) and a new symbol table, that possibly extends $\Delta.Bij$ with a newly verified binding.

Rule BIJV1 requires that x already has a binding, denoted $(\text{Rcd}_2, (\text{Sgm}_2, \text{Img2}))$, where Sgm_2 denotes a segmented shape (e^m, k, e_k), as introduced in Fig. 5 and Section 2.3. The rule succeeds if (1) the queried image Img_1 is equivalent to the one of the binding Img_2, and (2) it can be proven that $\text{Img}_2 \subseteq \text{Rcd}_1 \subseteq \text{Rcd}_2$.[4] Since by construction $\text{Img}_2 \subseteq \text{Rcd}_2$, it follows that x can have no points in $\text{Rcd}_2 - \text{Img}_2$ therefore the restriction to co-domain $\text{Rcd}_1 \subseteq \text{Rcd}_2$ also has bijective image Img_1.

Rule BIJV2 covers the case when x does not have a binding in $\Delta.Bij$. It creates an index function that maps all points inside the queried Rcd to itself (`x[i]`) and the other points to a special ∞ symbol. Then it simplifies it (see Section 4) and tries to prove bijectivity of the resulted index function.

This is achieved by a judgment of form $\Delta \vdash ((k, \text{Img}), f) \stackrel{Bij}{\rightarrow} (\textbf{true}, (\text{Sgm}, \text{Img}))$ that similarly returns **true** for success, together with the (segmented) bijective image; Rcd is missing because all the points outside it have been mapped to ∞.

Rule BIJF1 covers the case when the index function f is segmented, but Img, denoted by interval $[e^{lb}, e^{ub}]$, is not, i.e., the image does not depend on k. Bijectivity is verified by checking that (1) all guarded expressions that do not correspond to ∞ yield values within Img, (2) f is injective outside ∞ points, and (3) the cardinal of Img is equal with the number of indices that are not mapped to ∞.

Rule BIJF2 covers two other cases[5]: one that refers to when both f and Img are segmented, in which case bijectivity is verified for each segment of f, and another one, in which f has the linear domain and the property is considered non segmented. Verification is similar to BIJF1.

[4]Verification of $[e^{lb}_1, e^{ub}_1] \subseteq [e^{lb}_2, e^{ub}_2]$ is achieved by trying to prove $(e^{lb}_1 > e^{ub}_1) \vee (e^{lb}_1 \geq e^{lb}_2 \wedge e^{ub}_1 \leq e^{ub}_2)$.
[5]A special case (not shown) refers to when f is segmented, Img is equal to the whole (unsegmented) domain of f, and each segment $[e_k, e_{k+1})$ contains values that are a permutation of $[e_k, e_{k+1})$. The result is the segmented bijection $[e_k, e_{k+1})$.

Verifying bijectivity of a variable x in restricted co-domain. $\boxed{\Gamma; \Delta \vdash (x, \text{Rcd}, (k, \text{Img})) \xrightarrow{Bij} (\textbf{true}, \Delta')}$

$$\frac{\begin{array}{c}\Delta.Bij(x) = (\text{Rcd}_2, (\text{Sgm}_2, \text{Img}_2)) \\ \text{Sgm}_2 = (e^m, k, e_k) \\ \text{Img}_2 \text{ unifies with } \text{Img}_1\{k' \mapsto k\} \\ \Delta \vdash 0 \le k \le e^m \overset{?}{\Rightarrow} \text{Img}_2 \subseteq \text{Rcd}_1 \subseteq \text{Rcd}_2\end{array}}{\Gamma; \Delta \vdash (x, \text{Rcd}_1, (k', \text{Img}_1)) \xrightarrow{Bij} (\textbf{true}, \Delta)} \text{(BijV1)}$$

$$\frac{\begin{array}{c}e^n = \text{length } x \qquad \text{fresh } i \\ f = \text{for } i < e^n \,.\, (x[i] \in \text{Rcd} \Rightarrow x[i]) \wedge (x[i] \notin \text{Rcd} \Rightarrow \infty) \\ \Gamma; \Delta \vdash f \rightsquigarrow f' \quad \Delta \vdash ((k, \text{Img}), f') \xrightarrow{Bij} (\textbf{true}, (\text{Sgm}', \text{Img}')) \\ \Delta' = \Delta \text{ with } Bij = \Delta.Bij \cup \{x \mapsto (\text{Rcd}, (\text{Sgm}', \text{Img}'))\end{array}}{\Gamma; \Delta \vdash (x, \text{Rcd}, (k, \text{Img})) \xrightarrow{Bij} (\textbf{true}, \Delta')} \text{(BijV2)}$$

Verifying bijectivity in image of an index function f. $\boxed{\Delta \vdash ((k, \text{Img}), f) \xrightarrow{Bij} (\textbf{true}, (\text{Sgm}, \text{Img}))}$

$$\frac{\begin{array}{c}D = \bigcup_{k=0}^{e^m} .\text{for } i \ge e_k \quad k' \notin \mathcal{FV}(e^{lb}) \cup \mathcal{FV}(e^{ub}) \qquad \forall h \in 1, \ldots, v \,:\, \Delta^{+\text{for } D} \vdash c_h \overset{?}{\Rightarrow} e^{lb} \le e_h \le e^{ub} \\ \Delta \vdash f \xrightarrow{Inj} \textbf{true} \qquad e_{k+1} = e_k\{k \mapsto k+1\} \qquad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} e^{ub} + 1 - e^{lb} = \sum_{k=0}^{e^m} (\sum_{i=e_k}^{e_{k+1}-1} (c_1 \vee \ldots \vee c_v))\end{array}}{\Delta \vdash ((k', [e^{lb}, e^{ub}]), f = \text{for } D \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h)[\wedge(c^\infty \Rightarrow \infty)]) \xrightarrow{Bij} (\textbf{true}, ((0, k', 0), [e^{lb}, e^{ub}]))} \text{(BijF1)}$$

$$i, \text{Sgm}, e_{k+1}, \text{Img} = \begin{cases} i, (e^m, k, e_k), e_{k+1}, \text{Img}'\{k' \mapsto k\} & \textbf{if } D = \bigcup_{k=0}^{e^m} .\text{for } i \ge e_k, \textbf{ where } e_{k+1} = e_k\{k \mapsto k+1\} \\ i, (0, k', 0), e^n, \text{Img}' & \textbf{if } D = \text{for } i < e^n \end{cases}$$

$$\frac{\begin{array}{c}\Delta \vdash f \xrightarrow{Inj} \textbf{true} \qquad \text{Img} = [e^{lb}, e^{ub}] \qquad \forall h \in \{1, \ldots, v\} : \Delta^{+\text{for } D} \vdash c_h \overset{?}{\Rightarrow} e^{lb} \le e_h \le e^{ub} \\ \text{Sgm} = (e^m, k, e_k) \quad e_{k+1} = e_k\{k \mapsto k+1\} \qquad \Delta^{+\text{for } D} \vdash \textbf{true} \overset{?}{\Rightarrow} e^{ub} + 1 - e^{lb} = \sum_{i=e_k}^{e_{k+1}-1} (c_1 \vee \ldots \vee c_v)\end{array}}{\Delta \vdash ((k', \text{Img}'), f = \text{for } D \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h)[\wedge(c^\infty \Rightarrow \infty)]) \xrightarrow{Bij} (\textbf{true}, (\text{Sgm}, \text{Img}))} \text{(BijF2)}$$

Verifying injectivity of a variable x in a (restricted) co-domain. $\boxed{\Gamma; \Delta \vdash (\text{Rcd}, x) \xrightarrow{Inj} (true, \Delta')}$

$$\frac{\begin{array}{c}\text{Rcd}_2 = \Gamma.Inj(x) \\ \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} \text{Rcd}_1 \subseteq \text{Rcd}_2\end{array}}{\Gamma; \Delta \vdash (\text{Rcd}_1, x) \xrightarrow{Inj} (\textbf{true}, \Delta)} \text{(InjV1)}$$

$$\frac{\begin{array}{c}(i, e^n, c, e^v) = \begin{cases}(i, \text{ length } x, c^f \wedge x[i] \in \text{Rcd}, x[i]) & \textbf{if } \Delta.FP(y) = (x, \lambda i.c^f, \_) \\ (\text{fresh } i, \text{ length y}, y[i] \in \text{Rcd}, y[i]) & \textbf{otherwise}\end{cases} \\ f = \text{for } i < e^n \,.\, (c \Rightarrow e^v) \wedge (\neg c \Rightarrow \infty) \quad \Gamma; \Delta \vdash f \rightsquigarrow f' \quad \Delta \vdash f' \xrightarrow{Inj} \textbf{true}\end{array}}{\Gamma; \Delta \vdash (\text{Rcd}, y) \xrightarrow{Inj} (\textbf{true}, \Delta \text{ with } Inj = \Delta.Inj \cup \{y \mapsto \text{Rcd}\})} \text{(InjV2)}$$

Verifying injectivity of an index function f. $\boxed{\Delta \vdash f \xrightarrow{Inj} \textbf{true}}$

$$\frac{\begin{array}{c}f = \text{for } j < e^n . \overline{ge}\,[\wedge(\_ \Rightarrow \infty)] \\ \text{fresh } j' \\ \Delta' = \Delta \wedge 0 \le j < e^n \wedge 0 \le j' < e^n \\ (\quad \Delta' \vdash_{\neq, j, j'} \overline{ge} \xrightarrow{Mon} (\text{true}, \_) \\ \textbf{OR} \quad \Delta' \vdash_{j, j'} \overline{ge} \xrightarrow{Inj=} \text{true} \quad)\end{array}}{\Delta \vdash f \xrightarrow{Inj} \textbf{true}} \text{(InjF1)}$$

$$\frac{\begin{array}{c}fresh\ k'\ j' \quad e_{k+1} = e_k\{k \mapsto k+1\} \quad e'_k = e_k\{k \mapsto k'\} \\ \Delta' = \Delta \wedge (0 \le k \le e^m) \wedge (e_k \le j, j' < e_{k+1}) \\ (\ \Delta' \vdash_{\neq, j, j'} \overline{ge} \xrightarrow{Mon} (\textbf{true}, \_) \quad \textbf{OR} \quad \Delta' \vdash_{j, j'} \overline{ge} \xrightarrow{Inj=} \textbf{true}\ ) \\ e'_{k+1} = e_k\{k \mapsto k'+1\} \quad \overline{ge}' = \overline{ge}\{j \mapsto j', k \mapsto k'\} \\ \Delta'' = \Delta \wedge (0 \le k < k' \le e^m) \wedge (e_k \le j < e_{k+1}) \wedge (e'_k \le j' < e'_{k+1}) \\ \Delta'' \vdash_{\neq} (\overline{ge}, \overline{ge}') \xrightarrow{Cmp} \textbf{true}\end{array}}{\Delta \vdash \bigcup_{k=0}^{e^m} .\text{for } j \ge e_k \,.\, \overline{ge}\,[\wedge(\_ \Rightarrow \infty)] \xrightarrow{Inj} \text{true}} \text{(InjF2)}$$

Fig. 6. Verifying injectivity and bijectivity of variables and index functions denoting arrays.

Rules InjV1 and InjV2 verify injectivity in a restricted co-domain Rcd of a variable x and are similar in the form of judgments and treatment with rules BijV1 and BijV2. What differs is that InjV2 also checks whether the target array y is a filtering partitioning of some array x, in which case it reasons in terms of x while taking into consideration the filtering predicate.

The rules InjF1 and InjF2, of form $\Delta \vdash f \xrightarrow{Inj} \textbf{bool}$, verify the injectivity of the non-∞ elements of index function f. Rule InjF1 treats index functions having linear domains and uses the helper inference rules $\xrightarrow{Inj=}$ and $\xrightarrow{Mon}$ defined in Fig. 7. Injectivity is verified in either one of two ways:

Guarded-Exps Helpers. $\boxed{\Delta \vdash_{\odot,i,j} \overline{ge} \overset{Mon}{\rightarrow} (\mathbf{true}, \sigma) \quad \Delta \vdash_{i,j} \overline{ge} \overset{Inj=}{\rightarrow} \mathbf{true} \quad \Delta \vdash_{\odot} (\overline{ge}^1, \overline{ge}^2) \overset{Cmp}{\rightarrow} \mathbf{true}}$

$$\frac{\begin{array}{c}\forall h \in 1 \ldots v \text{ it holds: } (\quad c'_h = c_h\{i \mapsto j\} \quad e'_h = e_h\{i \mapsto j\} \quad \Delta \wedge i < j \vdash c_h \wedge c'_h \overset{?}{\Rightarrow} e_h \odot e'_h \quad) \\ \text{There exists a sorting permutation } \sigma \text{ of } \{1,\ldots,v\} \text{ such that for all } h \in \{1,\ldots,v\} \text{ with } \sigma(h) < v: \\ \Delta \vdash (j < i \vee i < j) \wedge c'_{\sigma(h)} \wedge c_{\sigma(h)+1} \overset{?}{\Rightarrow} e'_{\sigma(h)} < e_{\sigma(h)+1}\end{array}}{\Delta \vdash_{\odot,i,j} \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \overset{Mon}{\rightarrow} (\mathbf{true}, \sigma)} \text{(MonGe)}$$

$$\frac{\begin{array}{c}\sigma = \{i \mapsto j\} \;\; V = \{1,\ldots,v\} \;\; \forall h,l \in V \times V: \\ \Delta \vdash (e_h = \sigma(e_l)) \wedge c_h \wedge \sigma(c_l) \overset{?}{\Rightarrow} i = j\end{array}}{\Delta \vdash_{i,j} \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \overset{Inj=}{\rightarrow} \mathbf{true}} \text{(InjGe)} \qquad \frac{\forall h,l \in \{1,\ldots,v\} \times \{1,\ldots,w\}: \;\; \Delta \vdash c^1_h \wedge c^2_l \overset{?}{\Rightarrow} e^1_h \odot e^2_l}{\Delta \vdash_{\odot} (\bigwedge_{h=1}^{v} (c^1_h \Rightarrow e^1_h), \bigwedge_{l=1}^{w} (c^2_l \Rightarrow e^2_l)) \overset{Cmp}{\rightarrow} \mathbf{true}} \text{(CmpGe)}$$

Filtering-partitioning property of a variable. $\boxed{\Gamma; \Delta \vdash x \overset{FP}{\rightarrow} (\mathbf{true}, \Delta')}$

$$\frac{\Delta.FP(y) = (x, \text{fp\_prop})}{\Gamma; \Delta \vdash y \overset{FP}{\rightarrow} (\mathbf{true}, \Delta)} \text{(FPV1)} \qquad \frac{\begin{array}{c}\Gamma(y) = \text{for } i < e^y.\; true \Rightarrow x[is^{-1}[i]] \qquad \text{Img} = [0, e^y) \\ \Gamma; \Delta \vdash (\; is, \text{Img}\;) \overset{IFP}{\rightarrow} (\mathbf{true}, \Delta') \qquad \Delta'.IFP(is) = (\_, \text{prop})\end{array}}{\Gamma; \Delta \vdash y \overset{FP}{\rightarrow} (\mathbf{true},\; \Delta' \text{ with } FP = \Delta'.FP \vee \{y \mapsto (x, \text{prop})\})} \text{(FPV2)}$$

Fig. 7. Guarded Expressions Helpers & Translating Filter-Partitioning Property to Inverse Filtering Partitioning

The first approach uses $\overset{Inj=}{\rightarrow}$ to prove that $x[i_1] = x[i_2]$ implies $i_1 = i_2$ for the values of the guarded expressions other than the one leading to ∞, i.e., $\_ \Rightarrow \infty$. This is achieved by the query-solver technique presented in Section 5.4. The second approach uses $\overset{Mon}{\rightarrow}$ to prove a sufficient condition based on piecewise monotonicity, namely: ***If*** for any guarded expression $c_h \Rightarrow e_h$ the e_h values are distinct—which is denoted by $\neq$ in $\vdash_{\neq,j,j'}$ and typically comes down to strict monotonicity—***and*** there exists a sorting permutation σ that reorganizes the guarded expressions such that their values always increase across them ***then*** the values of those guarded expressions are injective.

Rule InjF2 is concerned with index functions that have segmented domains. It applies a similar reasoning to InjF1 within a segment, and in addition, it uses the $\Delta \vdash_{\neq} (\overline{ge}, \overline{ge'})$ rule to prove that values belonging to different segments are different; $e_1 \neq e_2$ is solved by checking $e_1 < e_2$ or $e_1 > e_2$.

Rules FPV1 and FPV2 in Fig. 7 sketch the treatment of filtering-partitioning properties. Since such properties can only result in an index function of form for $i < e^y.\; true \Rightarrow x[is^{-1}[i]]$, rule FPV2 pattern matches said form and tries to infer the inverse filter-partitioning property (*IFP*) on *is* in bijective image $[0, e^y)$. *IFP*'s semantics was discussed in Section 2.3. Its implementation (not shown) is similar to bijectivity rule BijV2, except that: (1) index function f is filtering out the points outside image $[0, e^y)$; (2) the calls to rule InjGe (*Inj* =) are eliminated from InjF1 and InjF2; (3) the calls to rules MonGe (*Mon*) and CmpGe (*Cmp*) are instantiated with $<$ instead of $\neq$, to check strictly *increasing* monotonicity. The predicates are obtained from the guards of the simplified f: the filtering one by negating the guard of the ∞ value, and the partitioning ones from the guards producing legal indices, ordered by the σ permutation computed by rule MonGe.

3.3 Inferring New Properties at a High Level

Figure 8 demonstrates several illustrative rules for inferring properties at a high level. Judgments take the form $\Gamma; \Delta \vdash (y, e^0) \overset{\Delta U}{\rightarrow} \Delta'$, in which the argument indicates a source-language binding `let` $y = e^0$ and the result is a potentially extended symbol table.

Rule ΔUeBij states that if y is a partitioning of x and x is bijective in a restricted co-domain, then the bijectivity property is transferred to y. Other (not shown) properties are similarly derived,

A Few Samples of Inferring Properties at a High Level.

$$\boxed{\Gamma;\Delta \vdash (y, e^0) \overset{\Delta U}{\rightarrow} \Delta'}$$

$$\frac{\Delta.FP(y) = (x,\ \lambda i.e^f,\ \ldots)\quad e^f = \textbf{true}\quad \Delta.Bij(x) = \text{RcdSgIm}}{\Gamma;\Delta \vdash (y,e) \overset{\Delta U}{\rightarrow} \Delta \text{ with } Bij = \Delta.Bij \cup \{y \mapsto \text{RcdSgIm}\}} \quad (\Delta\text{UEBIJ})$$

$$\frac{\begin{array}{c}\overline{y}^v = \text{res}(e^1)\qquad \overline{z}^v = \text{res}(e^2)\\ \Gamma;\Delta \vdash (x_1, x^c, y_1, z_1) \overset{IfFP}{\rightarrow} \Delta_1 \quad \ldots\\ \Gamma;\Delta_{v-1} \vdash (x_v, x^c, y_v, z_v) \overset{IfFP}{\rightarrow} \Delta_v\\ \Gamma;\Delta_v \vdash (\overline{x}^v, x^c, \overline{y}^v, \overline{z}^v) \overset{IfBij}{\rightarrow} \Delta' \quad (\ldots)\end{array}}{\Gamma;\Delta \vdash (\overline{x}^v, \texttt{if } x^c \texttt{ then } e^1 \texttt{ else } e^2) \overset{\Delta U}{\rightarrow} \Delta'} \quad (\Delta\text{UIF})$$

$$\frac{\begin{array}{c}\textbf{FOR ANY } h = 1\ldots v \textbf{ SUCH THAT:}\\ (R^1, \text{SgIm}^1) = \Delta.Bij(x_h^1)\{\overline{x_l^1 \mapsto y_l}^{l=1\ldots v}\}\\ (R^2, \text{SgIm}^2) = \Delta.Bij(x_h^2)\{\overline{x_l^2 \mapsto y_l}^{l=1\ldots v}\}\\ \text{SgIm}^1 \text{ unifies with } \text{SgIm}^2 \qquad R = \begin{cases} R^1 & \textbf{if } \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} R^1 \subseteq R^2 \\ R^2 & \textbf{if } \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} R^2 \subseteq R^1 \end{cases}\\ \textbf{UPDATE } M \leftarrow M \cup \{y_h \mapsto (R, SgIm^1)\}\end{array}}{\Gamma;\Delta \vdash (\overline{y}^v, \_, \overline{x^1}, \overline{x^2}) \overset{IfBij}{\rightarrow} \Delta \text{ with } Bij = \Delta.Big \cup M} \quad (\text{IFBIJ})$$

$$\frac{\begin{array}{c}\Delta.FP(x^1) = (z^1, p_0^1, \overline{p_h^1}^{h=1..v_1})\\ \Delta.FP(x^2) = (z^2, p_0^2, \overline{p_h^2}^{h=1..v_2})\quad z^1 = z^2\quad v_1 = v_2\\ \textbf{for all } h = 0\ldots v_1 : \Gamma;\Delta \vdash (x^c, p_h^1, p_h^2) \overset{predU}{\rightarrow} p_h\\ \Delta' = \Delta \text{ with } FP = \Delta.FP \cup \{y \mapsto (z_1, \overline{p_h}^{h=0..v_1})\}\end{array}}{\Gamma;\Delta \vdash (y, x^c, x^1, x^2) \overset{IfFP}{\rightarrow} \Delta'} \quad (\text{IFFP})$$

$$\frac{p^1 = \lambda i.\ e^1 \qquad p^2 = \lambda i.\ e^2 \qquad e^{1,2} \neq \textbf{false}\qquad \Gamma;\Delta \vdash (x^c \wedge e^1)\ \vee\ (\neg x^c \wedge e^2\{i' \mapsto i\}) \rightsquigarrow e}{\Gamma;\Delta \vdash (x^c, p^1, p^2) \overset{predU}{\rightarrow} \lambda i.\ e} \quad (\text{PREDUS})$$

$$\frac{p^1 = \lambda i.\ \textbf{false} \qquad \textbf{OR} \qquad p^2 = \lambda i.\ \textbf{false}}{\Gamma;\Delta \vdash (x^c, p^1, p^2) \overset{predU}{\rightarrow} \lambda i.\ \textbf{false}} \quad (\text{PREDUF})$$

Fig. 8. Inferring and Verifying Properties at a High Level.

e.g., filtering preserves monotonicity and filtering-partitioning preserves injectivity, and can be used to refine ranges (by min/maxing the existent upper/lower bounds of y with those of x).

Rule ΔUIF aims to unify properties across if expressions, which is demonstrated for bijectivity (IFBIJ) and filtering-partitioning properties (IFFP). Rule IFBIJ denotes by $\overline{y}$ the variable bound to the if expression and by $\overline{x^1}$ and $\overline{x^2}$ the variable results of the then and else expressions, respectively, all of them necessarily having the same (tuple) cardinality v. If for any $h \in 1\ldots v$ it happens that both x_h^1 and x_h^2 are bijective, with equivalent images and restricted co-domains that are in an inclusion relation, then the bijective property is transmitted to the corresponding if result by choosing the smaller restricted co-domain. The rule uses an improvement that substitutes the results of the then/else branches for the ones of the if result in the bijective property before unifying them. This enables the human in the loop, i.e., even if the images are not equal, the user can "force" unification by wrapping the then/else expressions into functions whose post-conditions use dependent typing across results—e.g., if both function have post-condition $\lambda n\ X.\ Bij\ X\ (0, 8 * n)\ (\_, (n, 2 * n))$ then bijectivity will unify even if the n result might differ across branches.

Rule IFFP uses a notation similar to IFBIJ, except that, for simplicity, it does not use the dependent-typing refinement, i.e., y, x_1, x_2 denote single variables. In addition, it denotes with x^c the if condition. The rules states that if both x^1 and x^2 are filtering partitioning of the same array (variable z) with a matching number of partitions v, then they can be accurately unified across the if expression. The resulting predicates are computed by PREDUS as $p\ i = (x^c \wedge p^1(i)) \vee (\neg x^c \wedge p^2(i))$, where p^1 and p^2 corresponds to the predicates of x^1 and x^2, unless one of them denotes unknown ($\lambda i.\textbf{false}$), in which case they unify as unknown by rule PREDUF. If the number of partitions do not match, then they unify as a fully-unknown partitioning, denoted by an empty sequence of predicates (not shown).

4 INFERRING INDEX FUNCTIONS

Our source programs are compositions of array combinators, encouraging a point-free programming style. However, pointful reasoning naturally emerges as a proof discipline for proving properties. For instance, to prove $Inj\ xs\ (-\infty, \infty)$ it is sufficient to verify the query

$$\Delta \vdash (0 \leq i < \text{length } xs\ \wedge\ 0 \leq j < \text{length } xs\ \wedge\ xs[i] = xs[j]) \overset{?}{\Rightarrow} i = j.$$

Conversion from source to index functions $\boxed{\Gamma; \Delta \vdash e^* \overset{Src}{\rightarrow} (\Gamma', \Delta', f)}$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n \overset{Src}{\rightarrow} (\Gamma, \Delta, \text{for } \bullet \,.\, \texttt{true} \Rightarrow n)}\,(\textsc{Const}) \qquad \frac{x \notin \Gamma \quad \text{fresh } i \quad f = \text{for } i < \text{length } x \,.\, \texttt{true} \Rightarrow x[i]}{\Gamma; \Delta \vdash x \overset{Src}{\rightarrow} (\Gamma\{x \mapsto f\}, \Delta, f)}\,(\textsc{Var1}) \qquad \frac{\Gamma(x) = f}{\Gamma; \Delta \vdash x \overset{Src}{\rightarrow} (\Gamma, \Delta, f)}\,(\textsc{Var2})$$

$$\frac{\Gamma; \Delta \vdash e^0 \rightsquigarrow (\Gamma', \Delta', f_1) \quad \Gamma'\{x \mapsto f_1\}; \Delta' \vdash e^* \rightsquigarrow (\Gamma'', \Delta'', f_2)}{\Gamma; \Delta \vdash \texttt{let } x = e^0 \texttt{ in } e^* \overset{Src}{\rightarrow} (\Gamma'', \Delta'', f_2)}\,(\textsc{Let}) \qquad \frac{\Gamma(x_2) = \text{for } \bullet \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \quad \forall h = 1, \ldots, v \,.\, \Delta \vdash c_h \overset{?}{\Rightarrow} 0 \leq e_h < \text{length } x_1}{\Gamma; \Delta \vdash x_1[x_2] \overset{Src}{\rightarrow} (\Gamma, \Delta, \text{for } \bullet \,.\, x_1[x_2])}\,(\textsc{Idx})$$

$$\frac{\Gamma(x_2) = \text{for } i < e^n \,.\, g_2 \quad \Gamma; \Delta^{+\text{for } i<e^n} \vdash e^*\{x_1 \mapsto x_2[i]\} \rightsquigarrow (\Gamma', \Delta', \text{for } \bullet \,.\, g_1)}{\Gamma; \Delta \vdash \texttt{map } (\lambda x_1 \,.\, e^*)\ x_2 \overset{Src}{\rightarrow} (\Gamma', \Delta, \text{for } i < e^n \,.\, g_1)}\,(\textsc{Map}) \qquad \frac{\Gamma(x) = \text{for } \bullet \,.\, \texttt{true} \Rightarrow e \quad \text{fresh } i}{\Gamma; \Delta \vdash \texttt{iota } x \overset{Src}{\rightarrow} (\Gamma, \Delta, \text{for } i < e \,.\, \texttt{true} \Rightarrow i)}\,(\textsc{Iota})$$

$$\frac{\begin{array}{c}\Gamma(x) = \text{for } \bullet \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \\ \Gamma; \Delta \vdash c_1 \wedge e_1 \vee \ldots \vee c_v \wedge e_v \rightsquigarrow c_T \quad \Gamma; \Delta \vdash \neg c_T \rightsquigarrow c_F \\ \Gamma; \Delta, c_T \vdash e_2^* \rightsquigarrow (\Gamma', \Delta', \text{for } \bullet \,.\, g_2) \quad \Gamma'; \Delta', c_F \vdash e_3^* \rightsquigarrow (\Gamma'', \Delta'', \text{for } \bullet \,.\, g_3) \\ \text{fresh } x_T, x_F \quad \Gamma''' = \Gamma''\{x_T \mapsto \text{for } \bullet \,.\, g_2, x_F \mapsto \text{for } \bullet \,.\, g_3\}\end{array}}{\Gamma; \Delta \vdash \texttt{if } x \texttt{ then } e_2^* \texttt{ else } e_3^* \overset{Src}{\rightarrow} (\Gamma''', \Delta'', \text{for } \bullet \,.\, (x \Rightarrow x_T) \bigwedge (\neg x \Rightarrow x_F))}\,(\textsc{If})$$

$$\frac{x_1 \text{ has type } \tau \rightarrow \mathit{bool}}{\Gamma; \Delta \vdash x_1\ x_2 \overset{Src}{\rightarrow} (\Gamma, \Delta, \text{for } \bullet \,.\, 1 \Rightarrow x_1[x_2])}\,(\textsc{Unint.})$$

Fig. 9. Converting the source language to index functions.

$$\begin{array}{rcl} \mathcal{G} & ::= & \square \mid g \bigwedge \mathcal{G} \mid \mathcal{G} \bigwedge g \\ \mathcal{K} & ::= & \square \mid e + \mathcal{K} \mid e \cdot \mathcal{K} \mid x[\mathcal{K}] \mid \sum_{x=\mathcal{K}}^{e}(s) \mid \sum_{x=e}^{\mathcal{K}}(s) \mid \sum_{x=e}^{e}(\mathcal{K}) \\ & \mid & \neg\mathcal{K} \mid \mathcal{K} \wedge c \mid c \wedge \mathcal{K} \mid \mathcal{K} \vee c \mid c \vee \mathcal{K} \mid \mathcal{K} \odot e \mid e \odot \mathcal{K} \end{array}$$

Fig. 10. Reduction context grammar for guarded expressions ($\mathcal{G}$) and symbols ($\mathcal{K}$).

To bridge the gap between program and query, we do away with the abstraction provided by array combinators by transforming source programs to index functions, enabling precise reasoning about array elements. Figure 9 provides an initial set of rules for inferring index functions (excluding `scan` and `scatter`). For simplicity, all variables are treated as flat arrays (scalars are single-element arrays), the rules target a subset of the language without tuples (SoA), and we sometimes write $\bullet$ for $i < 1$. Function declarations are translated similarly to let-bindings, with these additions: (1) index functions are created for the formal arguments and bound in Γ; (2) preconditions on arguments are added to Δ; and (3) the postcondition is verified on the resulting index functions.

We demonstrate the rules on our running example `partition2` from Fig. 2. Recall that the example takes a predicate p as argument, which we treat as an uninterpreted function by inserting an indexing symbol. The first source expression **`let`** `cs =` **`map`** `(\x -> p x) xs` is transformed by:

$$\frac{\dfrac{p \text{ has type } \texttt{f64} \rightarrow \texttt{bool}}{\Gamma; \Delta^{+\text{for } i<n} \vdash p\ xs[i] \overset{Src}{\rightarrow} (\Gamma,\ \Delta^{+\text{for } i<n},\ \text{for } j < 1 \,.\, \texttt{true} \Rightarrow p[xs[i]])}\,(\textsc{Unint}) \quad \Gamma(xs) = \text{for } i < n \,.\, \texttt{true} \Rightarrow xs[i]}{\Gamma; \Delta \vdash \texttt{map } (\lambda x.\, p\ x)\ xs \overset{Src}{\rightarrow} (\Gamma,\ \Delta,\ \text{for } i < n \,.\, \texttt{true} \Rightarrow p[xs[i]])}\,(\textsc{Map})$$

To track positional dependencies backwards to the formal arguments of a function declaration, we substitute index functions into other index functions by reduction over indexing symbols and variables. Reduction contexts (Fig. 10) define where a reduction is to occur in an index function. $\mathcal{G}\langle c \Rightarrow e \rangle$ denotes the guarded expression obtained by replacing $\square$ with $c \Rightarrow e$ in the context $\mathcal{G}$. For instance, if $\mathcal{G} = (c_1 \Rightarrow e_1) \bigwedge \square \bigwedge (c_3 \Rightarrow e_3)$, then $\mathcal{G}\langle c_2 \Rightarrow e_2 \rangle$ is equivalent to $(c_1 \Rightarrow e_1) \bigwedge (c_2 \Rightarrow$

Rewrite rules $\boxed{\Gamma;\Delta \vdash f \to f' \qquad \Gamma;\Delta \vdash e \to e' \qquad \Gamma;\Delta \vdash g \to g'}$

$$\frac{\Gamma(x) = \text{for } i < e^n \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \quad \mathcal{FV}(e_0) \cap \mathcal{BV}(\mathcal{K}\langle x[e_0]\rangle) = \emptyset \quad \sigma = \{i \mapsto e_0\}}{\Gamma;\Delta \vdash \text{for } D \,.\, \mathcal{G}\langle c \Rightarrow \mathcal{K}\langle x[e_0]\rangle\rangle \to \text{for } D \,.\, \mathcal{G}\langle \bigwedge_{h=1}^{v} (c \wedge \sigma(c_h) \Rightarrow \mathcal{K}\langle \sigma(e_h)\rangle)\rangle}\,(\text{I.Sub1}) \qquad \frac{\Gamma(x) = \text{for } \bullet \,.\, g}{\Gamma \vdash \mathcal{K}\langle x\rangle \to \mathcal{K}\langle x[0]\rangle}\,(\text{E.Lift})$$

$$\frac{\Gamma(x) = \text{for } i < e^n \,.\, \texttt{true} \Rightarrow e_1}{\Gamma;\Delta \vdash \mathcal{K}\langle x[e_0]\rangle \to \mathcal{K}\langle e_1\{i \mapsto e_0\}\rangle}\,(\text{E.Sub1}) \qquad \frac{\Gamma;\Delta \vdash e^n \rightsquigarrow e^{n'}}{\Gamma;\Delta \vdash \text{for } i < e^n \,.\, g \to \text{for } i < e^{n'} \,.\, g}\,(\text{I.1}) \qquad \frac{\Gamma;\Delta^{+D} \vdash g \rightsquigarrow g'}{\Gamma;\Delta \vdash D \,.\, g \to D \,.\, g'}\,(\text{I.2})$$

$$\frac{\Delta \vdash \texttt{true} \overset{?}{\Rightarrow} e_1 \odot e_2}{\Gamma;\Delta \vdash \mathcal{K}\langle e_1 \odot e_2\rangle \to \mathcal{K}\langle 1\rangle}\,(\text{E.Query}^{\odot}) \qquad \frac{\Gamma;\Delta \vdash c \rightsquigarrow c'}{\Gamma;\Delta \vdash \mathcal{G}\langle c \Rightarrow e\rangle \to \mathcal{G}\langle c' \Rightarrow e\rangle}\,(\text{G.1}) \qquad \frac{\Gamma;\Delta \wedge c \vdash e \rightsquigarrow e'}{\Gamma;\Delta \vdash \mathcal{G}\langle c \Rightarrow e\rangle \to \mathcal{G}\langle c \Rightarrow e'\rangle}\,(\text{G.2})$$

$$\frac{c_1 \wedge \ldots \wedge c_v = \text{CNF}(c_0) \qquad \exists h \in \{1,\ldots,v\} \,.\, \bigwedge_{l=\{1,\ldots,v\}\setminus\{h\}} c_l \overset{?}{\Rightarrow} \neg c_h}{\Gamma \vdash \mathcal{G}\langle c_0 \Rightarrow e\rangle \to \mathcal{G}\langle 0 \Rightarrow e\rangle}\,(\text{G.Falsify}) \qquad \frac{\Gamma;\Delta \wedge c_0 \vdash e_1 \rightsquigarrow e_0}{\Gamma;\Delta \vdash \mathcal{G}\langle (c_0 \Rightarrow e_0) \wedge (c_1 \Rightarrow e_1)\rangle \to \mathcal{G}\langle c_0 \vee c_1 \Rightarrow e_0\rangle}\,(\text{G.Join-1})$$

$$\frac{}{\Gamma;\Delta \vdash c_1 \Rightarrow e_1 \wedge \ldots \wedge 0 \Rightarrow e \wedge \ldots \wedge c_v \Rightarrow e_v \to c_1 \Rightarrow e_1 \wedge \ldots \wedge c_v \Rightarrow e_v}\,(\text{G.Elim})$$

Fig. 11. Rewrite rules for index functions.

$e_2) \wedge (c_3 \Rightarrow e_3)$. Similarly, $\mathcal{K}\langle e\rangle$ denotes the symbol (or expression) obtained by replacing $\square$ with e in the context $\mathcal{K}$. For example, if $\mathcal{K} = e_1 + x_1[\square]$, then $\mathcal{K}\langle x_2\rangle$ is $\mathcal{K} = e_1 + x_1[x_2]$.

Reductions are used in the rewrite rules shown in Fig. 11. I.Sub1 substitutes one index function into another by combining their guarded expressions, given that variables in the indexing expression, e_0, are free. E.Lift treats scalars as one-element arrays, enabling substitution. E.Sub1 substitutes an indexing symbol anywhere given that the index function being indexed into has only one guarded expression. Via rules I.1, I.2, G.1, and G.2, this enables substitutions into domains and Sum symbols, such as $\sum_{j=e_1}^{e_2}(x_1[j])$ where j is not free. Note how in G.2, the truth of the guard is used to rewrite its expression. E.Query$^{\odot}$ uses the solver to simplify symbols beyond the scope of syntactical rewrites, while G.Falsify uses the solver to falsify guards by converting the guard to CNF and then checking whether there exists a false conjunct under assumption of all other conjuncts. G.Join-1 merges two guarded expressions given that the value of the second guarded expression simplifies to the value of the first guarded expression under assumption of the first guard. G.Elim eliminates guarded expressions for which the guard is false. For brevity, we omit analogous rules E.Query$^{\wedge}$ and E.Query$^{\vee}$ where $\odot$ in E.Query$^{\odot}$ is replaced by $\wedge$ and $\vee$, respectively; G.Join-2 where the roles of c_0 and c_1, and e_0 and e_1, are swapped; and rules for binary operations (similar to If).

Returning to `partition2`, we have, via Map, If, Const, E.Lift, and I.Sub1:

$$\texttt{flagsT = map (\c -> if c then 1 else 0) cs} \quad | \quad \text{for } i < n \,.\, p[xs[i]] \Rightarrow 1 \wedge \neg p[xs[i]] \Rightarrow 0$$

Note how the positional relation between xs and the values of $flagsT$ is tracked automatically by substitution on cs. In related systems, these must be established by the user (see Section 7).

Scan: recurrences and sums. Figure 12 extends the rule set to convert `scan` into an index function with a special recurrence symbol $\circlearrowleft$ used for pattern matching in rewrites. Rule I.Sum introduce sums by matching recurrences on linear polynomials where

$$\text{sum}(\text{n}_1 \cdot \text{c}_1 + \text{n}_2 \cdot \text{c}_2 + \cdots + \text{n}_\text{v} \cdot \text{c}_\text{v},\ \text{a},\ \text{b}) = n_1 \cdot \textstyle\sum_{j=a}^{b}(c_1) + \ldots + n_v \cdot \sum_{j=a}^{b}(c_v).$$

G.Neg rewrites negations on symbols found outside guards into integer arithmetic. G.BoolToInt flattens guarded expressions matching user conversion from bool to integer. Both rules facilitate rewriting recurrences into sums over boolean symbols. Converting *indicesT* now yields:

$$\texttt{indicesT = scan (\x y -> x + y) 0 flagsT} \quad | \quad \text{for } i < n \,.\, true \Rightarrow \textstyle\sum_{j=0}^{i}(p[xs[j]])$$

Rules for scan and scatter

$$\boxed{\Gamma;\Delta \vdash e^* \overset{Src}{\rightarrow} (\Gamma',\Delta',f) \qquad \Gamma;\Delta \vdash f \rightarrow f' \qquad \Gamma;\Delta \vdash g \rightarrow g'}$$

$$\frac{\Gamma(x_3) = \text{for } i < e^n \,.\, g_3 \qquad \Gamma;\Delta^{+\text{for } i<e^n} \vdash e^*\{x_2 \mapsto x_3[i]\} \rightsquigarrow (\Gamma',\Delta' \text{for } \bullet \,.\, g_1)}{\Gamma;\Delta \vdash \texttt{scan}\ (\lambda x_1\ x_2.\ e^*)\ v\ x_3 \overset{Src}{\rightarrow} (\Gamma',\Delta, \text{for } i < e^n \,.\, g_1\{x_1 \mapsto \circlearrowleft\})}\ (\textsc{Scan})$$

$$\frac{(e \text{ is linear and } \circlearrowleft \text{ does not occur in } e) \quad \text{fresh } j \quad e' := \text{sum}(e\{i \mapsto j\},\ 0,\ i)}{\Gamma;\Delta \vdash \text{for } i < e^n \,.\, 1 \Rightarrow \circlearrowleft + e \rightarrow \text{for } i < e^n \,.\, 1 \Rightarrow e'}\ (\textsc{I.Sum}) \qquad \frac{(\circlearrowleft \text{ does not occur in } e)}{\Gamma;\Delta \vdash \text{for } i < e^n \,.\, (i = 0 \Rightarrow e) \bigwedge (i \neq 0 \Rightarrow \circlearrowleft) \rightarrow \text{for } i < e^n \,.\, 1 \Rightarrow e\{i \mapsto 0\}}\ (\textsc{I.Carry})$$

$$\frac{}{\Gamma;\Delta \vdash \mathcal{G}\langle c_1 \Rightarrow \mathcal{K}\langle \neg c_2\rangle\rangle \rightarrow \mathcal{G}\langle c_1 \Rightarrow \mathcal{K}\langle 1 - c_2\rangle\rangle}\ (\textsc{G.Neg})$$

$$\frac{}{\Gamma;\Delta \vdash (c_0 \Rightarrow n_0) \bigwedge (c_1 \Rightarrow n_1) \rightarrow (true \Rightarrow c_0 \cdot n_0 + c_1 \cdot n_1)}\ (\textsc{G.BoolToInt})$$

$$\frac{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Safe}{\rightarrow} \textbf{true} \qquad e^n = \text{length } xs \qquad (\text{Img},\Delta'') = \begin{cases} (X,\Delta) & \text{if } \Delta.Bij(is) = (\text{Rcd},X) \text{ and } \Delta \vdash \texttt{true} \overset{?}{\Rightarrow} X \subseteq [0,e^n) \subseteq \text{Rcd} \\ ([0,e^n),\Delta') & \text{if } \Gamma;\Delta \vdash (is,\ [0,e^n),\ (\_,[0,e^n))) \overset{Bij}{\rightarrow} (\textbf{true},\Delta') \end{cases}}{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Src}{\rightarrow} (\Gamma,\Delta'', \text{for } i < e^n \,.\, (i \in \text{Img} \Rightarrow vs[is^{-1}[i]]) \bigwedge (i \notin \text{Img} \Rightarrow xs[i]))}\ (\textsc{Sc1})$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Safe}{\rightarrow} \textbf{true} \qquad e^n = \text{length } xs \qquad e^m = \text{length } is \\ f_{is} = \text{for } i < e^m \,.\, (0 \leq is[i] < e^n \Rightarrow is[i]) \bigwedge (\neg(0 \leq is[i] < e^n) \Rightarrow \infty) \\ \Gamma;\Delta \vdash f_{is} \rightsquigarrow (\Gamma',\Delta', \text{for } i < e^m \,.\, (c \Rightarrow e)[\bigwedge(\neg c \Rightarrow \infty)]) \qquad e_0 = e\{i \mapsto 0\} \qquad \text{fresh } k_1,k_2 \\ \Delta' \wedge 0 \leq k_1 < k_2 < e^m \vdash \textbf{true} \overset{?}{\Rightarrow} 0 \leq e\{i \mapsto k_1\} \leq e\{i \mapsto k_2\} \leq e^n \qquad \text{fresh } j,k \quad e_k = e\{i \mapsto k-1\} \quad c_k = c\{i \mapsto k-1\}\end{array}}{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Src}{\rightarrow} (\Gamma',\Delta', \text{for } i < e_0 \,.\, (\texttt{true} \Rightarrow xs[i]) \ \cup \bigcup_{k=1}^{e^m} \text{for } j \geq e_k \propto e^n \,.\, (j = e_k \wedge c_k \Rightarrow vs[k-1]) \bigwedge (j \neq e_k \vee \neg c_k \Rightarrow xs[j])}\ (\textsc{Sc2})$$

Scatter safety

$$\boxed{\Gamma;\Delta \vdash f \overset{Safe}{\rightarrow} (\textbf{true},\Delta')}$$

$$\frac{\Gamma;\Delta \vdash ([0,\text{length } xs),\ is) \overset{Inj}{\rightarrow} (\textbf{true},\Delta')}{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Safe}{\rightarrow} \textbf{true}}\ (\textsc{Ss1}) \qquad \frac{\Gamma(vs) = \text{for } i < e^n \,.\, \texttt{true} \Rightarrow e \quad i \notin \mathcal{FV}(e)}{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Safe}{\rightarrow} \textbf{true}}\ (\textsc{Ss2})$$

$$\frac{\Delta.Range(vs) = (e_1,e_2) \qquad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} e_1 = e_2}{\Gamma;\Delta \vdash \texttt{scatter}\ xs\ is\ vs \overset{Safe}{\rightarrow} \textbf{true}}\ (\textsc{Ss3})$$

Fig. 12. Additional rewrite rules as well as conversion of `scan` and `scatter`.

Scatter: safety checks and segmented domains. Figure 12 converts `scatter` to index functions while verifying safety; if safety cannot be shown, no rule will be matched and the analysis terminates in failure. The two conversion rules Sc1 and Sc2 first check safety using Ss1–3. Ss1 verifies that the indices *is* have no duplicates in the bounds of the destination array *xs*, and Ss2 and Ss3 both verify that the values in *vs* do not depend on its indices (allowing for duplicate indices in *is*). Sc1 further verifies that the indices *is*, that fall within the bounds of *xs*, are a permutation of the indices of *xs*. The resulting index function couples is^{-1} to *is*, which is used in query solving (e.g., FPV2). Sc2 verifies that *is* is monotonically increasing, producing an index function that has a domain with $e^m + 1$ segments (per Section 2.2). Out-of-bounds indices map to empty segments; this is crucial, because the number of non-empty segments is generally statically unknown. The resulting index function is verbose, but simplifies via the rewrite system. For example, if e_0 rewrites to 0, the leading for $i < e_0$. $(true \Rightarrow xs[i])$ simplifies away (its domain is empty)—this is the case when

Substituting index functions $\boxed{\Gamma; \Delta \vdash f \to f'}$

$$\frac{\Gamma(x) = \bigcup_{k=0}^{e^m} \text{for } j \geq e_k \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \qquad \forall h = 1, \ldots, v \,.\, \Delta^{+\bigcup_{k=0}^{e^m} \text{for } i \geq e_k} \vdash c_h \overset{?}{\Rightarrow} e_k \leq e_0 < e_k\{k \mapsto k+1\} \qquad \mathcal{FV}(e_0) \cap \mathcal{BV}(\mathcal{K}\langle x[e_0]\rangle) = \emptyset \quad \Gamma; \Delta \vdash e_k\{k \mapsto m+1\} \rightsquigarrow_s e_n \quad \sigma = \{j \mapsto e_0\}}{\Gamma; \Delta \vdash \text{for } i < e_n \,.\, \mathcal{G}\langle c \Rightarrow \mathcal{K}\langle x[e_0]\rangle\rangle \to \bigcup_{k=0}^{e^m} \text{for } i \geq e_k \,.\, \mathcal{G}\langle \bigwedge_{h=1}^{v} (c \wedge \sigma(c_h) \Rightarrow \mathcal{K}\langle \sigma(e_h)\rangle)\rangle} \text{(Sub2)}$$

$$\frac{\Gamma(x) = \bigcup_{k'=0}^{e^{m'}} \text{for } j \geq e_{k'} \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \qquad \forall h = 1, \ldots, v \,.\, \Delta^{+\bigcup_{k=0}^{e^m} \text{for } i \geq e_k} \vdash c_h \overset{?}{\Rightarrow} e_k \leq e_0 < e_k\{k \mapsto k+1\} \qquad \mathcal{FV}(e_0) \cap \mathcal{BV}(\mathcal{K}\langle x[e_0]\rangle) = \emptyset \quad \bigcup_{k=0}^{e^m} \text{for } i \geq e_k \text{ unifies with } \bigcup_{k=0}^{e^{m'}} \text{for } i \geq e'_k \quad \sigma = \{k' \mapsto k, j \mapsto e_0\}}{\Gamma; \Delta \vdash \bigcup_{k=0}^{e^m} \text{for } i \geq e_k \,.\, \mathcal{G}\langle c \Rightarrow \mathcal{K}\langle x[e_0]\rangle\rangle \to \bigcup_{k=0}^{e^m} \text{for } i \geq e_k \,.\, \mathcal{G}\langle \bigwedge_{h=1}^{v} (c \wedge \sigma(c_h) \Rightarrow \mathcal{K}\langle \sigma(e_h)\rangle)\rangle} \text{(Sub3)}$$

$$\frac{\Gamma(x) = \bigcup_{k=0}^{e^m} \text{for } j \geq e_k \,.\, \bigwedge_{h=1}^{v} (c_h \Rightarrow e_h) \qquad \{k \mapsto e'_0\} = \text{unify}\,(e_k,\ e_0) \quad \textbf{or} \quad \{k \mapsto e'_0\} = \text{unify}\,(e_k\{k \mapsto k+1\} - 1,\ e_0) \qquad \mathcal{FV}(e_0) \cap \mathcal{BV}(\mathcal{K}\langle x[e_0]\rangle) = \emptyset \quad \sigma = \{k \mapsto e'_0, j \mapsto e_0\}}{\Gamma; \Delta \vdash \text{for } D \,.\, \mathcal{G}\langle c \Rightarrow \mathcal{K}\langle x[e_0]\rangle\rangle \to \text{for } D \,.\, \mathcal{G}\langle \bigwedge_{h=1}^{v} (c \wedge \sigma(c_h) \Rightarrow \mathcal{K}\langle \sigma(e_h)\rangle)\rangle} \text{(Sub4)}$$

Fig. 13. Substitution rules for segmented domains.

scattering ones into a flag array using an exclusive scan of the shape array (Fig. 2). It also handles scattering at an inclusive scan's output, forming an initial linear segment. A subsequent segmented scan with either flag array simplifies to the same index function. A further simplification omits c_k from the guards if c is implied by the segment being non-empty, checked via the queries

$$\Delta^{+\text{for } i<e^m} \vdash e\{i \mapsto i+1\} - e > 0 \overset{?}{\Rightarrow} c \quad \text{and} \quad \Delta \wedge i = e^m - 1 \vdash e^n - e > 0 \overset{?}{\Rightarrow} c.$$

Substitutions of segmented index functions into other index functions can be reduced to Sub1 by first transforming the index function to have a linear domain using an *II* array (or by treating k as an uninterpreted function of the domain iterator i). However, we define additional substitutions in Fig. 13 to automatically propagate structural information through index functions. Sub2 is similar to Sub1, but replaces the linear domain of the target index function with the segmented domain of the function being indexed into. A query checks that e_0 is within bounds of segment k, allowing the segmented domain to be adopted without modification. Sub3 further requires that the domains of two segmented index functions are identical for substitution to go through. Finally, Sub4 eliminates the iterator variable, k, of the segmented index function being indexed into, given unification of e_k with e_0 yields a substitution for k. This is useful for scalar expressions that index into segmented index functions, in which case the domain, naturally, cannot be propagated. If solving fails, k can always be replaced by indexing into an *II* array corresponding to the segmented index function. Segmented versions of Map, Scan, I.1, and I.Sum follow similar ideas and are ommited for brevity.

Rewrite system and final remarks. When converting the source language to index functions, each conversion step is followed by rewrites applied to a fixed point:

$$\Gamma; \Delta \vdash e^* \overset{Src}{\to} (\Gamma', \Delta', f), \quad \Gamma; \Delta \vdash f \rightsquigarrow (\Gamma'', \Delta'', f'),$$

Rewrite rules are simply tried in the order that they appear here, except for I.Sum and I.Carry, which are only applied after Let as they introduce indexing with bound variables and hence limit

$$
\begin{array}{rcll}
X & ::= & x & \text{Non-Boolean Array} \\
 & | & DOR\ \overline{x} & \text{Disjoint Bool Arrays} \\
s & ::= & x & \text{Variable} \\
 & | & X[Poly^{ind}] & \text{Indexing} \\
 & | & \sum X[Poly^{b} : Poly^{e}] & \text{Slice Sum} \\
Term & ::= & s^{k} \mid s^{k} * Term & \\
Poly & ::= & k \mid Poly + k * Term &
\end{array}
$$

Example of Legal Ranges

$$
\begin{array}{ccc}
3 & \leq n \leq & \text{infty} \\
n & \leq 3 \cdot i \leq & \{5 \cdot n, n^2\} \\
i+n & \leq j \leq & i^2 \\
\{i+1, 5\} & \leq \sum X[i : i+j] \leq & j-1
\end{array}
$$

Example of Illegal Ranges

$$
\begin{array}{ccc}
i & \leq n \leq & i \cdot i \\
n & \leq 2 \cdot i \leq & 2 \cdot n - 5
\end{array}
$$

Algorithm 1 SOLVE$^{\leq 0}_{\Delta}$

Require: a $Poly$ expression $\mathbf{p}$ in which monotonic indices have been translated to sums.
Ensure: **true** if $\mathbf{p} \leq 0$ was verified, else **false**

```
p' = PeelOnRng_Δ(Simplify_Δ p)
if p' is constant k then return k ≤ 0
s = FindSym_Δ p'
(lbs, k, ubs) = getRange_Δ s
rewrite p' as p_a · s + p_b
for each (l, u) ∈ lbs × ubs do
    (p1, p2) = (l · p_a + k · p_b, u · p_a + k · p_b)
    if ( Solve^{≤0}_Δ (−p_a) ∧ Solve^{≤0}_Δ p2 ) ∨
       ( Solve^{≤0}_Δ p_a ∧ Solve^{≤0}_Δ p1 ) then
        return true
return false
```

Algorithm 2 SIMPLIFY$_{\Delta}$

Require: a $Poly$ expression
Ensure: a $Poly$ expression semantically equivalent with the input

```
while Fix Point Not Reached do
    Apply SubstEquivs_Δ
    Apply 0Sum
    Apply to a fix point UnBef
    Apply to a fix point B1-5
    Apply 0Sum
    Apply to a fix point UnAft1-5
return fix-point expression
```

Fig. 14. Grammar for symbols (s), polynomial ($Poly$) and guarded expressions (g), and index functions $ixfn$.

substitutions. The source program is converted top-to-bottom; top-level functions must be declared before using them. This means all top-level functions have index functions before they are applied. Thus application of function declarations (1) verifies that the actual arguments' index functions satisfy the formal arguments' pre-conditions; (2) substitutes formal arguments with actual ones via the indexing reduction rules; and (3) adds the previously verified post-condition to Δ.

5 SOLVING QUERIES

The query solver has three main tasks: to solve equality and inequality queries and to simplify internal expressions, e.g., enabling the side conditions related to unification. Section 5.1 discusses the language of the query solver and the construction of symbol tables such as the ones recording symbols' ranges *Ranges* and equivalences *Equivs*. Section 5.2 presents our adaptation of the Fourier-Motzkin algorithm for solving inequalities, and Section 5.3 presents the simplification engine that is aimed to support array indexing and sum symbols. While equality queries are often satisfied through simplification, Section 5.4 presents a more advanced method that exploits injective properties.

5.1 Query Solver Language Lowering and Symbol Tables

The left side of Fig. 14 shows the language of the query solver. Capital X denotes either the name of an integral array x or a disjunction of a sequence of disjoint/orthogonal boolean arrays, denoted DOR $\overline{x}^{v}$, i.e., for any legal index i, (DOR $\overline{x}^{v}$)$[i] = x_1[i] \vee \ldots \vee x_v[i]$ *and* there is at most one $h \in 0 \ldots v$ holding a true value. A singleton sequence is always legal, and an empty sequence results in false. Computationally, boolean values are treated as integers, 1 for true and 0 for false.

The solver uses a polynomial representation similar to the one of index functions, denoted $Poly$, except that its symbols are restricted to scalar variables x, indexing and sums of array slices (of unit stride) over X. High-level queries are lowered to $Poly$ in the standard way, by binding (bilaterally) untranslatable expressions to fresh variables, whose properties are documented, as much as possible, in other symbol tables. For example, the guards $\overline{c}^{v}$ of an index function give raise to v fresh symbols $\overline{x}^{v}$, where each one of them is bound in the $\Delta.DOR$ symbol table to the whole sequence $\overline{x^{v}}$. While the exact meaning of the predicates is lost, the solver can still conduct reasoning based on the property that at any index, exactly one of $\overline{x}^{v}$ is true, which subsumes disjointness.

Similarly, for the purpose of inequality solving, a monotonically-increasing array x of length n is translated to the sum of a slice of a fresh array variable a, whose elements are positive. Assuming that x's elements are known to be within $[e^{lb}, e^{ub}]$, then a valid index $x[i]$ is translated to $e^{lb} + \sum a[0 : i]$,

while encoding in *Ranges* that a has positive elements (and $\sum a[0:n-1] \leq e^{ub} - e^{lb}$). Strictly-increasing monotonicity is similarly handled by means of a strictly positive array.

The *Ranges* and *Equiv* symbol tables are constructed to conform with a "triangular" shape that essentially requires that an ordering of the bound symbols exists, such that the range or equivalent rewrite of a symbol may only depend on its predecessors. Bindings $sym \mapsto bnd$ are added by processing boolean expressions denoting (in)equalities, e.g., corresponding to code properties (e.g., branch conditions or loop counters) or query premises. From them, a set of candidate bindings are identified and the invalid ones are rejected, whenever it is found that the leading name of sym appears in the transitive closure of variable names through *Ranges* and *Equiv* that starts from bnd. If there are several legal candidates for the same (in)equality, the winning symbol is selected by a set of heuristics, e.g., the one appearing latest in program order. For example, assuming an empty *Equiv*, the binding $i \mapsto B[i]$ is illegal because $i \in \{i, B\}$, but $B[i] \mapsto i$ is legal because the leading symbol $B \notin \{i\}$. Examples of legal and illegal *Ranges* tables are also shown in Fig. 14.

5.2 Solving Inequalities

Inequalities are reduced to form $p \leq 0$ and solved by the adaptation of Fourier-Motzkin elimination [21, 66] presented in Algorithm 1, which proceeds by simplifying p, which is necessary in the presence of index and sum symbols, otherwise the canonical polynomial representation *Poly* is optimal. The next line implements the base case, i.e., when the simplified expression p' is a value k. If not, FindSym$_\Delta$ determines the next symbol s to eliminate, as the symbol of p' whose range transitively depends on the largest number of (other) distinct symbols. The range of s, computed by GetRange$_\Delta$, takes the form $(\overline{lbs}, k, \overline{ubs})$, where $k > 0$ is a constant and $\overline{lbs}$ and $\overline{ubs}$ are expression sequences having the semantics $\texttt{max}(\overline{lbs}) \leq k \cdot s \leq \texttt{min}(\overline{ubs})$. GetRange$_\Delta$ looks up the ranges recorded in *Ranges*, e.g., an index $x[i]$ may possibly have several lower/upper bounds originating in program branches or query premises, which are further expanded with the range of x's elements, if x has one. As well, GetRange$_\Delta$ infers ranges, e.g., if array x is strictly positive, then a lower bound for $\sum x[lb:ub]$ is $ub - lb + 1$, if the later can be proven positive, and similar for the upper bound of $\sum(\text{DOR}\ \overline{x})[lb:ub]$.

Finally, $p' \leq 0$ is re-written as $p_a \cdot (k \cdot s) + k \cdot p_b \leq 0$, such that $s \notin p_b$, and $k \cdot s$ is conservatively replaced—i.e., with its upper bound if $p_a \geq 0$ and with its lower bound otherwise—generating two sub-problems for each combination of lower and upper bounds from the sequence. If any succeeds the query has been verified, otherwise the result is unknown. Our adaptation allows polynomial ranges and polynomial queries, as well; for example, assuming the valid *Ranges* in Fig. 14, the query $n^2 + 3 \cdot n - j^2 - j \leq 0$ succeeds. The first eliminated symbol is j, since its ranges transitively depend on both i and n, and the next one is i. Note that if n is chosen as the first symbol to eliminate, the query fails, immediately resulting in $\infty \leq 0$. Solve$^{\leq 0}_\Delta$ is guaranteed to terminate because (1) the "triangular" shape of *Ranges* and *Equivs* do not introduce cycles, and (2) at every step, the target symbol will be eliminated in a number of steps equal to its polynomial degree, which is finite.

5.3 Simplifying Index and Sum-of-Slice Expressions

We motivate our technique on one of the simpler queries generated by `partition2`, namely: $j + \sum C[j+1:n-1] - \sum C[0:i-1] > 0$, where the context is $Ranges = \{\ 0 \leq n \wedge 0 \leq i \leq n-1 \wedge 0 \leq j \leq i-1\}$ and $Equivs = \{C[i] = 1 \wedge C[j] = 0\}$ and $C = \text{DOR}\ c$ has elements in the implicit range $[0, 1]$. Note that (1) *Equivs* cannot be applied because $C[i]$ and $C[j]$ do not appear in the sums, and (2) the sum subtraction cannot be simplified because it cannot be proven that the two slices overlap. Applying Algorithm 1 will end in a clearly false subquery $j > i$.

Unary Simplification $\overset{UnS}{\rightarrow}$ *uses helper* $\overset{Eqv}{\rightarrow}$ *to rewrite symbol* s *as Poly-expression* p. $\boxed{\Delta \vdash s \overset{UnS}{\rightarrow} p}$

$$\frac{\Delta.Equiv(s) = p}{\Delta \vdash s \overset{Ev}{\rightarrow} p}\ (\textsc{Eqv1}) \qquad \frac{s = (\text{DOR}\ \overline{x}^{v})[p^{ind}] \quad \exists i \in 1 \ldots v \mid \Delta.Equiv(x_i[p^{ind}]) = 1}{\Delta \vdash s \overset{Eqv}{\rightarrow} 1}\ (\textsc{Eqv2}) \qquad \frac{s = \sum x[p^b : p^e] \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^b > p^e}{\Delta \vdash s \overset{UnS}{\rightarrow} 0}\ (\textsc{0Sum})$$

$$\frac{s = \sum X[p^b\ :\ p^e] \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^e + 1 \geq p^b \quad \Delta \vdash X[p^b - 1] \overset{Eqv}{\rightarrow} p_b^x \quad s' = \sum x[p^b - 1 : p^e]}{\Delta \vdash s \overset{unS}{\rightarrow} s' - p_b^x}\ (\textsc{UnBef}) \qquad \frac{s = \sum x[p^b : p^e] \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^b = p^e}{\Delta \vdash s \overset{UnS}{\rightarrow} X[p^b]}\ (\textsc{UnAft1}) \qquad \frac{\Delta \vdash s \overset{Eqv}{\rightarrow} p}{\Delta \vdash s \overset{UnS}{\rightarrow} p}\ (\textsc{UnAft2})$$

$$\frac{s = \sum X[p^b\ :\ p^e] \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^e \geq p^b \quad \Delta \vdash X[p^b] \overset{Eqv}{\rightarrow} p_b^x \quad s' = \sum x[p^b + 1 : p^e]}{\Delta \vdash s \overset{UnS}{\rightarrow} s' + p_b^x}\ (\textsc{UnAft3}) \qquad \frac{s = \sum X[\_ : \_] \ \textbf{OR}\ s = X[\_] \quad X = \text{DOR}\ \overline{x}^{v} \quad v = 0}{\Delta \vdash s \overset{UnS}{\rightarrow} 0}\ (\textsc{UnAft4}) \qquad \frac{s = \sum X[p^b\ :\ p^e] \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^e \geq p^b \quad \Delta \vdash X[p^e] \overset{Rng}{\rightarrow} \_ \quad s' = \sum x[p^b : p^e - 1]}{\Delta \vdash s \overset{UnS}{\rightarrow} s' + X[p^e]}\ (\textsc{PeelOnRng})$$

Rewrites a sum of two terms $k_1 \cdot s_1 \cdot t_1 + k_2 \cdot s_2 \cdot t_2$ *as a Poly* p. $\boxed{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} p}$

$$\frac{s_1 = \sum X_1[p_1^b\ :\ p_1^e] \quad s_2 = \sum X_2[p_2^b\ :\ p_2^e] \quad X_1 = X_2 \quad k_1 = k_2 \quad t_1 = t_2 \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p_1^e + 1 = p_2^b \quad (\Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p_1^e + 1 \geq p_1^b \ \textbf{OR}\ \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p_2^e + 1 \geq p_2^b)}{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} k_1 \cdot t_1 \cdot \sum X_1[p_1^b : p_2^e]}\ (\text{B1})$$

$$\frac{s_1 = \sum X_1[p^b\ :\ p^e] \quad s_2 = X_2[p^i] \quad X_1 = X_2 \quad k_1 = k_2 \quad t_1 = t_2 \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^e + 1 \geq p^b \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p^i = p^e + 1 \quad s' = \sum X_1[p^b\ :\ p^e + 1]}{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} k_1 \cdot s' \cdot t_1}\ (\text{B2})$$

$$\frac{k_1 = k_2 \quad t_1 = t_2 \quad s_1 = \sum(\text{DOR}\ \overline{x}^{v})[p_x^b : p_x^e] \quad s_2 = \sum(\text{DOR}\ \overline{y}^{w})[p_y^b : p_y^e] \quad y_1 \in \Delta.DOR(x_1) \quad \overline{x} \cap \overline{y} = \emptyset \quad \overline{z} = \overline{x} \cup \overline{y} \quad v > 0 \quad w > 0 \quad (s_2', s_3') = (\sum(\text{DOR}\ \overline{z})[p_y^b : p_x^e],\ \sum(\text{DOR}\ \overline{y})[p_x^e + 1 : p_y^e]) \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p_x^b \leq p_y^b \leq p_x^e \leq p_y^e}{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} k_1 \cdot t_1 \cdot \sum(\text{DOR}\ \overline{x})[p_x^b : p_y^b - 1] \ +\ k_1 \cdot s_2' \cdot t_1 \ +\ k_1 \cdot s_3' \cdot t_1}\ (\text{B3})$$

$$\frac{k_1 = -k_2 \quad t_1 = t_2 \quad s_1 = \sum X[p_x^b\ :\ p_x^e] \quad s_2 = \sum Y[p_y^b\ :\ p_y^e] \quad X = Y \quad (s_1', s_2') = (\sum X[p_x^b : p_y^b - 1],\ \sum X[p_y^e + 1 : p_x^e]) \quad \Delta \vdash \textbf{true} \overset{?}{\Rightarrow} p_x^b \leq p_y^b \leq p_y^e \leq p_x^e}{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} k_1 \cdot s_1' \cdot t_1 + k_2 \cdot s_2' \cdot t_2}\ (\text{B4})$$

$$\frac{k_1 = -k_2 \quad t_1 = t_2 \quad s_1 = \sum(\text{DOR}\ \overline{x})[p_x^b : p_x^e] \quad s_2 = \sum(\text{DOR}\ \overline{y})[p_y^b : p_y^e] \quad \overline{z} = \overline{x} \cap \overline{y} \quad \overline{z} \neq \emptyset \quad \overline{x'} = \overline{x} - \overline{w} \quad \overline{y'} = \overline{y} - \overline{w} \quad (y_1 = x_1 \ \textbf{OR}\ y_1 \in \Delta.DOR(x_1)) \quad Z = \text{DPR}\ \overline{z} \quad \Delta \vdash ((k_1,\ Z[p_x^b : p_x^e], t_1),\ (k_2,\ Z[p_y^b : p_y^e],\ t_2)) \overset{BinS}{\rightarrow} p^z}{\Delta \vdash ((k_1, s_1, t_1), (k_2, s_2, t_2)) \overset{BinS}{\rightarrow} k_1 \cdot t_1 \cdot \sum(\text{DOR}\ \overline{x'})[p_x^b : p_x^e] \ +\ k_2 \cdot t_2 \cdot \sum(\text{DOR}\ \overline{y'})[p_y^b : p_y^e] \ +\ p^z}\ (\text{B5})$$

Fig. 15. Rules for Unary and Binary Simplification of Expressions in Polynomial Representation.

Our strategy is organized in three steps: The first is to extend the ends of the summed slices with the indices that have bindings in *Equiv*. This enables the second step, which perform simplifications across sum-sum or sum-index pairs. Finally, the indices that have a binding in *Equivs* are separated from sums and substituted with their rewrite. With our example, the first step will result in $j + \sum C[j : n-1] - \sum C[0 : i] + 1 > 0$. Since both slices are now provably overlapping, i.e., context says $i > j$, the common part can be eliminated, resulting in $j + \sum C[i+1 : n-1] - \sum C[0 : j-1] + 1 > 0$. Applying Algorithm 1 will first replace[6] $\sum C[0 : j-1]$ with its upper bound j then will replace $\sum C[i+1 : n-1]$ with its lower bound 0, resulting in $j - j + 1 > 0$, which succeeds after simplification.

[6] $\sum C[0 : j-1]$ has upper bound j, resulting in transitive closure $\{i, j, n\}$, hence it is the most "dependent" symbol.

Figure 15 shows the inference rules of the simplification engine, which are composed as depicted in Algorithm 2 (SIMPLIFY$_\Delta$). Unary simplifications replace a symbol with an equivalent expression. Helper rules EQV1 and EQV2 replace a symbol with its binding in *Equiv*, and a DOR index with 1 if one of the arrays in the disjunction has 1 as the binding of that index. 0SUM replaces sums of provably empty slices with 0. UNBEF extends a slice sum with an ending index that is bound in *Equivs*. It requires that the slice is provably potentially empty in a nice way *PENW*, i.e., its lower bound is at most 1 higher than its upper bound, which guarantees that the extended slice contains said element. SIMPLIFY's first stage consists of applying the equivalences in *Equivs* (SUBSTEQUIVSΔ), 0SUM, and a fixed-point application of UNBEF on any matching symbols of degree one.

The second stage consists of applying the binary rules B1-5 to a fix point. They denote an equivalent rewrite p across any (sum of) two terms $k_{1,2} \cdot s_{1,2} \cdot t_{1,2}$ of a polynomial p^{tgt}, where $k_{1,2}$ are constants, $s_{1,2}$ are symbols of degree 1, and $t_{1,2}$ are a product of symbols not containing $s_{1,2}$.

The rewrite of p^{tgt} is thus $p^{tgt} - k_1 \cdot s_1 \cdot t_1 - k_2 \cdot s_2 \cdot t_2 + p$. B1 rewrites two sums of slices that are in continuation of each other as one sum, and requires that at least one of them is *PENW*. B2 extends a slice sum with an end index. B3 simplifies two sums of overlapping slices corresponding to DOR arrays, whose sequences of variable names do not overlap, but are in the same $\Delta.DOR$ class. B4 eliminates the common part of two overlapping slice sums, corresponding to the same array, that are subtracted, and B5 extends this simplification of slice-sum subtraction to DOR arrays. Two additional rules (not shown), analogous to B3 and B4, handle the cases when $p^b_x \leq p^b_y \leq p^e_y \leq p^e_x$ and $p^b_x \leq p^b_y \leq p^e_x \leq p^e_y$, respectively.

The final pass applies 0SUM followed by a fix-point application of unary rules UNAFT1-6. They collapse a singleton slice to an index (UNAFT1), replace symbols bound in *Equivs* (UNAFT2), peel-off an index bound in *Equivs* from an end of a slice sum (UNAFT3), and eliminate null DOR slices/indices. Finally, PEELONRNG, not part of simplification, peels off an end-of-slice-sum index that has a more specialized range than the one of the whole array, which improves the accuracy of SOLVE$_\Delta$.

5.4 Exploiting Injective Properties with the Equality Solver

Queries $\Delta \vdash c_1 \overset{?}{\Rightarrow} c_2$ are discharged by the equality solver through syntactical rewrites on the internal language (Fig. 3). First the antecedent of the query, c_1, is expanded with its transitive equalities. Then, the transitive equalities are rewritten using properties found in Δ. This expanded query is sent to the inequality solver (which may solve $s_1 == s_2$ as the queries $s_1 \geq s_2 \wedge s_1 \leq s_2$.)

More specifically, c_1 is converted to CNF yielding c_1', and the first conjunct is chosen to be the *guide equation*. The right hand side of the guide equation is then substituted for the left hand side everywhere in c_1'. Transitive equalities are then extracted by treating each equality as edges in a graph (symbols in c_1 being the nodes) and then computing a spanning forest using depth-first search. Finally, transitive equalities are conjoined with c_1' and the syntactical rewrites are applied. For example, a rule exploiting injectivity is:

$$\frac{\text{Rcd} = \Delta.Inj(x) \qquad \Delta \vdash \texttt{true} \overset{?}{\Rightarrow} i \in \text{Rcd} \wedge j \in \text{Rcd}}{\Delta \vdash \mathcal{K}\langle x[i] = x[j]\rangle \rightarrow i = j} \text{(EQ.INJ)}$$

In rules such as INJGE the appropriate guide is obvious (equality on the values of the guarded expressions); if there is no obvious choice, the guides can be tried exhaustively.

```
def filter_by [n] 't (cs: [n]bool) (xs: [n]t)
  : []t | \ys -> FiltPart ys xs (\i -> cs[i]) (\_ -> true) =
  Analogous to partition2 in Fig. 2; uses scatter.

def get_smallest_pairs [n]
    (n_verts: i64) (n_es: i64)
    (es: [n]i64 | Range es (0, n_verts))
    (is: [n]i64 | Inj is (−∞,∞))
    : ([]i64, []i64) | \(es', is') ->
        Inj es' (−∞,∞) && Inj is' (−∞,∞) =
  let H = hist i64.min n_es n_verts es is
  let cs = map2 (\i j -> H[i] == j) es is
  let xs = filter_by cs es
  let ys = filter_by cs is
  in (xs, ys)
```

```
def kmeans_ker [num_cols] [nnz] [n]
    (row: i64 | Range row (0,n))
    (pointers: [n+1]i64 | Range pointers (0,nnz))
    (cluster: [num_cols]f32) (values: [nnz]f32)
    (indices: [nnz]i64 | Range indices (0,num_cols))
    : f32 =
  let index_start = pointers[row]
  let nnz_sgm = pointers[row+1] − index_start
  in loop (correction) = (0) for j < nnz_sgm do
      let element_value = values[index_start+j]
      let column = indices[index_start+j]
      let cluster_value = cluster[column]
      let diff = element_value − 2 * cluster_value
      let res = correction + diff*element_value
      in res
```

Fig. 16. Compute kernels from Maximal Matching (left) and sparse k-means (right).

6 EVALUATION

We illustrate the practicality of our system through three case studies.

Statically safe scatters. We verify Futhark's maximal matching benchmark,[7] which implements a graph algorithm from the Problem Based Benchmark Suite [1] that iteratively filters graph edges using `scatter` operations. The most challenging kernel to verify, `get_smallest_pairs`, is shown in Fig. 16. The postcondition says that the values in each output array `es'` and `is'` are unique. We automatically verify both of these properties using the injectivity of `is`. After index function inference, we have $\Gamma(es) = \text{for } i < n \,.\, true \Rightarrow es[i]$ and $\Delta.Inj(is) = (-\infty, \infty)$ in the environment. The postcondition `Inj es' (−∞,∞)` starts a proof query, which will match first InjV2 and then InjF1. InjF1 attempts two rewrites matching InjGE and MonGe, respectively. InjGe succeeds:

$$\dfrac{\dfrac{\dfrac{\Delta' \vdash es[i] = es[j] \wedge H[is[es[i]]] = is[i] \wedge H[is[es[j]]] = is[j] \overset{?}{\Rightarrow} i = j}{(\text{fresh } j) \quad \Delta \wedge 0 \le j < n \wedge 0 \le i < n \vdash_{i,j} (H[is[es[i]]] = is[i] \Rightarrow es[i]) \overset{Inj=}{\rightarrow} \textbf{true}}\,(\textsc{InjGe})}{\Delta \vdash \text{for } i < n \,.\, H[is[es[i]]] = is[i] \Rightarrow es[i] \overset{Inj}{\rightarrow} \textbf{true}}\,(\textsc{InjF1})}{\Gamma; \Delta \vdash \text{for } i < n \,.\, \begin{array}{l}(c \wedge es[i] \in (-\infty,\infty) \Rightarrow es[i]) \\ \bigwedge(\neg c \vee es[i] \notin (-\infty,\infty) \Rightarrow \infty)\end{array} \rightsquigarrow \text{for } i < n \,.\, H[is[es[i]]] = is[i] \Rightarrow es[i]}$$

$$\dfrac{\Delta.FP(es') = (es, \lambda i.\ \overbrace{H[is[es[i]]] = is[i]}^{c}, \_)}{\Gamma; \Delta \vdash ((-\infty,\infty), es') \overset{Inj}{\rightarrow} (\textbf{true}, \Delta)}\,(\textsc{InjV2})$$

The query ($\overset{?}{\Rightarrow}$) antecedent is sent to the equality solver with guide $es[i] = es[j]$, which is substituted into the other terms before expanding the antecedent with transitive equalities:

$$es[i] = es[j] \wedge H[is[es[i]]] = is[i] \wedge H[is[es[i]]] = is[j] \wedge is[i] = is[j]$$

Eq.Inj is then matches and simplifies the conjunct $is[i] = is[j]$ to $i = j$ since $\Delta.Inj(is) = (-\infty, \infty)$. Finally, the expanded (now trivial) query is sent to the solver: $\Delta' \vdash es[i] = es[j] \wedge \ldots \wedge i = j \overset{?}{\Rightarrow} i = j$.

Obviating dynamic bounds checks. We have verified that indexing is within bounds for a key computational kernel from the sparse k-means benchmark in [52], shown in Fig. 16 (right). For each indexing statement, a query is sent to the solver that checks whether the indexing expression is within bounds of the array. By gradually strengthening the preconditions, the user can let the compiler guide them towards the weakest preconditions needed. For example, removing all preconditions shown, the compiler reports:

[7] Found at https://github.com/diku-dk/futhark-benchmarks.

Program	Properties & annotations	Safe	#S	#A	Check time	% of Compile time
maxMatching	Range, Equiv, Inj	✓	6	14	0.7s	29%
kmeans_ker	Range	✓	0	3	0.1s	12%
partition2	Equiv, FP	✓	1	3	0.4s	34%
partition3	Equiv, FP	✓	1	3	0.6s	44%
partition2L	Range, Equiv, FP	✓	1	3	3.6s	82%
filter	Equiv, FP	✓	1	3	0.3s	27%
filter_seg	Range, Equiv	✓	1	3	1.6s	68%

Program & Data	Base (ms)	Speedup Static	Speedup +Opt
kmeans			
movielens	280	2.2×	
nytimes	315	1.9×	
scrna	861	2.2×	
partition2			
50M	12	4.4×	4.7×
100M	38	7.0×	7.5×
200M	135	12.2×	12.8×

Fig. 17. Left: Summary of evaluated programs. IFP and FP abbreviate InvFiltPart and FiltPart, respectively. Safe reports whether all indexing and scatters are verified. #S and #A denote the number of scatters and annotations in the program. Check time is the time taken to infer index functions and prove properties (on a modern AMD CPU). Right: Performance results on an A100 GPU using Futhark's CUDA backend. Base and static denote, respectively, dynamic and static verification. +Opt removes initialisation of the scattered array.

```
kmeans.fut:7:21-34: Unsafe indexing: pointers[row] (failed to show: True => 0 ≤ row).
```

which is rectified by adding precondition `Range row (0,∞)`. Repeating this process until all errors are addressed yields the preconditions shown in the figure. We compile k-means to a CUDA program using the Futhark compiler [36]. In Fig. 17 (right), an average 2-times speed up is observed on datasets movielens [25], nytimes and scrna [39] (using parameters from [52]) by eliminating dynamic bounds checks. In Futhark, dynamic bounds checking is handled by unstructured jumps to the end of a CUDA kernel [27], which likely inhibits the `nvcc` compiler from effectively optimizing instruction-level parallelism. A dynamic approach based on slicing a safety predicate [30] may be cheaper in this case, but it will incur larger overheads to the simpler (common) cases.

Segmented operations. We conclude our case studies with `partition2L`, a batched version of `partition2` (Fig. 2), that applies to a jagged array with potentially empty rows. The function takes as input an array `shp` of type `[m]i64` whose elements denote the size of each row in the jagged array and an array `csL` of type `[n]bool` where `n = sum shp`, and outputs an array of indices, that when scattered will produce a partitioning of each row in the jagged array according to `csL`. The full program (not shown here) makes use of `sgmSum`, `mkSgmDescr`, and `mkII` from Fig. 2. The inferred index function of the indices used by the final `scatter`, denoted ys, intuitively describes the semantics of the program:

$$\bigcup_{k=0}^{m} .\text{for } i \geq \sum_{j=0}^{k-1}(shp[j]) \;.\; (cs[i] \Rightarrow \sum_{j=0}^{k-1}(shp[j]) + \sum_{j=\sum_{j'=0}^{k-1}(shp[j'])}^{i-1}(cs[j])) \bigwedge (\neg cs[i] \Rightarrow i + \sum_{j=i+1}^{\sum_{j'=0}^{k}(shp[j'])-1}(cs[j]))$$

For each segment $k = 0, \ldots, m$, if $cs[i]$ is true, then index i maps to the row offset, $\sum_{j=0}^{k-1}(shp[j])$, plus the number of trues in $cs[i]$ that come before i in that segment; otherwise, index i maps to i plus the number of trues that come after i in that segment. Verification of `partition2L` essentially comes down to verifying $InvFiltPart\ ys\ (\lambda i.\ true)\ (m, k, \sum_{j=0}^{k-1}(shp[j]), \lambda i.\ cs[i])$, which succeeds.

Additional benchmarks. We also verify and report on the following programs: `partition2` and `partition3` which partition arrays into two and three parts, respectively, and `filter`/`filter_seg` which filter a (segmented) array according to a predicate. Figure 17 summarizes the evaluation; compilation is timed using Futhark's CUDA backend. Check times increase when the index functions are complex (`part2indicesL`) or there are many annotations (`maxMatching`). Still, most programs take less than one second to check.

Fig. 17 (right) shows the impact of static *vs.* dynamic verification of scatter in `partition2` compiled to CUDA and run on arrays of random floats with 50, 100, and 200 million elements. The static version is further optimised by leaving the scattered array uninitialized—safe because we prove that all locations are overwritten (Static+Opt). Speedup factors of 4–12 are observed:

dynamic verification of scatter breaks fusion opportunities, increases the amount of irregular accesses to memory, and requires the use of reduce-by-index [29] to conform with the work asymptotic. The latter uses atomic (min) accumulations that thresh the L2 cache, thus exacerbating the overhead.

7 RELATED WORK

Liquid (Haskell). Liquid Types enhance Hindley-Milner type systems with refinement types [49, 63], generating verification conditions (VCs) in the QF-EUFLIA logic that SMT solvers check. In Liquid Haskell, refinement reflection [64] integrates source functions into refinements and automates unfolding their definitions for proofs, but still requires manual proofs (in a similar fashion to theorem provers like Rocq or Agda) for non-trivial reasoning and mandates top-level functions for each component. For example, verifying `partition2` (without the final `scatter`) demands program restructuring and manual proofs for properties like injectivity, which we still could not establish. Verification of `partition3` is harder still and `partition2L` adds another layer of complexity through the use of segmented shapes and `scatter`, which is not an operation that can be easily expressed in (Liquid) Haskell. Our system, by contrast, uses index functions to automatically infer dependencies and prove array properties without structural constraints, freeing programmers from proof-writing, aiding domain experts, and separating program and proof concerns.

F⋆ and Pulse. F⋆ [58] is a programming language and proof assistant with dependent and refinement types, supporting effects and combining SMT-based proof automation with interactive proof writing. It automates term reasoning via reductions (similar to Liquid Haskell's rewriting) and uses monadic effects to separate computation and proof. Still, it shares many of Liquid Haskell's limitations, requiring manual proofs for properties our system handles automatically, despite a more ergonomic programming discipline. Pulse [59], embedded in *F⋆*, targets concurrent programming with mutable state using Concurrent Separation Logic [11] at the statement level to reason about heaps. For instance, verifying an in-place version of `partition2` in the context of quicksort checks the two buffers against a pivot and ensures non-overlap, but this doesn't extend to a data-parallel context that fisses the computation into bulk operations and uses scatter to write all indices at once.

The reviewed approaches rely on powerful SMT solvers such as Z3 [18], which uses e-graphs [17] to generate and test all equivalent rewrites. Our rewrites of index and sum symbols (Section 5.3) are effective, but are challenging to express in Z3. Our simplification strategy uses directed rewrites, which are much cheaper computationally than e-graphs, given that (1) simplification must be performed after each symbol elimination, (2) each simplification may trigger many rewrites, and (3) the only possible objective function is whether the query succeeds.

Linear Array Logics. Dependent ML [67] and its extension ATS [68] restrict dependent values to a limited language to ensure decidability. Dependent ML is parametric in this language, enabling specialization for tasks like static array bounds checking via linear constraints [69]. ATS further allows explicit proof terms, though it requires intertwining proof and program despite their syntactic separation. Other systems, such as the quantifier-free logic by Daca et al. [15] for counting and partitioning, Bradley et al.'s logic [10] for index ranges and sortedness using Presburger arithmetic, and Qube [60] for verifying array indexing and shape matching, are restricted to linear indexing, which exclude operations like `a[b[i]]`. In comparison, we target data-parallel programs with non-linear indexing (e.g., gather/scatter/scan), which is the case of all benchmarks evaluated in Section 5.4.

Dependece-Analyses on Arrays. Our verification analysis takes inspiration from work in automatic optimization of loop-based code. Related analyses have shown that choosing a suitable representation for access patterns [35, 41, 50] is key to scaling analysis interprocedurally, either statically [24, 43, 65] or by combining static and dynamic [16, 42, 50] to cover challenging non-affine code instances. Importantly, dynamic analyses use complex transformations such as program

slicing and hoisting to extract and test at runtime sufficient conditions for statically irreducible queries. These often boil down to establishing array properties such as permutation [20, 57], injectivity [16, 50], monotonicity [42, 43]. By establishing array properties early, our approach could in principle simplify dependence analyses and eliminate expensive runtime overheads.

In hopeless cases that require some dynamic verification,[8] property specification still allows efficient code generation and placement of the inspector code (e.g., avoiding program slicing and hoisting). Finally, reminded by work on parametric polymorphism [13, 46], specification of array properties should be supported across multi-language components, which may separate the definition of an array from its use; otherwise optimization of such codes may require suboptimal speculation [45].

Scheduling Languages & Verfication of Compiler Transformations. Work on verifying the specification, code-transformations and the resulting low-level code of scheduling DSLs such as Halide [48] include improvements to its term rewriting system [38], end-to-end translation validation of *affine* specifications [14], and HaliVer [62], which aims to verify properties of (1) the specification, which utilize *linear* indexing, and (2) of the low-level generated code, which employs permission-based separation logic [8] to verify, e.g., memory safety. Finally, bounded translation validation tools such as Alive2 [33, 34] are used to verify code transformations of LLVM. These directions are complementary to our work.

8 CONCLUSIONS

We presented a framework for inferring and verifying properties of integral arrays in the context of a pure data-parallel array language. Our solution supports a small but powerful set of properties—namely range, equivalence, monotonicity, injectivity, bijectivity, and filtering/partitioning—that are easy to use and expose a rich compositional algebra to the compiler. This is used to scale (automate) the analysis (restricting user intervention to strategic places) and to optimize the program.

Our evaluation demonstrates that our framework is capable of verifying challenging code patterns from graph algorithms and from flattening irregular nested parallelism (e.g., the batch application of two-way partitioning to jagged arrays in flat form). It also shows that eliminating dynamic checks and redundant initializations results in significant GPU speedups, and that our design facilitates debugging—e.g., by user inspection of the context and index function(s).

REFERENCES

[1] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 445–447. https://doi.org/10.1145/3503221.3508422

[2] Lubin Bailly, Troels Henriksen, and Martin Elsman. 2023. Shape-Constrained Array Programming with Size-Dependent Types. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. 29–41.

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. ACM, Article 81, 14 pages. https://doi.org/10.1145/3295500.3356173

[4] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver D. Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. 2022. Productive performance engineering for weather and climate modeling with Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE Press, Article 73, 14 pages.

[5] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[8] One example is the Brownian Bridge component of the option pricing describe in [40], which uses three indirect arrays.

[6] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

[7] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. ACM, New York, NY, USA, 213–225. https://doi.org/10.1145/232627.232650

[8] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

[9] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 73–78.

[10] Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2006. What's decidable about arrays?. In *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings 7*. Springer, 427–442.

[11] Stephen Brookes and Peter W O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65.

[12] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[13] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. 2004. Parametric Polymorphism for Computer Algebra Software Components. In *Procs. of the 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '04)*. Mirton Publishing House, 119–130.

[14] Basile Clément and Albert Cohen. 2022. End-to-end translation validation for the halide language. 6, OOPSLA1, Article 84 (April 2022), 30 pages. https://doi.org/10.1145/3527328

[15] Przemysław Daca, Thomas A Henzinger, and Andrey Kupriyanov. 2016. Array folds logic. In *International Conference on Computer Aided Verification*. Springer, 230–248.

[16] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*. IEEE Computer Society, USA.

[17] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[19] Ivo Gabe de Wolff, David P. van Balen, Gabriele K. Keller, and Trevor L. McDonell. 2024. Zero-Overhead Parallel Scans for Multi-Core CPUs. In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24)*. Association for Computing Machinery, 52–61. https://doi.org/10.1145/3649169.3649248

[20] Chen Ding and Ken Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/301618.301670

[21] Joseph Fourier. 1827. Histoire de l'Académie, partie mathématique (1824). *Mémoires de l'Académie des sciences de l'Institut de France* 7 (1827).

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Int. Symposium on Code Generation and Optimization (CGO)* (Vienna, Austria) *(CGO 2018)*. ACM, 100–112. https://doi.org/10.1145/3168824

[23] Mary Hall, Cosmin E. Oancea, Anne Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. 2025. Scheduling Language Chronology: Past, Present, and Future. *ACM Trans. Archit. Code Optim.* (June 2025). https://doi.org/10.1145/3743135

[24] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 2005. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)* 27(4) (2005), 662–731.

[25] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19 pages. https://doi.org/10.1145/2827872

[26] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language*. Ph. D. Dissertation. University of Copenhagen, Universitetsparken 5, 2100 Copenhagen.

[27] Troels Henriksen. 2021. Bounds checking on GPU. *International Journal of Parallel Programming* 49, 6 (2021), 761–775.

[28] Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, 1–14. https://doi.org/10.1145/3460944.3464310

[29] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 97, 14 pages.

[30] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. Association for Computing Machinery, New York, NY, USA, 88–94. https://doi.org/10.1145/2627373.2627388

[31] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571.

[32] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[33] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[34] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2018. Practical verification of peephole optimizations with Alive. *Commun. ACM* 61, 2 (Jan. 2018), 84–91. https://doi.org/10.1145/3166064

[35] Sungdo Moon and Mary W. Hall. 1999. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, Georgia, USA) *(PPoPP '99)*. Association for Computing Machinery, New York, NY, USA, 84–95. https://doi.org/10.1145/301104.301112

[36] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. 2021. Dataset Sensitive Autotuning of Multi-versioned Code Based on Monotonic Properties. In *Trends in Functional Programming*, Viktória Zsók and John Hughes (Eds.). Springer International Publishing, Cham, 3–23.

[37] Philip Munksgaard, Troels Henriksen, Ponnuswamy Sadayappan, and Cosmin Oancea. 2022. Memory Optimizations in an Array Language . In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–15. https://doi.org/10.1109/SC41404.2022.00036

[38] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and improving Halide's term rewriting system with program synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428234

[39] Corey J Nolet, Divye Gala, Edward Raff, Joe Eaton, Brad Rees, John Zedlewski, and Tim Oates. 2022. GPU semiring primitives for sparse neighborhood methods. *Proceedings of Machine Learning and Systems* 4 (2022), 95–109.

[40] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial software on GPUs: between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC '12)*. Association for Computing Machinery, 61–72. https://doi.org/10.1145/2364474.2364484

[41] Cosmin E. Oancea and Alan Mycroft. 2008. *Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS)*. Springer-Verlag, Berlin, Heidelberg, 156–171. https://doi.org/10.1007/978-3-540-89740-8_11

[42] Cosmin E. Oancea and Lawrence Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Languages and Compilers for Parallel Computing*, Sanjay Rajopadhye and Michelle Mills Strout (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75.

[43] Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable Conditional Induction Variables (CIV) Analysis. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, Washington, DC, USA, 213–224. http://dl.acm.org/citation.cfm?id=2738600.2738627

[44] Cosmin Eugen Oancea, Ties Robroek, and Fabian Gieseke. 2020. Approximate Nearest-Neighbour Fields via Massively-Parallel Propagation-Assisted K-D Trees. In *2020 IEEE International Conference on Big Data (Big Data)*. 5172–5181. https://doi.org/10.1109/BigData50022.2020.9378426

[45] Cosmin E. Oancea, Jason W. A. Selby, Mark Giesbrecht, and Stephen M. Watt. 2005. Distributed Models of Thread-Level Speculation. In *Procs. of the Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '05)*. 920–927.

[46] Cosmin E. Oancea and Stephen M. Watt. 2005. Parametric polymorphism for software component architectures. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, 147–166. https://doi.org/10.1145/1094811.1094823

[47] Yunheung Paek, Jay Hoeflinger, and David Padua. 2002. Efficient and Precise Array Access Analysis. *Trans. on Prog. Lang. and Sys. (TOPLAS)* 24(1) (2002), 65–109.

[48] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[49] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. [n. d.]. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008-06-07) *(PLDI '08)*. Association for Computing Machinery, 159–169. https://doi.org/10.1145/1375581.1375602

[50] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2002. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. Association for Computing Machinery, 274–284. https://doi.org/10.1145/514191.514229

[51] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298.

[52] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. 2022. AD for an array language with nested parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 58, 15 pages. https://doi.org/10.1109/SC41404.2022.00063

[53] Dmitry Serykh, Stefan Oehmcke, Cosmin Oancea, Dainius Masiliūnas, Jan Verbesselt, Yan Cheng, Stéphanie Horion, Fabian Gieseke, and Nikolaj Hinnerskov. 2023. Seasonal-Trend Time Series Decomposition on Graphics Processing Units. In *IEEE International Conference on Big Data (BigData)*. 5914–5923. https://doi.org/10.1109/BigData59044.2023.10386208

[54] Wilfried Sieg and Barbara Kauffmann. 1993. *Unification for quantified formulae.* Carnegie Mellon [Department of Philosophy].

[55] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 205–217. https://doi.org/10.1145/2784731.2784754

[56] M. Steuwer, T. Koehler, B. Köpcke, and F. Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. arXiv:2201.03611 [cs.PL]

[57] Michelle Mills Strout and Paul D. Hovland. 2004. Metrics and models for reordering transformations. In *Proceedings of the 2004 Workshop on Memory System Performance* (Washington, D.C.) *(MSP '04)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/1065895.1065899

[58] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 256–270.

[59] Nikhil Swamy, Guido Martínez, and Aseem Rastogi. 2023. Proof-Oriented Programming in F.

[60] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643–664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[61] Lars B. van den Haak, Trevor L. McDonell, Gabriele K. Keller, and Ivo Gabe de Wolff. 2020. Accelerating Nested Data Parallelism: Preserving Regularity. In *Euro-Par 2020: Parallel Processing*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer International Publishing, Cham, 426–442.

[62] Lars B. van den Haak, Anton Wijs, Marieke Huisman, and Mark van den Brand. 2024. HaliVer: Deductive Verification and Scheduling Languages Join Forces. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 71–89.

[63] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[64] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. [n. d.]. Refinement reflection: complete verification with SMT. 2 ([n. d.]), 1–31. Issue POPL. https://doi.org/10.1145/3158141

[65] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[66] H Paul Williams. 1986. Fourier's Method of Linear Programming and its Dual. *The American Mathematical Monthly* 93, 9 (1986), 681–695. https://doi.org/10.1080/00029890.1986.11971923 arXiv:https://doi.org/10.1080/00029890.1986.11971923

[67] Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *Journal of Functional Programming* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216

[68] Hongwei Xi. 2017. Applied type system: An approach to practical programming with theorem-proving. *arXiv preprint arXiv:1703.08683* (2017).

[69] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732

[70] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Association for Computing Machinery, Article 1, 13 pages. https://doi.org/10.1145/3295500.3357156