Verifying Array Properties in Pure Data-Parallel Programs

NIKOLAJ HEY HINNERSKOV, University of Copenhagen, Denmark
ROBERT SCHENCK, Northeastern University, USA
COSMIN OANCEA, University of Copenhagen, Denmark

In functional data-parallel programs, index array computations are separated into sequences of bulk-parallel operators—map, prefix sum, scatter—and used to gather or scatter data array elements, thus determining data array properties. This programming style is problematic for general-purpose verification frameworks (e.g., Dafny, F*, Liquid Haskell), which are flexible and powerful, but require verbose annotations and non-trivial user proofs, making them inaccessible to non-experts. We present a *compiler approach* to verifying array properties with high automation, aimed at making verification of data-parallel programs more accessible to users without verification expertise. We support a small but powerful predefined set of properties—equivalence, range, injectivity, bijectivity, monotonicity, filtering, partitioning—that enable the compiler to (automatically) reason at a higher level of abstraction. We evaluate our approach on challenging applications with non-linear indexing, including graph algorithms, Cooley-Tukey FFT, filtering, multi-way partitioning, and flattened irregular nested parallel programs that are difficult to verify, such as batch operations on arrays of different sizes.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; **Automated static analysis**; **Software verification**; **Compilers**; **Functional languages**.

Additional Key Words and Phrases: Array programming, data parallelism, irregular nested parallelism

ACM Reference Format:
Nikolaj Hey Hinnerskov, Robert Schenck, and Cosmin Oancea. 2026. Verifying Array Properties in Pure Data-Parallel Programs. *Proc. ACM Program. Lang.* 10, PLDI, Article 226 (June 2026), 27 pages. https://doi.org/10.1145/3808304

1 Introduction

High-performance array languages (e.g., Futhark [33], Accelerate [69], Lift [26], DaCe [5], JAX [10]) and machine learning frameworks (e.g., TensorFlow [1], MLX [29], PyTorch [51]) express parallel algorithms by composing bulk-parallel operators such as map, scan (prefix sum), scatter (irregular write), and gather (irregular read). Unlike loops in imperative programming or folds in functional programming, where computation is typically manually fused, these operators stay separate.

General-purpose verification frameworks like Dafny [36], F* [65], and Liquid Haskell [54, 72] can encode data-parallel programs but lack native support and specialization for bulk-parallel operators. Proving even simple array properties often requires verbose annotations and manual inductive proofs, if the proof is possible at all. Scatter is the most challenging construct. For example, partitioning arrays is implemented by scattering array elements to computed target positions, or by using scatter to compute reordering indices then gathering elements using those indices. Proving this produces a valid partition requires recognizing that the scatter indices—which map each index i to its target position—form a permutation (i.e., the inverse of the final arrangement), and in the gather case, that gather inverts this index mapping. Users of general-purpose verifiers must encode

Authors' Contact Information: Nikolaj Hey Hinnerskov, University of Copenhagen, Copenhagen, Denmark, nhey@900901.xyz; Robert Schenck, Northeastern University, Boston, USA, r@bert.lol; Cosmin Oancea, University of Copenhagen, Copenhagen, Denmark, cosmin.oancea@di.ku.dk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART226
https://doi.org/10.1145/3808304

this relationship manually, breaking automation—and even having done so, frameworks like Dafny may still fail to verify it (see Section 2.1.3). Likewise, verifying a two-way partition requires auxiliary lemmas and proof hints specific to the code (Section 2.1.1). And small implementation changes may require fundamentally different proof strategies, further undermining automation (Section 2.1.2).

To address these shortcomings, this paper presents PropProp—a compiler-based system that automatically verifies data-parallel programs written as purely functional array computations over bulk-parallel operators. PropProp (P^2 for short), implemented in the Futhark compiler, allows users to annotate functions with pre- and postconditions using a small but powerful set of predefined properties: equivalence, range, injectivity, bijectivity, monotonicity, and filtering/partitioning. P^2 is not a general-purpose theorem prover; it is a deterministic static-analysis algorithm designed for the common case in data-parallel programming at reasonable compilation time, by leveraging the semantics of bulk-parallel operators to enable highly automated verification for the target array properties. It is not intended to be complete nor to support arbitrary properties and user proofs.

The key idea is to infer *index functions* from integer arrays in the source program. Index functions are functions from indices to the elements at those indices—defined piecewise by guarded expressions built from a carefully chosen algebra of primitives, including sums (of array slices) and inverses of bijective index functions (which enable reasoning about permutations). Translating bulk-parallel computations into index functions reduces verification to reasoning about inequalities using a set of high-level rewrite rules, which is easier to automate than an inductive approach.

This approach scales to challenging scenarios without extra proof machinery. For example, P^2 can reason about jagged arrays (arrays of variable-length rows)—represented as flat arrays for data along with independent auxiliary arrays that encode the irregular shape, which are computed as part of the program (Section 3.4). P^2 also reasons about nested parallel operations over irregular rows that have been manually flattened into regular bulk-parallel operations (e.g., segmented scan [6]).

P^2 works by translating source functions into index functions and then verifies and infers properties about them. It consists of three architectural components: (1) The *index function layer* infers the values and structure of arrays as index functions (Sections 3.1, 3.4 and 4.3). (2) The *property layer* tracks and proves properties over the index functions (Sections 3.2, 4.1 and 4.4). (3) The *algebra layer* (Sections 3.3 and 4.5) reasons about and dispatches algebraic queries (generated by the property layer) using a Fourier-Motzkin elimination-based [23] solver that supports sums of array slices, array indexing and mutually exclusive guards via rewrite rules applied at each elimination step.

The components are deeply interconnected (Section 4.2) and arguably support the minimal property set required in a data-parallel setting—e.g., the index function layer exploits injectivity, bijectivity and monotonicity properties to produce meaningful index functions for scatter that enable the derivation of filter/partition properties and flat expression of jagged arrays. The Sparse Polyhedral Framework (SPF) [62, 64] crucially depends on almost the same set of properties [41] (which are manually annotated and expensive to verify dynamically) to extend dependence analysis to challenging non-affine cases, in a principled and sound way that minimizes runtime overheads [42].

Our approach is complementary to other work that studies properties that we do not support, such as sortedness [19, 21, 61] or verification of sequentially-constructed arrays [8, 66, 70], e.g., by means of concurrent separation logic [12]. We make the following contributions:

(1) Static verification of a key property set on irregular index arrays in data-parallel programs.
(2) An end-to-end system that infers arrays' content as index functions and supports cheap derivation of properties at a high level and inequality solving using sums of array slices.
(3) An evaluation of challenging benchmarks, including graph algorithms and implementations that flatten nested parallelism, that verifies all indexing, scatters, and annotated properties.

$$\text{Variables}\quad x, y, z \qquad \text{Function variables}\quad F \qquad \text{Constants}\quad n \in \mathbb{Z}$$

$$\begin{array}{lll}
\tau & ::= [\,]\tau \mid \text{i64} \mid \text{f64} \mid \text{bool} \mid \ldots & \text{Types} \\
\prec & ::= \, < \mid \leq \mid > \mid \geq & \text{Linear orders} \\
Op & ::= \, \prec \mid + \mid - \mid * \mid = \mid \neq \mid \wedge \mid \vee & \text{Binary operators} \\
B & ::= n \mid x \mid |x| \mid 2^B \mid B \;\; Op \;\; B \mid x_{array}[\overline{B_{index}}] & \text{Base expressions} \\
\hline
E^\pi & ::= B \mid \text{Sum } x[E^\pi : E^\pi] & \text{Property expressions} \\
\pi & ::= \text{Range } x \; E^\pi..E^\pi \mid \text{Mono } x \prec \mid \text{Equiv } x \; E^\pi \mid \text{Inj } x \; E^\pi..E^\pi \mid \text{Bij } x \; E^\pi..E^\pi \; E^\pi..E^\pi \mid \text{Filt } x \; x \; (\lambda x.\, E^\pi) & \text{Properties} \\
 & \mid \text{Part } x \; x \; (\lambda x.\, E^\pi) \mid \text{InvFiltPart } x \; E^\pi..E^\pi \; (\lambda x.\, E^\pi) \; (\overline{\lambda x.\, E^\pi}) \mid \text{FiltPart } x \; x \; (\lambda x.\, E^\pi) \; (\overline{\lambda x.\, E^\pi}) \mid \text{For } (i : E^\pi..E^\pi) \; \pi & \\
\hline
E^\oslash & ::= B \mid B..B \mid F \; \overline{x} \mid \text{if } B \text{ then } E \text{ else } E & \text{Base expression, Sequence, Function application, Conditional} \\
 & \mid \text{map } (\lambda \overline{x}^{(n)}.\, E) \; \overline{x_{array}}^{(n)} \mid \text{scan } (\lambda \overline{x_1}^{(n)} \; \overline{x_2}^{(n)}.\, E) \; \overline{B}^{(n)} \; \overline{x_{array}}^{(n)} \mid \text{scatter } x_{dst} \; x_{idx} \; x_{val} & \text{SOACs} \\
 & \mid \textbf{loop } \overline{x}^{(n)} = \overline{x_{init}}^{(n)} \textbf{ while } x_n \textbf{ do } F \; \overline{x}^{(n)} \mid \textbf{loop } \overline{x}^{(n)} = \overline{x_{init}}^{(n)} \textbf{ for } x_{n+1} < B \textbf{ do } F \; \overline{x}^{(n+1)} & \text{Loops} \\
E & ::= \overline{x} \mid \overline{\textbf{let } \overline{x} = E^\oslash \textbf{ in }} \; E & \text{Expressions} \\
Prog & ::= \epsilon \mid \textbf{def } F \; \overline{(x : \tau \mid \overline{\pi})} : \; (\tau \;\mid\; \lambda \overline{x}.\, \overline{\pi}) = E \;\; Prog & \text{Programs/Function definitions}
\end{array}$$

Fig. 1. Source language syntax. $\overline{X}^{(n)}$ denotes a sequence $X_1, \ldots, X_n$; we write $\overline{X}$ if the length is not needed.

2 Source Language and Motivation

P² analyzes array programs expressed in a purely functional language using second-order array combinators (SOACs) [33] map, scan, and scatter. SOACs are bulk-parallel array operations parameterized by user-defined functions: they express the first-order primitives found in most array languages such as vectorized operations (using map), reductions and cumulative sums (using scan), scatters, and gathers (map $(\lambda i.\, xs[i])$ idx). The types and semantics of map and scan are:

$$\begin{array}{ll}
\text{map} \;:\; (\alpha \to \beta) \to [\,]\alpha \to [\,]\beta & \text{scan} \;:\; (\alpha \to \alpha \to \alpha) \to \alpha \to [\,]\alpha \to [\,]\alpha \\
\text{map } f \; [x_1, \ldots, x_n] = [f \; x_1, \ldots, f \; x_n] & \text{scan } \odot \; e_\odot \; [x_1, \ldots, x_n] = [x_1, x_1 \odot x_2, \ldots, x_1 \odot \ldots \odot x_n]
\end{array}$$

where $[\,]\alpha$ is an array of elements of type α and $\odot$ is an associative binary operator (function) with neutral element $e_\odot$ (e.g., 0 is the neutral element for +).[1] For convenience, we overload map to be variadic. For example, map $(\lambda x \; y.\, x + y)$ $[x_1, \ldots, x_n]$ $[y_1, \ldots, y_n] = [x_1 + y_1, \ldots, x_n + y_n]$.

The most complex array operator is scatter, which has the type and semantics:

$$\begin{array}{l}
\text{scatter} \;:\; [\,]\alpha \to [\,]\text{i64} \to [\,]\alpha \to [\,]\alpha \\
z = \text{scatter } y \; idx \; x
\end{array}
\quad \equiv \quad
z[i] = \begin{cases} x[j] & \text{if } \; \exists j \in [0, |x|) \;.\; idx[j] = i \\ y[i] & \text{otherwise} \end{cases} \tag{1}$$

The result z of scatter is a copy of y updated in place at indices idx with the corresponding values from x, but ignoring updates to indices that are out of bounds in z.[2] Scatter requires index and value arrays of equal lengths $|idx| = |x|$. Since scatter performs all the writes at once, its semantics requires that *duplicate in-bounds indices must correspond to idempotent (equal) values*,[3] i.e., $\forall j, k \in [0, |x|) \;.\; 0 \leq idx[j] = idx[k] < |y| \Rightarrow x[j] = x[k]$. This not only enables deterministic execution, but also eliminates the need for locking when the element type is a tuple and it enables an array-of-structures to structure-of-arrays transformation. Futhark adheres to this semantics and Pencil [4] similarly relaxes dependence analysis to permit idempotent updates in parallel loops.

Source Language. The source language syntax is shown in Fig. 1. Functions are in A-normal [56] structure-of-arrays form, where bodies consist of a list of non-nested let-expressions followed by one or more result variables. We also require that all variable names are unique. The function parameter $(x : \tau \mid \pi_{\text{pre}})$ says that x has type τ and is subject to precondition π_{pre}, which is assumed

[1] This is the standard type and semantics of inclusive scan. The neutral element $e_\odot$ is used only by the compiler for padding.
[2] Disallowing out-of-bounds indices would require explicit filtering of the index-value pairs, which requires additional memory accesses and is expensive in practice. The implementation of filter directly exploits this feature to discard values.
[3] In practice, common cases are either that the in-bounds indices of idx are unique or that all values of x are equal.

```
1  def partition (p : f64 → bool) (xs : []f64)
2                  : []f64 | λys. Part ys xs (λi. p xs[i]) =
3    let mask = map (λx. p x) xs
4    let left = map (λc. if c then 1 else 0) mask
5    let right = map (λx. 1 − x) left
6    let n_left = scan (λx y. x + y) 0 left
7    let n_right = scan (λx y. x + y) 0 right
8    let split = if |xs| > 0 then n_left[|xs| − 1] else 0
9    let idx = map (λc l r. if c then l − 1 else split + r − 1)
10                 mask n_left n_right
11   let zeros = map (λx. 0) xs
12   let ys = scatter zeros idx xs in ys
```

Example program state

```
>>> let xs = [1,4,2,4,3]
>>> partition (λx. x == 4) xs
mask = [false,true,false,true,false]
left = [0, 1, 0, 1, 0]
right = [1, 0, 1, 0, 1]
n_left = [0, 1, 1, 2, 2]
n_right = [1, 1, 2, 2, 3]
split = 2
idx = [2, 0, 3, 1, 4]
zeros = [0, 0, 0, 0, 0]
ys = [4, 4, 1, 2, 3]
```

Fig. 2. A source program implementing *partition* using SOACs.

when analyzing the function's body and is checked at call sites. The return type $\tau \mid \lambda y.\, \pi_{\text{post}}$ says that the function return has type τ and satisfies the postcondition π_{post}, which must be proved by P^2. The pre- and postconditions specify (conjunctions of) the supported set of properties (π). We write $|x|$ to denote the size of x's first dimension and we treat scalars as unit-length arrays. The source language does not permit irregular (jagged) arrays or anonymous functions (except in SOACs). We further disallow: properties on arguments of function type, passing functions with pre- and postconditions as arguments, and partial application.

Motivating Example. Most data-parallel array programs chain array operations over the inputs and intermediate variables, with gathers and scatters introducing indirect indexing. The program in Fig. 2 partitions an array xs according to a predicate p, placing elements that satisfy p before those that do not, while preserving the original element order within each group. Each computational step is a separate array operation: the target indices for elements satisfying the predicate are computed by mapping p over xs (line 3), converting booleans to integers (line 4), computing prefix sums via scan (line 6), and subtracting one (line 9). Failing elements are handled similarly using the negated predicate (line 5), with indices offset by *split*—the count of successful elements (line 8). Finally, scatter reorders the data array all at once. Fig. 2 (right) shows an example program state.

In data-parallel programs, scatters and gathers may propagate properties on index arrays to arrays of any type (e.g., data arrays). For example, *idx* is a permutation of xs' indices $(0, \ldots, |xs|-1)$. By proving and propagating this information, a compiler can reason that the output array ys is a permutation of xs and use this for verification and optimization (e.g., *zeros* does not need to be initialized since all of its elements are overwritten). In this case, the postcondition $\lambda ys.\, \text{Part}\ ys\ xs\ (\lambda i.\, p\ xs[i])$ further requires that the permutation must form a partition (Part) of xs according to p.

2.1 Challenges of Verifying Partition in Dafny

This section is a case study illustrating the challenges of verifying *partition* in Dafny [36]—an industrial-strength verification framework (e.g., used at AWS [16]) with good support for reasoning about data-dependent array accesses. We built a library of SOACs in Dafny, e.g., map is defined as:

```
function map<T1,T2>(f: T1 -> T2, xs: seq<T1>) : (ys: seq<T2>)
  ensures (|xs| == |ys|) && (forall i :: 0 <= i < |xs| ==> ys[i] == f(xs[i]))
  { seq(|xs|, i requires 0 <= i < |xs| => f(xs[i])) }
```

2.1.1 Verifying Properties of Scatter Indices. Figure 3a shows Dafny code for *partition_inds*, which constitutes the primary verification burden for *partition*. Dafny is able to verify the postconditions on `idx` (lines 3–8), but only with two manual user-defined lemmas (lines 19–21). Figure 3b shows an

```
1  method partition_inds(p: int -> bool, xs: seq<int>)
2                      returns (split: int, idx: seq<int>)
3    ✓ ensures |xs| == |idx| && (...)
4    ✓ ensures forall i, j :: 0 <= i < j < |xs| ==>
5              (p(xs[i]) && p(xs[j]) ==> idx[i] < idx[j])
6          && (!p(xs[i]) && !p(xs[j]) ==> idx[i] < idx[j])
7          && (p(xs[i]) && !p(xs[j]) ==> idx[i] < idx[j])
8          && (!p(xs[i]) && p(xs[j]) ==> idx[i] > idx[j])
9  { var mask := map(x => p(x), xs);
10   var left := map(c => if c then 1 else 0, mask);
11   var right := map(b => 1 - b, left);
12   var n_left := scan((x,y) => x + y, 0, left);
13   var n_right := scan((x,y) => x + y, 0, right);
14   split := if |xs| > 0 then n_left[|xs|-1] else 0;
15   var indsF := map(t => t + split, n_right);
16   idx := map3((c,l,r) => if c then l-1 else r-1,
17               mask, n_left, indsF);
18   // Lemmas needed to prove postconditions.
19   SumOverPositiveIsMonotonic(left, n_left);
20   SumOverPositiveIsMonotonic(right, n_right);
21   ComplementarySums(left,n_left,right,n_right); }
```

(a) Dafny source code for *partition_inds*.

```
lemma ComplementarySums( xs:seq<int>,
      sum_xs: seq<int>, ys: seq<int>, sum_ys: seq<int> )
  // sum_xs is a sum over xs.
  requires (|xs|==|sum_xs|) && (0 < |xs| ==> sum_xs[0] == xs[0])
  requires forall i:: 1<=i<|xs| ==> sum_xs[i]==xs[i]+sum_xs[i-1]
  // sum_ys is a sum over ys.
  requires (|ys|==|sum_ys|) && (0 < |ys| ==> sum_ys[0] == ys[0])
  requires forall i:: 1<=i<|ys|==> sum_ys[i]==ys[i]+sum_ys[i-1]
  // xs and ys are complementary booleans.
  requires |xs| == |ys|
  requires forall i :: 0 <= i < |xs| ==> 0 <= xs[i] <= 1
  requires forall i :: 0 <= i < |ys| ==> ys[i] == 1 - xs[i]
  ✓ ensures forall i :: 0<=i<|xs| ==> sum_xs[i]+sum_ys[i] == i+1
{ if xs == [] { assert ys == []; }
  else if |xs| == 1 { assert sum_xs[0] + sum_ys[0] == 1; }
  else {
    ComplementarySums(xs[..|xs|-1], sum_xs[..|xs|-1],
         ys[..|xs|-1], sum_ys[..|xs|-1]);
   assert (sum_xs[|xs|-1] + sum_ys[|xs|-1]
            == 1 + sum_xs[|xs|-2] + sum_ys[|xs|-2]);
 }}
```

(b) Manual lemma needed in *partition_inds*.

```
1  method scatter<T>(y: seq<T>, idx: seq<int>, x: seq<T>)
2                       returns (z: seq<T>)
3   requires (|idx| == |x|) && (injective(idx) || replicated(x))
4  ✓ ensures |y| == |z|
5  ✓ ensures forall k :: 0≤k<|idx| && 0 ≤ idx[k] < |z| ==> z[idx[k]] == x[k]
6  ✓ ensures forall i :: 0 ≤ i < |z| ==>
7   ((z[i] == y[i]) || (exists k :: 0 ≤ k < |idx| && idx[k] == i && z[i] == x[k]))
8  { ... }
9
10 method partition(p: int->bool, xs: seq<int>) returns (ys: seq<int>) {
11   var split, idx := partition_inds(p, xs);
12   var dest := seq(|xs|, i requires 0 <= i < |xs| => 0);
13   ys := scatter(dest, idx, xs);
14   × assert (forall i :: 0 ≤ i < split ==> p(ys[i]));
15   × assert (forall i :: split ≤ i < |xs| ==> !p(ys[i]));
16 }
```

(c) Dafny source code for scatter and *partition*.

```
method partition(p: int -> bool, xs: seq<int>)                      1
                    returns (ys: seq<int>) {                         2
  var split, idx := partition_inds(p, xs);                           3
  var dest := seq(|xs|, i requires 0≤i<|xs| => 0);                   4
  var iota := seq(|xs|, i requires 0≤i<|xs| => i);                   5
  var σ := scatter (dest, idx, iota);                                6
  ys := seq(|xs|, i requires 0≤i<|xs| => xs[σ[i]]);                  7
                                                                     8
  × assert (forall i :: 0 ≤ i < |xs| ==> idx[σ[i]] == i);            9
  assume (forall i :: 0 ≤ i < |xs| ==> idx[σ[i]] == i);             10
  // These fail without the assumption above.                       11
  ✓ assert (forall i :: 0 ≤ i < split ==> p(ys[i]));                12
  ✓ assert (forall i :: split ≤ i < |xs| ==> !p(ys[i]));            13
}                                                                   14
```

(d) Zooming in to identify the core verification challenge: the *inverse* property.

Fig. 3. Dafny source code demonstrating challenges of verifying *partition*. Lines marked with × and ✓ fail and succeed, respectively. Succeeding lines that come after **assume** would fail without that assumption.

inductive proof for one lemma. Although Dafny verifies the postcondition, this experiment demonstrates that it requires non-trivial manual effort—in particular, coming up with the needed lemmas.

2.1.2 Small Alterations Require New Proof Strategies. Dafny's verification process is brittle on SOAC-based programs—small changes to the implementation may require an entirely new proof strategy. We illustrate this with two simple examples.

First, if `partition_inds` is rewritten to compute the scattered indices of failing elements using a reverse prefix sum, verifying the index properties requires proving a query of the form:

$$0 \leq j < i < |xs| \ \wedge \ mask[j] = 0 \ \wedge \ mask[i] = 1 \Rightarrow \ j + \sum_{k=j+1}^{|xs|-1} mask[k] > \sum_{k=0}^{i-1} mask[k] \quad (2)$$

This is challenging to solve because the quantified variables j and i define the lower and upper bounds of their respective sums. In fact—even when using a lemma over a recursive definition of sum in which the bounds are passed as arguments—we were unable to prove this query in Dafny.

A second example that implements an exclusive prefix sum: $\text{sum}^{exc}\ [a_1, \ldots, a_n] = [0,\ a_1,\ a_1 + a_2,\ \ldots,\ a_1 + \ldots + a_{n-1}]$ by composing a map (to shift elements) and an inclusive scan reveals a related limitation. Dafny can prove that the output is a prefix sum over the shifted array, but we were unable to prove that it is a prefix sum over the original array. (More details are in Appendix C.)

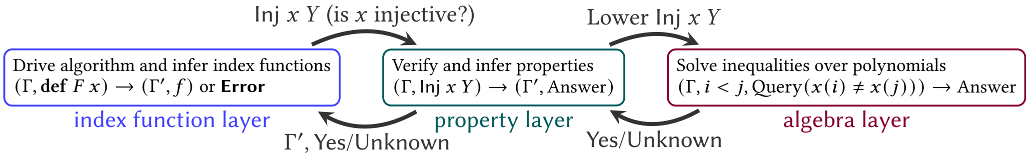
2.1.3 Scatter is Challenging in Dafny. Figure 3c shows the pre- and postconditions of our scatter implementation in Dafny, ensuring the semantics given in Eq. (1). Figure 3c also shows *partition* as presented in Fig. 2: scatter is applied to the indices `idx` resulting from the successfully verified call to *partition_inds* (line 11). However, the verified properties of scatter and `idx` are not transferable to the result array `ys`; e.g., Dafny fails to prove that the indices smaller than the split point correspond to elements that succeed under the predicate (line 14) and the others to the ones that fail (line 15).

To assist Dafny's reasoning, we can make the permutation explicit by altering the implementation of *partition* (Fig. 3d) to scatter $[0, \dots, |xs| - 1]$ at positions idx to produce $\sigma = idx^{-1}$ and then gather xs using σ. Dafny verifies the postconditions (lines 12, 13) given a manual assumption that σ is idx's inverse (line 10), but fails to verify this assumption. Even when guided by assertions establishing that $0 \leq i < |xs| \Rightarrow \sigma[idx[i]] == i$, which should allow Dafny to infer that σ's elements are unique, it can't verify this assumption. (Dafny is able to show that $0 \leq i < j < |xs| \Rightarrow \sigma[idx[i]] \neq \sigma[idx[j]]$, but can't show $0 \leq i < j < |xs| \Rightarrow \sigma[i] \neq \sigma[j]$.) Not all scatters can be rewritten like this: out-of-bounds indices are ignored, so the indices do not necessarily form a permutation.

3 Overview

The key idea behind P^2 is to transform array programs into a representation where properties become algebraic (in)equalities over *index functions*—functions that map the indices of an array to its values. As we'll see in this section, this index-centric transformation enables automatic reasoning about scatter-gather patterns that require manual proof writing in general-purpose tools.

We present P^2 as a transformation $\mathcal{P}$ from type-checked source programs to index functions and properties over these index functions: $\mathcal{P}$: source program $\rightarrow$ index functions $\times$ properties. P^2 translates each function definition (**def**) to an index function to verify and infer properties over it. At call sites, P^2 reuses inferred index functions, checks preconditions, and propagates postconditions into the caller's context. If a specified property fails to verify, P^2 terminates in error.



The above figure shows P^2's three constituent components: the *index function layer*, the *property layer*, and the *algebra layer*, and how these interact to verify the *partition* program. The index function layer translates programs to index functions via inference rules, applying them by querying the property layer to verify rule premises. The property layer records and proves properties using a two-level strategy: a higher level infers new properties from proven ones; a lower level proves new properties by reducing them to equivalent algebraic (in)equalities, which are dispatched to the algebra layer. The algebra layer normalizes algebraic expressions and solves (in)equalities using a Fourier-Motzkin elimination-based solver. In the following sections, we'll walk through how each of P^2's components works and interacts to verify the *partition* program (see Figs. 2 and 4).

3.1 Index Function Layer

The index function layer translates each function definition (**def**) by first populating P^2's environment with argument information. Preconditions on formal arguments are assumed and entered into the property environment (*partition* has no preconditions). The boolean values false and true are syntactic sugar for 0 and 1, respectively. Scalars are treated as unit-length arrays. The postcondition, $\lambda ys.\, \mathsf{Part}\ ys\ xs\ (\lambda i.\, p\ xs[i])$, is treated after the body has been analyzed. Function bodies are translated one let-binding at a time into index functions. Since bodies are A-normal,

$\mathcal{P}(\textbf{def}\ \mathit{partition}\ (p : \text{f64} \rightarrow \text{bool})\ (xs : [\,]\text{f64}) = \ldots)$

$$\begin{aligned}
&\mathit{mask} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * p(xs(i)) \\
&\mathit{left} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * p(xs(i)) \\
&\mathit{right} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * (1 - p(xs(i))) \\
&\ldots \\
&\boxed{\mathit{n_right} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * (i + 1 - \textstyle\sum_{j=0}^{i}(p(xs(i))))} \\
&\ldots \\
&\mathit{idx} = \lambda\,(i : 0..n)\ .\ \begin{cases} [\,p(xs(i))\,] * \sum_{j=0}^{i-1}(p(xs(j))) \\ [\,\neg p(xs(i))\,] * (i + \sum_{j=i+1}^{n-1}(p(xs(j)))) \end{cases} \\
&\mathit{zeros} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * 0 \\
&\mathit{ys} = \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * xs(\mathit{idx}^{-1}(i))
\end{aligned}$$

Notation: $n = |xs|$.

$$\left\{\begin{aligned}
&\mathcal{P}(\textbf{let}\ \mathit{n_right} = \text{scan}\ (\lambda x\ y.\, x + y)\ 0\ \mathit{right}) \\
&1.\ \lambda\,(i : 0..n)\ .\ \begin{cases} [\,i = 0\,] * \mathit{right}(i) \\ [\,i \neq 0\,] * (\circlearrowleft + \mathit{right}(i)) \end{cases} \\
&2.\ \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * (\mathit{right}(0) + \textstyle\sum_{j=1}^{i}(\mathit{right}(i))) \\
&3.\ \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * \textstyle\sum_{j=0}^{i}(\mathit{right}(i)) \\
&4.\ \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * \textstyle\sum_{j=0}^{i}([\,\text{true}\,] * (1 - p(xs(i)))) \\
&5.\ \lambda\,(i : 0..n)\ .\ [\,\text{true} \wedge \text{true}\,] * \textstyle\sum_{j=0}^{i}(1 - p(xs(i))) \\
&6.\ \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * (\textstyle\sum_{j=0}^{i}(1) - \sum_{j=0}^{i}(p(xs(i)))) \\
&7.\ \lambda\,(i : 0..n)\ .\ [\,\text{true}\,] * (i + 1 - \textstyle\sum_{j=0}^{i}(p(xs(i))))
\end{aligned}\right.$$

Fig. 4. P² transforms programs to index functions, enabling algebraic reasoning about properties.

each let-binding applies a SOAC or other source construct to earlier bindings. For example, P² first translates **let** $\mathit{n_right}$ = scan $(\lambda x\ y.\, x + y)$ 0 right into an index function (right of Fig. 4):

$$\mathit{n_right} = \lambda\,(i : 0..|xs|)\ .\ [\,i = 0\,] * \mathit{right}(i) + [\,i \neq 0\,] * (\circlearrowleft + \mathit{right}(i))$$

The index function, denoted by a lambda abstraction, consists of a *domain* $(i : 0..|xs|)$, which says that indices i range from 0 to $|xs| - 1$ and a *guarded expression* that defines the value at each index. The guards must partition the domain; for any index in the domain, exactly one guard is true. Here, the guarded expression has two guards $[\,i = 0\,]$ and $[\,i \neq 0\,]$ with corresponding expressions $\mathit{right}(i)$ and $\circlearrowleft + \mathit{right}(i)$ that define the value at index i when their guard holds. (The notation is inspired by [37].) The $\circlearrowleft$ symbol represents the result of scan at the previous index; term rewriting eliminates $\circlearrowleft$ in favor of a sum-based formula. In general, conditionals are lifted into guards—producing mutually exclusive and collectively exhaustive predicates over the index function domain.

Formal arguments like xs and p are treated as uninterpreted functions, which exhibit function congruence and may be the subject of properties in the environment. Variables are overloaded: xs is used to refer both to the source-level array as well as the corresponding index function that P² reasons with. For example, $xs(i)$ is an application of the function xs to the index i, while $xs[i]$ is source-level expression that indexes into the array xs.

Normalization. Expressions are normalized via rewrite rules. For example, the rewrites for $\mathit{n_right}$ in Fig. 4 are: (1) introduces a recurrence $\circlearrowleft$ based on scan's semantics (Scan); (2) converts the recurrence to a closed-form sum (RecSum); (3) absorbs the base term into the sum; (4) substitutes $\mathit{right}(i)$ with its guarded expression (Sub); (5) hoists the nested guard $[\,\text{true}\,]$ to the outer expression (Hoist); (6) splits the resulting sum; and (7) eliminates any constant sums. Substituting and hoisting guarded expressions enables P² to automatically track positional dependencies backward to function arguments. For example, idx's index function (Fig. 4) says if $p(xs(i))$ is true, index i maps to the count of preceding true elements; if false, it maps to i plus the count of subsequent true elements.

Translating Scatter. Scatters are only translatable when they are deterministic; since there are different ways of satisfying this condition, there are multiple translation rules. For example, at line 12 in Fig. 2 (**let** ys = scatter zeros idx xs), the indices idx used to update the destination array zeros are a permutation of zeros' indices. Semantically, $ys[\mathit{idx}[i]] = xs[i]$ for each index i and $|xs| = |\mathit{zeros}| = |\mathit{idx}|$. The index function layer exploits idx's invertibility to produce ys' index function in Fig. 4 which is bound to the *partition* function in the environment—ready for verification against its postcondition. The following rule formalizes this by verifying an equivalent property,

Bij idx $0..|xs|$ $0..|xs|$, which states that idx bijectively maps to $0..|xs|$ when restricted to valid destination indices ($idx|_{idx^{-1}(0..|xs|)}$ is bijective). This restriction captures scatter's semantics, which ignores out-of-bounds values in the source array. P² verifies this via the property layer ($\rightarrow_{\text{Prop}}$).

$$\textsc{Scatter2}\quad \frac{\Gamma \vdash \text{Bij } x_{idx}\ 0..|x_{dst}|\ 0..|x_{dst}| \rightarrow_{\text{Prop}} (\Gamma', \text{Yes}) \qquad \Gamma \vdash \text{Query } (|x_{idx}| = |x_{val}|) \rightsquigarrow_{\text{Q}} \text{Yes} \qquad \text{fresh } i}{\Gamma \vdash \text{scatter } x_{dst}\ x_{idx}\ x_{val} \rightarrow (\Gamma', \lambda\, (i : 0..|x_{dst}|)\ .\ [\,\text{true}\,] * x_{val}(x_{idx}^{-1}(i)))}$$

3.2 Property Layer

The property layer proves properties needed by the index function layer and verifies pre- and postconditions. It operates at two levels: proving properties using high-level reasoning (e.g., partition preserves range) and decomposing properties into low-level algebraic queries for the algebra layer.

Low-Level Algebraic Reasoning. To prove the Bij idx $0..|xs|$ $0..|xs|$ property required above, the property layer decomposes it into its proof obligation (i.e., its definition) and dispatches this to the algebra layer: injectivity ($\forall i, j \in 0..|xs|\ .\ i < j \Rightarrow idx(i) \neq idx(j)$) and surjectivity ($\forall i \in 0..|xs|\ .\ 0 \leq idx(i) < |xs|$). These are discharged by our algebraic solver (introduced in the next section), after which the bijection property is recorded into the environment.

After traversing the body and populating its environment with inferred facts, P² verifies the postcondition under the constructed context. The P² rule for verifying the $\lambda ys.$ Part ys xs $(\lambda i.\ p\ xs[i])$ postcondition of *partition* has two steps. First, P² checks that ys' index function has form $\lambda\, (i : 0..n)\ .\ [\,\text{true}\,] * xs(z^{-1}(i))$ for some index array z; here $z = idx$. Second, it proves the partition property by establishing the corresponding inverse-partition property on idx (shown in Fig. 7) via queries

$$\text{Bij } idx\ 0..|xs|\ 0..|xs| \qquad \text{(bijective)}$$

$$\begin{aligned}\forall i, j \in 0..|xs|\ .\ ((p(xs(i)) \lor \neg p(xs(i)))&\\ \land\ (i < j \land p(xs(i)) \land p(xs(j)) &\Rightarrow idx(i) < idx(j))\\ \land\ (i < j \land p(xs(i)) \land \neg p(xs(j)) &\Rightarrow idx(i) < idx(j)) \qquad \text{(partition)}\\ \land\ (i < j \land \neg p(xs(i)) \land \neg p(xs(j)) &\Rightarrow idx(i) < idx(j))\\ \land\ (i < j \land \neg p(xs(i)) \land p(xs(j)) &\Rightarrow idx(i) > idx(j)))\end{aligned}$$

The property layer proves each formula by querying the environment: (bijective) follows immediately from the environment, while (partition) is dispatched to the algebraic solver.

High-Level Property Reasoning. A key component of P²'s property reasoning is property propagation. Properties about range, injectivity, bijectivity, and filtering are all preserved under permutation. Since *partition* has the Part property in its postcondition (which P² verified), applying it to an input array xs (e.g., **let** ys = *partition* p xs) with any of these properties automatically propagates them to ys for further exploitation. Property propagation enables inference over uninterpreted functions by propagating index array properties to data arrays (e.g., in the above example, ys' index function may be uninterpreted) and reduces the annotation burden in general.

P² can extend compilers with advanced property-based reasoning, such as reasoning about partitions across conditionals (Appendix B.4.2) and verifying properties of uninterpreted expressions without direct propagation. We illustrate the latter using *get_smallest_edges* from the PBBS maximal matching algorithm [2]. This function filters an array *edges* based on arrays H and *edgeIds*:

$$\textbf{let } cs = \text{map } (\lambda i.\ H[edges[i]] = edgeIds[i])\ 0..|edges| \ \textbf{ in } \ \textbf{let } edges' = filter\ cs\ edges \ \textbf{ in } \ edges'$$

Given the precondition that *edgeIds* is injective (Inj $edgeIds$ $(-\infty)..\infty$), our goal is to prove the filtered output $edges'$ is also injective (postcondition Inj $edges'$ $(-\infty)..\infty$). Similar to *partition*

(Fig. 2), *filter*'s result is converted to an uninterpreted index function but has the postcondition $\text{Filt}\ \mathit{edges}'\ \mathit{edges}\ (\lambda i.\, H(\mathit{edges}(i)) = \mathit{edgeIds}(i))$. Because this restricts *edges*′ to elements satisfying $H[\mathit{edges}[i]] = \mathit{edgeIds}[i]$, P² can verify injectivity for *edges*′ by instead proving it over the satisfying elements of *edges*. This yields the augmented query:

$$\forall i, j \in 0..|\mathit{edges}|\ .\ H(\mathit{edges}(i)) = \mathit{edgeIds}(i) \wedge H(\mathit{edges}(j)) = \mathit{edgeIds}(j) \wedge \mathit{edges}(i) = \mathit{edges}(j) \Rightarrow i = j$$

Enriching this query with transitive equalities derived from $\mathit{edges}(i) = \mathit{edges}(j)$, yields $H(\mathit{edges}(i)) = H(\mathit{edges}(j))$ and, crucially, $\mathit{edgeIds}(i) = \mathit{edgeIds}(j)$. By the injectivity of *edgeIds*, $\mathit{edgeIds}(i) = \mathit{edgeIds}(j)$ implies $i = j$, completing the proof for *edges*′. The property layer implements this reasoning, using it to verify the maximal matching program in Section 5. (Detailed in Appendix D.3.)

3.3 Algebra Layer

P²'s algebra layer extends Fourier-Motzkin elimination [23, 75] with reasoning about sums, indexing and mutually exclusive boolean variables. Zooming in on the (partition) query above, P² substitutes applications of *idx* for its guarded expressions in Fig. 4, conjoining guards with the antecedent of the query. For example, for the last conjunct of the (partition) query, the system rewrites

$$(i < j \wedge \neg p(xs(i)) \wedge p(xs(j)) \Rightarrow idx(i) > idx(j))$$

by substituting *idx* for the guarded expressions in Fig. 4 (highlighted in green).

$$i < j \wedge \neg p(xs(i)) \wedge p(xs(j)) \wedge \neg p(xs(i)) \wedge p(xs(j)) \Rightarrow i + \sum_{k=i+1}^{|xs|-1}(p(xs(k))) > \sum_{k=0}^{j-1}(p(xs(k)))$$

This corresponds to Eq. (2)—the failing query in Section 2.1.2. To check this query, P² exploits its environment of properties, which are converted into ranges and equivalences to make them amenable to Fourier-Motzkin elimination. Performing this translation for the properties that are relevant to the third conjunct yields the following algebraic environment:

$$\overbrace{\begin{array}{l} 0 \le i < j < |xs| \\ \forall k\ .\ 0 \le p(xs(k)) \le 1 \end{array}}^{\text{Ranges}} \quad \overbrace{\begin{array}{l} p(xs(i)) = 0 \\ p(xs(j)) = 1 \end{array}}^{\text{Equivalences}} \quad \overbrace{i + \sum_{k=i+1}^{|xs|-1}(p(xs(k))) > \sum_{k=0}^{j-1}(p(xs(k)))}^{\text{Inequality to solve}} \tag{3}$$

Standard Fourier-Motzkin elimination fails to verify the inequality, yielding $i + 0 > j$ when minimizing the LHS and maximizing the RHS under the ranges. Notice that the LHS term i is an upper bound for $\sum_{k=0}^{i-1}(p(xs(k)))$, which overlaps with the RHS sum $\sum_{k=0}^{j-1}(p(xs(k)))$. Hence, the LHS is an upper bound on those terms on the RHS and also includes $p(xs(j)) = 1$ since $i < j$. The standard method cannot reason about overlapping sums nor exploit the equivalences in the environment because neither of the sums include the terms $p(xs(i))$ or $p(xs(j))$. P² uses a three-step tactic to eliminate overlap between sums and facilitate the use of equivalences and ranges:

Step 1 Extend sums to include terms with known equivalences.
Step 2 Simplify sums (e.g., eliminate overlap between terms of different signs and absorb terms).
Step 3 Split sums to separate out terms with known equivalences or more specialized ranges.

Steps 1–2 simplify Eq. (3) to $i + 1 + \sum_{k=j+1}^{|xs|-1}(p(xs(k))) - \sum_{k=0}^{i-1}(p(xs(k))) > 0$. Step 3 does nothing here, but the result is now solvable by Fourier-Motzkin: $j + 1 + 0 - j > 0 \iff 1 > 0$.

3.4 Segmented Parallel Operations

High-performance array languages typically only support regular arrays due to hardware mapping constraints; therefore jagged arrays must be represented as flat data arrays with auxiliary arrays encoding shape. For example, the *shape* (s) and *flag* auxiliary arrays for the jagged array $[[x_1, x_2], [], [x_3, x_4, x_5], [x_6, x_7, x_8]]$ are $[2, 0, 3, 3]$ and $[1, 0, 3, 0, 0, 4, 0, 0]$, where each non-zero element of the latter denotes the start of a row (thereby encoding the shape). Flag arrays are used

```
def flags (s : i64[] | Range s 0..∞)
          (x : []i64 | Equiv |s| |x|)
          : []i64 | λf. Equiv |f| (sum s) =
  let s_rotated = map (λi. ...) (0..|s|)
  let s_sum = scan (λx y. x + y) 0 s_rotated
  let idx = map (λsize i. ...) s s_sum
  let n =
    if |s| > 0 then s[|s| − 1] + s_sum[|s| − 1] else 0
  let zeros = map (λx. 0) (0..n)
  let f = scatter zeros idx x in f
```

(a) Computing the flag array.

1. $\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.$ $\begin{cases}[\,\sum_{j=0}^{i_1-1}(s(j)) + i_2 = 0 \lor f(\sum_{j=0}^{i_1-1}(s(j)) + i_2)\,] * y(\ldots)\\ [\,\sum_{j=0}^{i_1-1}(s(j)) + i_2 \neq 0 \land \neg f(\sum_{j=0}^{i_1-1}(s(j)) + i_2)\,] * (\circlearrowleft + y(\ldots))\end{cases}$

2. $\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.$ $\begin{cases}[\,i_2 = 0\,] * y(\sum_{j=0}^{i_1-1}(s(j)) + i_2)\\ [\,i_2 \neq 0\,] * (\circlearrowleft + y(\ldots))\end{cases}$

3. $\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.$ $\begin{cases}[\,i_2 = 0\,] * (i_1 + 1)\\ [\,i_2 \neq 0\,] * (\circlearrowleft + 0)\end{cases}$

4. $\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.\,[\,\mathsf{true}\,] * (i_1 + 1)$

(b) Rewrites of Eq. (6) where $f = y = flags$.

Fig. 5. P^2 infers and propagates flattened irregular structure.

to lift bulk-parallel operations to irregular arrays. For example, segmented scan [6]—which scans each jagged row—is implemented as a scan with lifted operator on the flag (f) and data arrays (y):

$$seg_sum\ f\ y = \mathsf{scan}\ (\lambda (f_1, y_1)\ (f_2, y_2).\ (f_1 + f_2, \mathsf{if}\ f_2 > 0\ \mathsf{then}\ y_2\ \mathsf{else}\ y_1 + y_2))\ (0, 0)\ (\mathsf{zip}\ f\ y) \quad (4)$$

In our context, one of the challenges of verifying flattened programs is establishing the connection between the data array and the auxiliary arrays they compute. To illustrate, Fig. 5a computes a flag array. The inferred index function, shown below, represents irregular arrays through a 2D index domain where the inner dimension depends on the index variable of the outer one:

$$flags = \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.\,[\,i_2 = 0\,] * x(i_1) + [\,i_2 \neq 0\,] * 0. \quad (5)$$

The syntax $\times$ denotes that the two dimensions map onto one flat dimension in the source program. This representation also captures empty rows: when $i_1 = 1$, then $0..s(i_1) = 0..0 = \emptyset$. The crucial point is that P^2 is able recover the original 2D irregular structure in the index function inferred from the flattened program. Indeed, we can see this explicitly by writing flags as an irregular nested array: $flags = \mathsf{map}\ (\lambda i.\ \mathsf{map}\ (\lambda j.\ \mathsf{if}\ j = 0\ \mathsf{then}\ x(i)\ \mathsf{else}\ 0)\ 0..s(i))\ 0..|s|$.

Inferring this two-dimensional structure from flattened source programs is essential for proving queries over index functions. Since these representations require flag arrays produced via scatter with data-dependent writes, P^2 infers this structure by a rule that matches scatters whose in-bounds indices are monotonically increasing (i.e., indices that define row offsets for non-empty rows).

Another auxiliary array is the *segment ids*: [1, 1, 3, 3, 3, 4, 4, 4], which can be computed with a segmented scan over the *flags* array. The index function inferred for *seg_sum* in Eq. (4) is

$$\lambda\,(i_0 : 0..|f|)\,.\,[\,i_0 = 0 \lor f(i_0) > 0\,] * y(i_0) + [\,i_0 \neq 0 \land f(i_0) \leq 0\,] * (\circlearrowleft + y(i_0)) \quad (6)$$

from which—in the case of segment ids—P^2 derives the index function as $\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.\,[\,\mathsf{true}\,] * (i_1 + 1)$ by substituting f and y with the *flags* array. Figure 5b shows key rewrites: (1) propagate *flags*' flat domain into Eq. (6), expressing i_0 as row offset $\sum_{j=0}^{i_1-1}(s(j))$ plus row index i_2; (2) substitute f and simplify; (3) substitute *flags* for y, yielding a recurrence with base cases at each irregular row and recurrent cases replicating the previous value. Details appear in Appendix D.1. This approach enables lifting *partition* over each row of a flat irregular array (Section 5.1.1).

4 Formalization

We begin the formalization by introducing common concepts.

Expressions e are polynomials over symbols s (Fig. 6). The index function and algebra layers both use this polynomial representation, but over different symbols s. Symbol and term

Variable	x, y, z, i, j, k	Term	$t ::= n \mid s \mid s \cdot t$	$t_1 + \ldots + t_n = \sum_{j=1}^{n} t_j = \begin{cases} t_1 \\ \vdots \\ t_n \end{cases}$
Constant	$n, m \in \mathbb{Z}$	Expression	$e ::= t \mid t + e$	

Fig. 6. Polynomial expression syntax parameterized by symbols s.

Property	Proof obligation
Range $x\ Y$	$\forall i \in 0..\lvert x\rvert\ .\ x(i) \in Y$
Mono $x \prec$	$\forall i, j \in 0..\lvert x\rvert\ .\ i < j \Rightarrow x(i) \prec x(j)$
Equiv $x\ y$	$\lvert x\rvert = \lvert y\rvert \wedge \forall i \in 0..\lvert x\rvert\ .\ x(i) = y(i)$
Inj $x\ Y$	$\forall i, j \in 0..\lvert x\rvert\ .\ x(i) \in Y \wedge x(i) = x(j) \Rightarrow i = j$ $\vee\ \forall i, j \in 0..\lvert x\rvert\ .\ x(i) \in Y \wedge x(j) \in Y \wedge i \neq j \Rightarrow x(i) \neq x(j)$
Bij $x\ Y\ Z$	Inj $x\ Y\ \wedge\ Z \subseteq Y\ \wedge\ \lvert Z\rvert = \lvert\{x(i) \in Y \mid i \in 0..\lvert x\rvert\}\rvert$
InvFiltPart $x\ Z\ p_f\ (p_1, \ldots, p_n)$	Bij $x\ Z\ Z\ \wedge\ \lvert Z\rvert = \sum_{j \in 0..\lvert x\rvert} p_f(j)$ $\wedge\ \forall q, r \in \{1, \ldots, n\}\ .\ \forall i, j \in 0..\lvert x\rvert\ .$ $\big(q \neq r \Rightarrow \neg p_q(i) \vee \neg p_r(i)\big)$ $\wedge\ \big(i < j \wedge q \leq r \wedge p_q(i) \wedge p_r(j) \wedge p_f(i) \wedge p_f(j) \Rightarrow x(i) < x(j)\big)$ $\wedge\ \big(i < j \wedge q > r \wedge p_q(i) \wedge p_r(j) \wedge p_f(i) \wedge p_f(j) \Rightarrow x(i) > x(j)\big)$
FiltPart $y\ x\ p_f\ (p_1, \ldots, p_n)$	InvFiltPart $z\ (0..\sum_{j \in 0..\lvert x\rvert} p_f(j))\ p_f\ (p_1, \ldots, p_n)$ where y's index function is of the form $\lambda\,\big(i : 0..\sum_{j \in 0..\lvert x\rvert} p_f(j)\big)\ .\ x(z^{-1}(i))$
Filt $y\ x\ p$	FiltPart $y\ x\ p\ (\lambda i.\,\text{true})$
Part $y\ x\ p$	FiltPart $y\ x\ (\lambda i.\,\text{true})\ (p, \lambda i.\,\neg p(i))$

Fig. 7. Array properties and the sufficient conditions to establish them.

order in expressions are syntactically irrelevant, and expression multiplication is defined by distribution over addition. For example: $x \cdot (y + x \cdot (1 + y)) = x \cdot y + x \cdot x \cdot (1 + y) = x \cdot y + x^2 + x^2 \cdot y$. Sums $\sum$ and case syntax $\{$ are shorthand for addition of terms (e.g., used for *idx*'s index function in Fig. 4), and $e_1 - e_2$ is sugar for $e_1 + (-1) \cdot e_2$.

Y, Z are used to range over contiguous integer sets. For example, $Y = 0..n$ ranges over 0 to $n - 1$.

$x|_Y$ is the restriction of index function x to a smaller domain Y: $x|_Y$ is a new index function identical to the original index function x except it is defined only over domain $Y \subseteq \text{dom}(x)$.

$x|_{x^{-1}(Y)}$ is the restriction of index function x to the preimage of a (smaller) codomain Y: $x|_{x^{-1}(Y)}$ is a new index function identical to the original index function x except it is defined only over domain $x^{-1}(Y) = \{i \in \text{dom}(x) \mid x(i) \in Y\}$.

Environments (Γ) map variables to index functions and properties, with subenvironments Γ_{Def} for function definitions, Γ_{Ixfn} for index functions, Γ_{Range} for ranges, and so on for each property in Fig. 7. Unbound variables map to $\varnothing$. We write $\Gamma, x \mapsto f$ to extend Γ_{Ixfn} mapping x to f, and $\Gamma, \text{Range}\ x\ 0..e$ to extend Γ_{Range}, etc. Environments are extended with predicates: $\Gamma, e_1 = e_2, 0 \leq i < n$ adds equivalences derived from $e_1 = e_2$ and ranges derived from $0 \leq i < n$.

$\Gamma \vdash \text{Query}\ (p)$ asks whether p holds under Γ. Queries are lowered to a more restricted language and solved by an adaptation of Fourier-Motzkin Elimination, yielding Yes or Unknown.

$e_1[x := e_2]$ substitutes e_2 for x in e_1.

$\text{fv}(\cdot)$ and $\text{bv}(\cdot)$ denote free and bound variables of an object, respectively.

4.1 Array Properties

P^2 proves properties from a small, curated set to ensure tractable reasoning. Figure 7 presents each property and its *proof obligation*—a conjunction of properties and/or algebraic (in)equalities

that must be verified to establish the property. Properties are expressed over (array) variables but reasoned about using their corresponding index functions.

Range x Y says that the values of array x are in Y. Mono $x \prec$ says that array x's values are ordered according to the relation $\prec$. Equiv x y says that the index functions of x and y are equivalent. Inj x Y says that index function $x|_{x^{-1}(Y)}$ is injective. Bij x Y Z says that index function $x|_{x^{-1}(Y)}$ is bijective and the image of $x|_{x^{-1}(Y)}$ is Z (a subset of Y). No indices are mapped into $Y - Z$. Bij x $0..\infty$ $0..n$ specifies that the indices mapping to non-negative values ($Y = 0..n$) form a bijection to $Z = 0..n$.

Since filtering and partitioning commute, they're unified into one property. Predicates p map indices to booleans. FiltPart y x p_f $(p_1, \ldots, p_n)$ says that y is equivalent to x with indices filtered by p_f and $n + 1$-way partitioned by $(p_1, \ldots, p_n)$, yielding an index function of the form $x(z^{-1}(i))$ that the proof obligation matches on. InvFiltPart x Z p_f $(p_1, \ldots, p_n)$ captures scatter/gather behavior with index array x. For arrays x, y, z, if x satisfies this property and $Z = 0..|z|$, scatter z x y is equivalent to filtering y by p_f and partitioning it by $(p_1, \ldots, p_n)$. P^2 verifies two conditions: (1) p_f holds for exactly $|Z|$ indices in x. (2) $x|_{x^{-1}(Z)}$ permutes Z such that each partition occupies a contiguous range: indices satisfying $p_f \wedge p_1$ map to $0, \ldots, \sum_{i \in Z}(p_f(i) \wedge p_1(i)) - 1$; indices satisfying $p_f \wedge p_2$ map to $\sum_{i \in Z}(p_f(i) \wedge p_1(i)), \ldots, \sum_{i \in Z}(p_f(i) \wedge p_1(i)) + \sum_{i \in Z}(p_f(i) \wedge p_2(i)) - 1$; and so on for each p_j. If $y = [0, \ldots, |x| - 1]$, gathering with the scatter result z as indices yields identical filter/partitioning behavior. Filt y x p and Part y x p are aliases for filtering and 2-way partitioning.

4.2 Algorithm Sketch Highlighting the Interaction Between Components

The source language is in A-normal form [56]. Variable names are unique. Syntactic category E consists of non-nested let expressions and $E^{\oslash}$ of bound expressions, i.e., $E ::= x \mid \textbf{let } x = E^{\oslash} \textbf{ in } E$. Fig. 8 (on the next page) sketches a deterministic static-analysis algorithm that terminates at the first failure to verify annotated properties or index/scatter memory safety. The algorithm targets the common case at reasonable compilation time and is not intended to be complete. Relations $\rightsquigarrow$ apply rules, disambiguated by type, to a fixed point. Rule FunDef verifies a program's function definitions in order. It parses pre- and postconditions ($\rightarrow_{\text{Parse}}$, not shown), and assumes preconditions while it infers the index function of the body (a let expression) and tries to prove the postcondition ($\rightarrow_{\text{Prop}}$). If successful, Γ_{Def} is extended with the pre/postconditions and the result's index function; if the free variable check on f_2 fails, uninterpreted functions are used for its domain/guards (not shown). Unprovable postconditions end in failure (FunDefFail) and empty programs trivially verify.

The Let rule processes bindings standardly while propagating properties (e.g., via Γ_1') and automatically inferring new ones via the property layer relation $\rightarrow_{\text{Infer}}$. Because the source language is in ANF, Γ_1' introduces no new index functions but may add properties on in-scope variables (e.g., via App). Bindings persist in the returned environment to verify postconditions in FunDef, and are discarded only at scope boundaries (e.g., function definition, map, scan, loop, if). Inference via $\rightarrow_{\text{Infer}}$ uses computationally cheap high-level rules—that are tried exhaustively to a fixpoint—and never fails (InferNothing). Examples include transferring function postconditions to bound variables (AppProp) and preserving monotonicity and injectivity across filtering operations.

The property layer relation $\rightarrow_{\text{Prop}}$ verifies user annotations and rule premises. It applies matching rules in increasing order of cost until one succeeds. For example, before trying InjNeq, it looks up x in Γ_{Inj} to see if it is already proven (over a larger codomain) and attempts cheaper high-level rules.

Finally, function calls, array indexing, scatter and loops may fail to convert ($E^{\oslash}$-Fail). Index function inference rules are prioritized, applying only the first match. For example, filter operations use a scatter whose in-bounds indices are both a permutation of the target domain and strictly monotonic, matching two rules. We prioritize the permutation rule because it is highly unlikely that the intent is to create a jagged array whose non-empty segments all consist of one element.

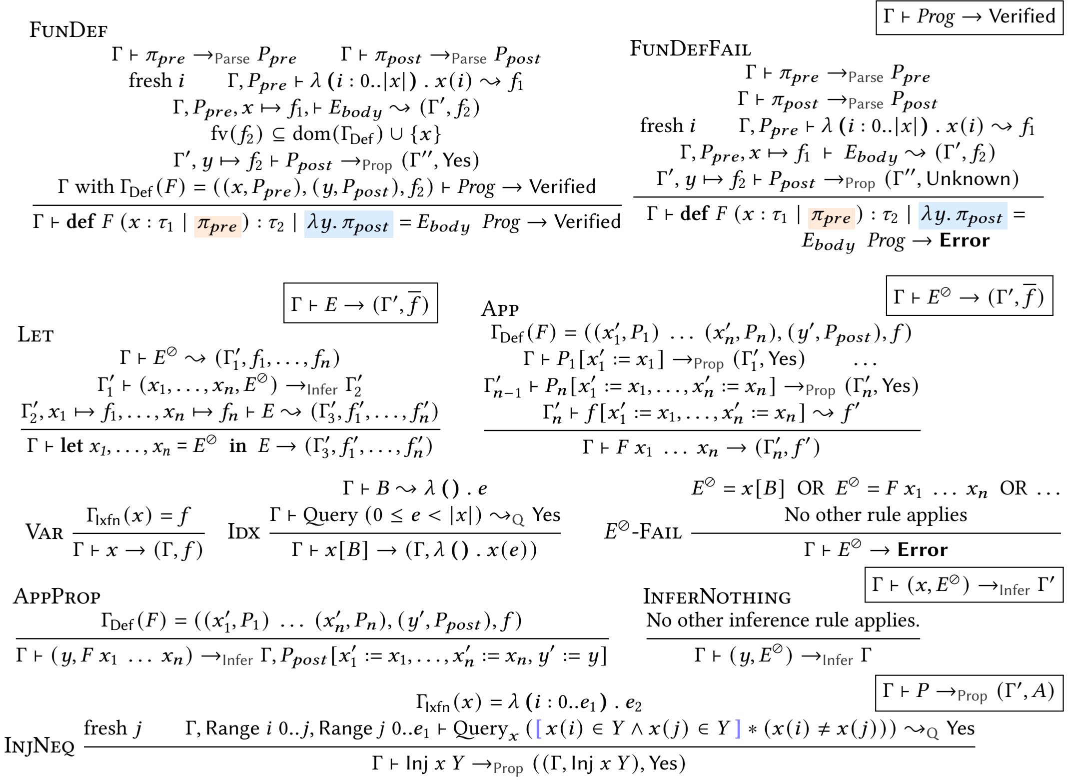


Fig. 8. Algorithm sketch illustrating the components' judgment forms, interaction and rule instances.

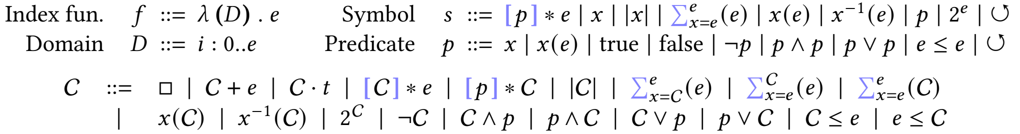


Fig. 9. Index function and expression syntax. Reduction context grammar (C).

4.3 Index Function Layer

Index functions have the form $\lambda\,(D)\;.\;e$ where D is the domain and e is a guarded expression (Fig. 9). Guarded expressions are polynomials (i.e., sums of products) defined piecewise by guards—predicates p in Iverson brackets $[\;\cdot\;]$—that must partition the domain, though this is not enforced syntactically (see Section 3.1 for details). Scalars are single element arrays: $\lambda\,()\;.\;g$ abbreviates $\lambda\,(i : 0..1)\;.\;g$. We may omit writing trivial guards and coefficients: $[\,\text{true}\,] * t = t$ and $[\,p\,] * 1 = [\,p\,]$. Multiplying guards yields logical conjunction ($[\,p_1\,] * 1 \cdot [\,p_2\,] * e_1 = [\,p_1 \wedge p_2\,] * e_1$) because a term is non-zero only if both guards hold, resulting in an expression where each term has at most one guard. Together with expression multiplication, this combines guards into a new partition of the domain:

$$([\,x\,]*y+[\,\neg x\,]*1)\cdot([\,z\,]*0+[\,\neg z\,]*y) = [\,x\wedge z\,]*0+[\,x\wedge\neg z\,]*y^2+[\,\neg x\wedge z\,]*2+[\,\neg x\wedge\neg z\,]*y \quad (7)$$

Comparisons are syntactic sugar: $e_1 \in 0..e_2$ and $0 \leq e_1 < e_2$ are both $(0 \leq e_1) \wedge (e_1 + 1 \leq e_2)$. Inference rules match to expressions and index functions via unification with bound variables [60].

$$\boxed{\Gamma \vdash E^{\oslash} \rightarrow (\Gamma', \overline{f})}$$

$$\textsc{Map}\ \frac{\Gamma(x_2) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \Gamma, \text{Range } i\ 0..e_1, x_1 \mapsto \lambda\,()\,.\,e_2 \vdash E \rightsquigarrow (\Gamma', \lambda\,()\,.\,e_3)}{\Gamma \vdash \text{map } (\lambda x_1 . E)\ x_2 \longrightarrow (\Gamma, \lambda\,(i : 0..e_1)\,.\,e_3)}$$

$$\textsc{Scan}\ \frac{\Gamma(x_3) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \Gamma, x_2 \mapsto \lambda\,()\,.\,e_2, \text{Range } i\ 0..e_1 \vdash E_1 \rightsquigarrow (\Gamma', \lambda\,()\,.\,e_3)}{\Gamma \vdash \text{scan } (\lambda x_1\ x_2 . E_1)\ E_2\ x_3 \longrightarrow (\Gamma, \lambda\,(i : 0..e_1)\,.\,[\,i = 0\,] * e_2 + [\,i \neq 0\,] * e_3[x_1 := \circlearrowleft])}$$

$$\textsc{WhileLoop}\ \frac{\begin{array}{c}\Gamma_{\text{Def}}(F) = (\overline{(x', P_{pre})}^{(n)}, (\overline{y}^{(n)}, \overline{P_{post}}^{(n)}), _) \qquad \Gamma_{\text{Ixfn}}(x_{init_n}) = \lambda\,()\,.\,[\,p\,] * 1 \qquad \text{fresh } \overline{i}^{(n)}, \overline{z}^{(n)} \\ \Gamma, p \vdash F\ \overline{x_{init}}^{(n)} \longrightarrow (\Gamma', \overline{f}^{(n)}) \qquad \forall j \in 1, \ldots, n\,.\, P_{pre_j} \text{ unifies with } P_{post_j}[y_1 := x_1', \ldots, y_n := x_n']\end{array}}{\Gamma \vdash \textbf{loop } \overline{x}^{(n)} = \overline{x_{init}}^{(n)} \textbf{ while } x_n \textbf{ do } F\ \overline{x}^{(n)} \longrightarrow (\Gamma, \lambda\,(i_1 : 0..|z_1|)\,.\,z_1(i), \ldots, \lambda\,(i_n : 0..|z_n|)\,.\,z_n(i))}$$

$$\boxed{\Gamma \vdash f \rightarrow f}$$

$$\textsc{RecSum}\ \frac{\circlearrowleft \text{ does not occur in } e_2 \text{ nor in } \sum_j t_j \qquad \text{fresh } x}{\Gamma \vdash \lambda\,(i : 0..e_1)\,.\,[\,i = 0\,] * e_2 + [\,i \neq 0\,] * (\circlearrowleft + \sum_j t_j) \rightarrow \lambda\,(i : 0..e_1)\,.\,e_2[i := 0] + \sum_j \sum_{x=1}^{i} (t_j[i := x])}$$

$$\textsc{Hoist}\ \frac{\Gamma \vdash [\,\bigvee_j p_j\,] \rightsquigarrow [\,\text{true}\,] \qquad \text{bv}(C) \cap (\bigcup_j \text{fv}(p_j)) = \emptyset}{\Gamma \vdash C\langle \sum_j [\,p_j\,] * e_j \rangle \rightarrow \sum_j [\,p_j\,] * 1 \cdot (C\langle e_j \rangle)} \qquad \textsc{Sub}\ \frac{\Gamma(x) = \lambda\,(i : 0..e_2)\,.\,e_3}{\Gamma \vdash C\langle x(e_1) \rangle \rightarrow C\langle (e_3[i := e_1]) \rangle} \qquad \boxed{\Gamma \vdash e \rightarrow e}$$

Fig. 10. Converting the source language to index functions (selected rules). Rewrites (selected rules).

Source Language Conversion. Figure 10 shows rules for converting the source language to index functions. The judgment $\Gamma \vdash E \longrightarrow (\Gamma', \overline{f})$ says that, under environment Γ, $E^{\oslash}$ has index functions $\overline{f}$ and produces environment Γ'. Arrays are assumed to be one-dimensional (this restriction is lifted in Section 4.3.1). Map and Scan convert the corresponding SOACs by binding the lambda's argument to the array argument's index function with the outer dimension dropped, extending the environment with the now-captured index variable i's range, and then deriving the lambda body. For Scan, two guards are introduced for the base and recursive cases, where the accumulator is replaced by the recurrence symbol $\circlearrowleft$ (demonstrated in Section 3.1). Index functions introduced by the SOAC rules inherit the array argument's domain, so that array shapes are propagated from the formal arguments of a function. Untranslatable expressions are bound to uninterpreted functions, i.e., to index functions that index a fresh name over some domain (possibly of unknown size).

Rule WhileLoop treats while loops where loop variables $\overline{x}^{(n)} = x_1, \ldots, x_n$ are initialized with $\overline{x_{init}}^{(n)}$ and are updated to the result of the loop body for the next iteration. Our source language restricts loop bodies to just function application—all loops can be (automatically) rewritten this way. The rule looks up pre- and postconditions in Γ_{Def} and verifies that they unify and that the preconditions hold on $\overline{x_{init}}^{(n)}$. If so they will be inductively satisfied by the values of $\overline{x}^{(n)}$ of any iteration. We assume that loops terminate. The loop result is uninterpreted and properties on the loop variables can be transferred to the loop result only if they are expressible in terms of the in-scope variables (by a rule similar to AppProp). The complete conversion rules are in Appendix B.3.

Expression rewriting. We formalize expression rewrite rules $\Gamma \vdash e \longrightarrow e'$ using expression contexts. Expression contexts C define where a subexpression occurs in an expression (shown in Fig. 9). A context contains a single hole $\square$, where $C\langle e \rangle$ denotes substituting the subexpression e into that hole. For example, given $C = [\,p_1\,] * e_1 + [\,p_2\,] * x_1(\square)$, then $C\langle e_2 \rangle$ is $[\,p_1\,] * e_1 + [\,p_2\,] * x_1(e_2)$.

Context matching over binary operators is inherently nondeterministic. For example, matching $C\langle x(i) \rangle$ against $x_1(i) + x_2(i)$ yields valid contexts $x_1(i) + \square$ and $\square + x_2(i)$. We resolve this by enforcing a strict lexicographical evaluation order on AST nodes: variables by name, then symbols, terms, and expressions by order of appearance in their respective grammars. Consequently, $\square + x_2(i)$ unambiguously matches first because x_1 precedes x_2. We apply this canonical ordering across all expression $(C + t)$, term $(C \cdot t)$, and predicate contexts $(C \leq e, \ldots)$.

Figure 10 shows normalizing rewrite rules that are applied, in order of appearance, to a fixed point after each source language conversion step. RecSum rewrites recurrences introduced by scan into

$$\textsc{Scatter3}\quad \boxed{\Gamma \vdash E^{\oslash} \longrightarrow (\Gamma', \overline{f})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Inj}\ x_{idx}\ 0..|x_{dst}| \longrightarrow_{\mathrm{Prop}} (\Gamma', \mathsf{Yes}) \qquad \text{fresh } x_{\perp}, k, l, i_1, i_2 \\ \Gamma' \vdash \lambda\ (i_1 : 0..|x_{idx}|)\ .\ \begin{cases}[\,x_{idx}(i_1) \in 0..|x_{dst}|\,] * x_{idx}(i_1) \\ [\,x_{idx}(i_1) \notin 0..|x_{dst}|\,] * x_{\perp}\end{cases} \rightsquigarrow \lambda\ (i_1 : 0..e_{|x_{idx}|})\ .\ \begin{cases}[\,p\,] * e_{row} \\ [\,\neg p\,] * x_{\perp}\end{cases} \\ \Gamma' \vdash \mathrm{Query}\ (e_{row}[i_1 := 0] = 0) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \qquad \Gamma' \vdash \mathrm{Query}\ (e_{row}[i_1 := |x_{idx}|] = |x_{dst}|) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \\ \Gamma', 0 \le j < k < |x_{idx}| \vdash \mathrm{Query}\ (0 \le e_{row}[i_1 := j] \le e_{row}[i_1 := k] \le |x_{dst}|) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{src} \longrightarrow \left(\Gamma', \lambda\ (i_1 : 0..|x_{idx}| \times i_2 : 0..(e_{row}[i_1 := i_1 + 1] - e_{row}))\ .\ \begin{cases}[\,i_2 = 0 \wedge p\,] * x_{src}(i_1) \\ [\,i_2 \neq 0 \vee \neg p\,] * x_{dst}(e_{row} + i_2)\end{cases}\right)}$$

$$\boxed{\Gamma \vdash f \longrightarrow f}$$

$$\textsc{PropFlatten}\ \frac{\begin{array}{c}\Gamma(x) = \lambda\ (i_2 : 0..e_2 \times i_3 : 0..e_3)\ .\ e_x \\ \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_2 - 1}(e_3[i_2 := j]) \rightsquigarrow e_{row} \qquad \Gamma \vdash \mathrm{Query}\ (e_1 = e_{row}[i_2 := e_2]) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \lambda\ (i_1 : 0..e_1)\ .\ C\langle x(e_{idx})\rangle \longrightarrow \lambda\ (i_2 : 0..e_2 \times i_3 : 0..e_3)\ .\ C\langle x(e_{idx})\rangle[i_1 := e_{row} + i_3]}$$

$$\textsc{Flatten}\ \frac{\Gamma(x) = \lambda\ (i_1 : 0..e_1, i_2 : 0..e_2, D)\ .\ e_3}{\Gamma \vdash \mathsf{flatten}\ x \longrightarrow (\Gamma, \lambda\ (i_1 : 0..e_1 \times i_2 : 0..e_2, D)\ .\ e_3)}$$

$$\textsc{SegRecSum}\ \frac{\circlearrowleft \text{ does not occur in } e_3 \text{ nor in } \sum_j t_j \qquad \text{fresh } x}{\begin{array}{c}\Gamma \vdash \lambda\ (i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ [\,i_2 = 0\,] * e_3 + [\,i_2 \neq 0\,] * (\circlearrowleft + \sum_j t_j) \\ \longrightarrow \lambda\ (i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ e_3[i_2 := 0] + \sum_j \sum_{x=1}^{i_2}(t_j[i_2 := x])\end{array}}$$

Fig. 11. Rules for the multi-dimensional flattened (segmented) case.

closed-form sums, as seen in rewrite (2) of *n_right* in Fig. 4. The rule requires that $\circlearrowleft$ appears exactly once in the recurrence step and not in the base case ($[\,i = 0\,]$) or summed terms. (Note $\sum$ is distinct from the sum symbol $\sum$.) Sub substitutes index functions into other index functions by reduction over indexing symbols, yielding nested guarded expressions. Multiplication of expressions and guards naturally distributes nested guards over the outer guards, while preserving the mutually exclusive and collectively exhaustive (MECE) property as shown in Eq. (7). Hoist moves guards that are nested inside a symbol to the root of the surrounding expression (the context C), provided that no guard depends on a variable bound in C. The MECE property of guards ensures the transformation's validity: exactly one term in the sum $\sum_j [\,p_j\,]$ equals 1 while all others equal 0. Together, Sub and Hoist enable P^2 to track positional dependencies backwards to the formal arguments of a top-level definition. Appendix B.3.6 includes rewrites that simplify terms under assumption of their guards, join guards in disjunction when all of their terms are equivalent under either of the guards, query predicates to prove them true, and falsify and eliminate contradictory guards, and similarly.

4.3.1 Multi-Dimensional Arrays. The source language allows multi-dimensional arrays with reshaping via flatten : $[][]\alpha \rightarrow []\alpha$, which collapses two dimensions into one. To support this, we augment the index function grammar (Fig. 9) with multiple arguments and flattened dimensions:

$$D ::= i : 0..e \mid i : 0..e, D \mid i : 0..e \times D \qquad s ::= \ldots \mid x(e, \ldots, e) \mid \%_D(e_{idx}) \mid \ldots$$

where $i : 0..e \times D$ denotes multiple dimensions flattened into one, and $\%_D(e_{idx})$ is used to rewrite a 1D index e_{idx} in terms of the multi-dimensional structure inferred for the flattened domain D. Because our solver does not support non-linear arithmetic, we rely on three specialized, division-free rules to convert $\%_D(e_{idx})$ to 2D indices (Appendix B.3.8). Rules Flatten and PropFlatten (Fig. 11) use the $\times$ syntax to preserve and propagate shape information about flattened arrays. Existing rules (Figs. 8 and 10) require few changes: Var creates domains matching array shapes, Idx checks bounds per dimension, other rules require minor arity changes to match array ranks (which we require to be statically known), and flat and multi-dimensional domains must be matched alike.

As explained in Section 3.4, we infer structure from flattened programs via scatter. Rule Scatter3 (Fig. 11) covers the case where in-bounds written indices (x_{idx}) are strictly monotonically increasing,[4] producing a flat jagged array representation where rows start at the written indices. The number of

[4]Checked by the injectivity of in-bounds indices of x_{idx} and the last query (establishing non-strict monotonicity of e_{row}).

rows is given by the index array length. Importantly, this allows for empty rows, which correspond to out-of-bound indices when $[\neg p]$ implies that the row is empty: $e_{row}[i_1 := i_1 + 1] - e_{row} = 0$ (or equivalently, row non-emptiness implies $[p]$). In Section 3.4 and Fig. 5a, Scatter3 infers the $flags$ index function: $\lambda\ (i_1 : 0..|s| \times i_2 : 0..s(i))\ .\ [i_2 = 0] * x(i_1) + [i_2 \neq 0] * 0$. The predicate p is $s(i) > 0$ here and was simplified away since it is equivalent to row non-emptiness. To derive the *segment ids* (Fig. 5b), PropFlatten (Fig. 11) propagates this jagged structure to the segmented scan result, and SegRecSum eliminates the recurrence $\circlearrowleft$.

4.3.2 *Concatenation.* Source-level concatenation of arrays x and y yields the index function:

$$z = \lambda\ (i_1 : 0..(|x| + |y|))\ .\ [i_1 < |x|] * x(i_1) + [i_1 \geq |x|] * y(i_1 - |xs|)$$

We also record a symbolic form $z = f_x \mathbin{++} f_y$ where f_x and f_y are x and y's index functions. The index function is used to substitute indexing $z(e)$, while the symbolic form enables decompositional reasoning: injectivity and monotonicity are verified part-wise rather than over the combined domain. Scatter3 requires that the first written index is 0 and the last is the destination array's length (out of bounds). This can be relaxed by concatenating index functions whose domains range over elements before the first write $(0..e_{row}[i_1 := 0])$ and elements after the last write $(e_{row}[i_1 := |x_{idx}|]..|x_{dest}|)$. Eliminating concatenations that have empty domains (checked by a query on the range bounds) makes the resulting index function invariant to whether e_{row} uses inclusive or exclusive scan.

4.4 Property Layer

Properties are verified either by expanding the proof obligations listed in Fig. 7 into (in)equalities discharged by the algebra layer (low-level reasoning), or by reasoning in terms of other properties in the environment (high-level reasoning).

Low-Level Algebraic Reasoning. Proof obligations are reified as solver queries. For example, given an index function x with n guards, $x = \lambda\ (i : 0..e_{dom})\ .\ [p_1] * e_1 + \ldots + [p_n] * e_n$, the query in InjNeq (Fig. 8), Query $([x(i) \in Y \wedge x(j) \in Y] * (x(i) \neq x(j)))$, is specialized to generate n^2 subqueries by substituting the index function, hoisting guards (via Sub and Hoist in Fig. 10), and distributing outer guards over inner ones as demonstrated in Eq. (7):

$$\begin{aligned}&[p_1 \wedge p_1[i := j] \wedge e_1 \in Y \wedge e_1[i := j] \in Y] * (e_1 \neq e_1[i := j])\\ &+ \ldots + [p_1 \wedge p_n[i := j] \wedge e_1 \in Y \wedge e_n[i := j] \in Y] * (e_1 \neq e_n[i := j])\\ &+ \ldots + [p_n \wedge p_n[i := j] \wedge e_n \in Y \wedge e_n[i := j] \in Y] * (e_n \neq e_n[i := j])\end{aligned}$$

Another rule that proves injectivity by $\forall i, j \in 0..|x|\ .\ x(i) = x(j) \Rightarrow i = j$ is treated similarly and solved by saturating transitive equalities, as demonstrated in Section 3.2 (and not detailed further).

The proof obligations of Range, Equiv, Mono, and Inj generalize to n dimensions by replacing i and j with n-length vectors $\vec{i}$ and $\vec{j}$ and using lexical ordering. For example, the proof obligation for Mono $x \prec$ expands to $2^n - 1$ distinct queries so that for all $\vec{i}$ and $\vec{j}$ in the domain of x:

$$\vec{i} < \vec{j} \Rightarrow x(\vec{i}) \prec x(\vec{j}) \quad \equiv \quad \begin{aligned}&(i_1 < j_1 \Rightarrow x(\vec{i}) \prec x(\vec{j})) \wedge (i_1 = j_1 \wedge i_2 < j_2 \Rightarrow x(\vec{i}) \prec x(\vec{j}))\\ &\wedge \cdots \wedge (i_1 = j_1 \wedge \cdots \wedge i_{n-1} = j_{n-1} \wedge i_n < j_n \Rightarrow x(\vec{i}) \prec x(\vec{j}))\end{aligned}$$

where omitted dimensions further expand to all possible combinations of $<$ and $\geq$. For example, $\vec{i_2} < \vec{j_2}$ and $\vec{i_2} \geq \vec{j_2}$ are both tried in the first conjunct of the lexical expansion. Flat arrays are similar.

High-Level Property Reasoning. Figure 12 shows the rules for both property layer relations $\rightarrow_{\text{Prop}}$ and $\rightarrow_{\text{Infer}}$ relations. For $\rightarrow_{\text{Prop}}$, premises like Inj x Y may use either level of reasoning. InjSubset concludes that $x|_{x^{-1}(Y_1)}$ is injective if $Y_1 \subseteq Y_2$ and $x|_{x^{-1}(Y_2)}$ is injective. FiltPartPresInj propagates injectivity over filter/partitioning. FilterInj proves injectivity of x_1 via x_2, which x_1 filters/partitions.

$$\boxed{\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma', A)}$$

InjSubset
$$\frac{\Gamma_{\text{Inj}}(x) = Y_2 \qquad \Gamma \vdash \text{Query } (Y_1 \subseteq Y_2) \rightarrow_{\text{Q}} \text{Yes}}{\Gamma \vdash \text{Inj } x\ Y_1 \rightarrow_{\text{Prop}} ((\Gamma, \text{Inj } x_1\ Y_1), \text{Yes})}$$

FiltPartPresInj
$$\frac{\Gamma_{\text{FiltPart}}(x_1) = (x_2, f_1, (f_2, \ldots, f_n)) \qquad \Gamma \vdash \text{Inj } x_2\ Y \rightarrow_{\text{Prop}} (\Gamma', \text{Yes})}{\Gamma \vdash \text{Inj } x_1\ Y \rightarrow_{\text{Prop}} ((\Gamma, \text{Inj } x_1\ Y), \text{Yes})}$$

FilterInj
$$\frac{\begin{array}{c}\Gamma_{\text{FiltPart}}(x_1) = (x_2, \lambda\,(i_1 : 0..e_1)\,.\,p, (f_1, \ldots, f_n)) \qquad \Gamma(x_2) = \lambda\,(i_2 : 0..e_1)\,.\,e_2 \\ \text{fresh } x_3 \qquad \Gamma \vdash \lambda\,(i_2 : 0..e_1)\,.\,[\,p[i_1 := i_2]\,] * x_2(i_2) + [\,\neg p[i_1 := i_2]\,] * \bot \rightsquigarrow f \qquad \Gamma, x_3 \mapsto f \vdash \text{Inj } x_3\ Y \rightarrow_{\text{Prop}} (\Gamma', \text{Yes})\end{array}}{\Gamma \vdash \text{Inj } x_1\ Y \rightarrow_{\text{Prop}} ((\Gamma, \text{Inj } x_1\ Y), \text{Yes})}$$

InjConcat
$$\frac{\begin{array}{c}\Gamma(x_1) = f_1 \mathbin{++} f_2 \qquad \text{fresh } x_2, x_3 \qquad \Gamma, x_2 \mapsto f_1 \vdash \text{Inj } x_2\ Y \rightarrow_{\text{Prop}} (\Gamma', \text{Yes}) \qquad \Gamma, x_3 \mapsto f_2 \vdash \text{Inj } x_3\ Y \rightarrow_{\text{Prop}} (\Gamma'', \text{Yes}) \\ \text{fresh } i, j \qquad \Gamma, x_2 \mapsto f_1, x_3 \mapsto f_2, \text{Range } i\ 0..e_1, \text{Range } j\ 0..e_2 \vdash \text{Query } ([\,x_2(i) \in Y \wedge x_3(j) \in Y\,] * x_2(i) \neq x_3(j))\end{array}}{\Gamma \vdash \text{Inj } x_1\ Y \rightarrow_{\text{Prop}} ((\Gamma, \text{Inj } x_1\ Y), \text{Yes})}$$

$$\boxed{\Gamma \vdash (x, E^{\oslash}) \rightarrow_{\text{Infer}} \Gamma'}$$

InferPart
$$\frac{\begin{array}{c}\Gamma(z) = \lambda\,(i_1 : 0..e_{|x_{dst}|})\,.\,[\,\text{true}\,] * x_{val}(x_{idx}^{-1}(i_1)) \qquad \Gamma_{\text{Bij}}(x_{idx}) = \text{Bij } Y\ Y \\ \Gamma(x_{idx}) = \lambda\,(i : 0..e_{|x_{dst}|})\,.\,[\,p_1\,] * e_1 + \ldots + [\,p_n\,] * e_n \qquad \text{fresh } \overline{j}^{(q+1)} \\ f_0 = \lambda\,(j_0 : 0..e_{|x_{dst}|})\,.\,\text{true} \qquad \forall q = 1 \ldots n : \ f_q = \lambda\,(j_q : 0..e_{|x_{dst}|})\,.\,p_q[i := j_q] \\ \Gamma \vdash \text{InvFiltPart } x_{ind}\ Y\ f_0\ (f_1, \ldots, f_n) \rightarrow_{\text{Prop}} (\Gamma', \text{Yes})\end{array}}{\Gamma \vdash (z, \text{scatter } x_{dst}\ x_{idx}\ x_{val}) \rightarrow_{\text{Infer}} \Gamma, \text{FiltPart } z\ x_{val}\ f_0\ (f_1, \ldots, f_n)}$$

FiltPartPresInj'
$$\frac{\begin{array}{c}\Gamma_{\text{FiltPart}}(x_1) = (x_2, f_1, (f_2, \ldots, f_n)) \\ \Gamma_{\text{Inj}}(x_2) = \text{Inj } x_2\ Y\end{array}}{\Gamma \vdash (x_1, _) \rightarrow_{\text{Infer}} \Gamma, \text{Inj } x_1\ Y}$$

Fig. 12. Verifying properties using other properties (selected rules).

The filter predicate is used to refine the values of x_2, possibly ignoring those that would violate injectivity: $\forall i, j \in 0..|x_2|\,.\,p(i) \wedge p(j) \wedge x_2(i) \in Y \wedge x_2(j) \in Y \wedge x(i) = x(j) \implies i = j$. We reify this as an index function to leverage both reasoning levels rather than directly invoking the solver. ($\bot$ is some sentinel value not in Y.) These three rules work even when x_1 is uninterpreted (a formal argument or untranslatable source expression). InjConcat verifies injectivity via concatenated functions, requiring that the functions' images are disjoint. Additional rules in Appendix B.4 include: filtering preserves range and monotonicity; partitioning preserves bijectivity; bijectivity implies injectivity; range inference over filtering/partitioning (like FilterInj); monotonicity over concatenation.

Properties can also be inferred from the source code (e.g., combining the ranges of each branch in a conditional). The last two rules belong to the $\rightarrow_{\text{Infer}}$ relation, which aims to automate inference, but relies on $\rightarrow_{\text{Prop}}$ to provide a base case, e.g., in FiltPartPresInj' the restricted codomain Y is not given and is determined by looking up Γ_{Inj}. InferPart infers a partitioning property by pattern matching the index function produced by Scatter2, which guarantees that the result has been proven to be (at least) a permutation of x_{val}; the rule concludes by checking InvFiltPart on x_{ind}.[5] These two rules can infer both the partitioning and injectivity of z = scatter $x_{dst}\ x_{ind}\ x_{val}$ (in the Let rule of Fig. 8). We allow properties to be expressed over the rows of arrays by introducing the property, For $(i_1 : 0..e_1)\ P$, where P is a property from Fig. 7 that may depend on i_1. P^2 verifies For properties by creating a new index function where the outer dimension is dropped, and then verifies P over this function where i_1 is a free variable.[6] (Shown in Appendix B.4.1.) The For property composes to express properties over inner dimensions.

4.5 Algebra Layer

The algebra layer solves (in)equalities over the guarded expressions of index functions by lowering the expressions to a simplified algebraic language (Fig. 13). The algebraic symbols s include variables, array indexing and *sum slices*, $\sum x[e_1 : e_2]$, which are sums over inclusive slices of arrays that enable binder-free reasoning over sums. In this section, expressions e are over algebraic symbols: a denotes either an array variable x or a disjunction of mutually exclusive boolean arrays $\vee_{\bot}\{x_1, \ldots, x_n\}$. $\vee_{\bot}\{x_1, \ldots, x_n\}[i]$ is equivalent to $x_1[i] \vee \cdots \vee x_n[i]$ since $x_1, \ldots, x_n$ are mutually exclusive.

[5] InferPart assumes that the $p_{1\ldots n}$ guards of x_{idx} are in ascending partial order. Filter-partitioning is derived similarly.
[6] Flattened domains are handled by matching the outermost iterator in the flat dimension.

Set of variables	X		
Array	a	$::=$	$x \mid \vee_{\perp} X$
Symbol	s	$::=$	$x \mid a[e] \mid \sum a[e : e]$
Term	t	$::=$	$n \mid s \mid s \cdot t$
Expression	e	$::=$	$t \mid t + e$

Legal range example

$$n \leq 3 \cdot i \leq \{5 \cdot n, n^2\}$$
$$\{i+1, 5\} \leq \sum X[i : i+j] \leq j - 1$$

Illegal range example

$$i \leq n \leq i \cdot i$$
$$n \leq 2 \cdot i \leq 2 \cdot n - 5$$

Algorithm 1 Solve($\Delta, e \leq 0$)

1 **let** e' = Apply PeelOnRange to Simplify(Δ, e)
2 **if** e' is a constant $n \in \mathbb{Z}$ and $n \leq 0$ **then return** Yes
3 **let** s = PickSym(Δ, e')
4 factorize e' by s yielding the form $s \cdot e_1 + e_2$
5 **let** (LowerB, n, UpperB) = $\Delta_{\text{Range}}(s)$ (*that is*, LowerB $\leq n \cdot s \leq$ UpperB)
6 **for each** $(l, u) \in$ LowerB $\times$ UpperB **do**
7   **if** Solve($\Delta, -e_1 \leq 0$) **then if** Solve($\Delta, u \cdot e_1 + n \cdot e_2 \leq 0$) **then return** Yes
8   **if** Solve($\Delta, e_1 \leq 0$) **then if** Solve($\Delta, l \cdot e_1 + n \cdot e_2 \leq 0$) **then return** Yes
9 **return** Unknown

Fig. 13. Algebraic language syntax and solver algorithm.

The algebraic context Δ consists of four maps (1) Δ_{Range} from symbols to ranges; (2) Δ_{Equiv} from symbols to equivalent expressions; (3) $\Delta_{\perp}$ from variables to their disjoint predicates (used to propagate the MECE property of the guards into the environment); and, finally, (4) Δ_{Untrans} is a bidirectional map between unlowerable expressions and fresh variables that represent them.

Δ_{Range} and Δ_{Equiv} are constructed to be cycle-free: when adding a range or equivalence on a symbol, the symbol must not be *used by* symbols appearing in the transitive closure of its ranges and equivalences. A symbol s is used by another symbol s' if the *leading variable* of s (Appendix B.5.2) is in the free variables of s'. For example, if Δ_{Equiv} is empty then the binding $i \mapsto x[i]$ is rejected because $i \in \{x, i\}$, but $x[i] \mapsto i$ is accepted because the leading variable $x \notin \text{fv}(i)$. Ranges are similarly rewritten into several candidates. If multiple legal candidates exist, one is selected by a set of heuristics, e.g., the one appearing latest in program order, or by giving preference to starting a new range over refining an existent one with a bound that depends on a symbol of unknown range.

4.5.1 Solver Algorithm. Inequalities are reduced to the form $e \leq 0$ and solved by the adaptation of Fourier-Motzkin elimination [23, 75] presented in Algorithm 1. It starts by simplifying e as presented in Section 4.5.2; this is essential in enabling Fourier-Motzkin elimination—which is usually only over variables—for our symbols, which include array slices and indexing. PeelOnRange peels off the last term in a sum slice if that term has a more specialized range than the one of the whole array. Line 2 is the base case, where e' is a constant. If not, PickSym selects the next symbol s to eliminate by choosing the symbol in e' whose range transitively depends on the most distinct symbols. Ranges are further refined based on existent knowledge, e.g., if array x is strictly positive, then a lower bound of $\sum x[l : u]$ is $u - l + 1$, if the latter is provably positive; similarly, an upper bound of the boolean-array disjunction $\sum \vee_{\perp} X[l : u]$ is also $u - l + 1$. Finally, lines 3–8 attempt to prove sufficient conditions satisfying the target inequality by suitably replacing s with its bounds.

4.5.2 Simplification Strategy. Our simplification strategy iterates three steps to a fixpoint; within each step, matching rules are applied deterministically across the expression's terms until a local fixpoint is reached. The rules and the algorithm are given in Appendix B.5.3. We outline each step here.

Step 1 substitutes symbols bound in Δ_{Equiv} with their equivalences and replaces $\vee_{\perp} X[e]$ with 1 whenever $\exists x \in X$ such that $\Delta_{\text{Equiv}}(x[e]) = 1$. It also replaces sums over provably empty slices with 0, and extends sum slices to include adjacent elements that have equivalences in Δ_{Equiv}:

$$\textsc{EmptySum}\ \frac{\text{Solve}(\Delta, e_1 > e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow 0} \quad \textsc{ExtendSum}\ \frac{\Delta \vdash a[e_1 - 1] \longrightarrow e_3 \qquad \text{Solve}(\Delta, e_1 - 1 \leq e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow \sum a[e_1 - 1 : e_2] - e_3}$$

Step 2 simplifies across sum-sum or sum-element terms (enabled by Step 1) by joining sums that are disjoint and eliminating and extracting overlaps between sums. For example, JoinSums4

1. **Query** $\Gamma \vdash \text{Query}_{idx}\,(idx(i_1) < idx(i_2))$ where $\Gamma = \Gamma', 0 \le i_1 < |xs|,\ 0 \le i_2 < i_1,\ p(xs(i_1)),\ \neg p(xs(i_2))$
$\Gamma_{\text{Ixfn}}(idx) = \lambda\,(i : 0..|xs|)\,.\,[\,p(xs(i))\,] * \sum_{j=0}^{i-1}(p(xs(j))) + [\,\neg p(xs(i))\,] * (i + \sum_{j=i+1}^{n-1}(p(xs(j))))$

2. **Build context**
$\Delta_{\text{Untrans}} = \{x_1 \leftrightarrow p(xs(\square)),\ x_2 \leftrightarrow \neg p(xs(\square), x_{|xs|} \leftrightarrow |xs|\}$
$\Delta_\perp = \{x_1 \mapsto \{x_2\},\ x_2 \mapsto \{x_1\}\}$
$\Delta_{\text{Equiv}} = \{x_1[i_1] \mapsto 1,\ x_1[i_2] \mapsto 0,\ x_2[i_1] \mapsto 0,\ x_2[i_2] \mapsto 1\}$
$$\Delta_{\text{Range}} = \left\{\begin{array}{c} 0 \le 1 \cdot x_1 \le 1 \\ 0 \le 1 \cdot x_2 \le 1 \\ 0 \le 1 \cdot x_{|xs|} \le \infty \\ 0 \le 1 \cdot i_1 \le x_{|xs|} - 1 \\ 0 \le 1 \cdot i_2 \le i_1 - 1 \end{array}\right\}$$

3. **Lower** $\text{Lower}_{\Gamma,idx}(idx(i_1) < idx(i_2))$
$= \text{Lower}_{\Gamma,idx}([\,p(xs(i_1)) \wedge p(xs(i_2))\,] * (\sum_{j=0}^{i_1-1}(p(xs(j))) < \sum_{j=0}^{i_2-1}(p(xs(j))))$
$+\ [\,p(xs(i_1)) \wedge \neg p(xs(i_2))\,] * (\sum_{j=0}^{i_1-1}(p(xs(j))) < i_2 + \sum_{j=i_2+1}^{n-1}(p(xs(j)))) + \ldots)$
$= (\Delta,\ \sum \vee_\perp\{x_1\}[0 : i_1 - 1] < i_2 + \sum \vee_\perp\{x_1\}[i_2 + 1 : x_{|xs|} - 1])$

4. **Solve** $\text{Solve}(\Delta,\ \sum \vee_\perp\{x_1\}[0 : i_1 - 1] < i_2 + \sum \vee_\perp\{x_1\}[i_2 + 1 : x_{|xs|} - 1])$

Fig. 14. Lowering the query discussed in Section 3.3.

simplifies two overlapping sum slices over mutually-exclusive boolean arrays, whose array variables do not overlap, but are mutually disjoint (in the same $\Delta_\perp$ class).

$$\textsc{JoinSums4}\ \frac{X \cap Y = \emptyset \qquad \bigcup_{x \in X} \Delta_\perp(x) = \bigcup_{y \in Y} \Delta_\perp(y) \qquad \text{Solve}(\Delta, e_1 \le e_3 \le e_2 \le e_4)}{\begin{array}{c}\Delta \vdash t \cdot \sum(\vee_\perp X)[e_1 : e_2] + t \cdot \sum(\vee_\perp Y)[e_3 : e_4] \\ \longrightarrow t \cdot \sum(\vee_\perp X)[e_1 : e_3 - 1] + t \cdot \sum(\vee_\perp(X \cup Y))[e_3 : e_2] + t \cdot \sum(\vee_\perp Y)[e_2 + 1 : e_4]\end{array}}$$

Step 3 applies EmptySum and then peels off elements from sums that have known equivalences (or more specialized ranges). Its rules collapse a singleton slice to an index, replace symbols bound in Δ_{Equiv}, peel off an index from the start/end of a sum, and eliminate empty sets from $\vee_\perp$ disjunctions.

4.5.3 Lowering. Lowering a query to the algebra layer has two stages (Fig. 14). First, the algebraic context Δ is built from the environment Γ and the index function associated with the query. In Fig. 14, x_1 and x_2 are fresh names for unlowerable expressions (guards) that map to reduction contexts (Fig. 9). E.g., $x_1\langle 0\rangle$ is the unlowerable expression $p(xs(0))$, so x_1 and x_2 are parametric over i—which we simply treat as arrays (i.e., $x_1[0]$). Next, the query expression is lowered. Specifically, we treat each term $[\,p\,] * e$ as an implication for inequalities nested inside guarded expressions: either (1) p must be falsifiable under Γ (see FalsifyGuard in Appendix B.3.6), or, (2) assuming p by extending Δ to Δ' with any ranges and equivalences in p, $\text{Solve}(\Delta', e)$ must return Yes. The bottom of Fig. 14 shows the only term that reaches case (2) (i.e., the expression guarded by $[\,p(xs(i_1)) \wedge \neg p(xs(i_2))\,]$). All other terms are invalidated by (1) since $p(xs(i_1))$ and $\neg p(xs(i_2))$ are assumed true in Δ.

5 Evaluation

We implement P^2 in the Futhark compiler [24, 44] supporting nD regular arrays and 2D flat jagged arrays. We support additional source constructs: tuples, (un)zip, concat, analysis over histograms [32], and exponentiation with integer base. The implementation prints index functions for each let-binding and reports failing queries with corresponding index functions and source variables.

We demonstrate P^2's ability to statically verify properties of bulk-parallel operations in 14 challenging Futhark programs that exercise different parts of our system. We demonstrate that dynamic verification of bounds checking and scatter's safety may incur big runtime overheads for GPU execution; our approach verifies them statically. Case studies demonstrate proving filter and partition properties (Section 5.1.1), verifying that scatters are deterministic (Sections 3.2, 5.1.1 and 5.1.2), and obviating dynamic bounds checks in indexing and gathers (all subsections).

Table 1. (a) Evaluation summary of verified properties and (b) performance speedups from static checks.

(a) FP abbreviates FiltPart. SAFE: all indexing and scatters are verified. #S and #A count scatters and annotations. Check time measures P²'s total runtime (Apple M4 chip). Compile time includes check time.

PROGRAM	PROPERTIES & ANNOTATIONS	SAFE	#S	#A	CHECK TIME	% OF COMPILE TIME
max_match	Range, Equiv, Inj, FP	✓	6	14	1.6 s	65 %
MIS	Range, Mono	✓	3	26	0.8 s	69 %
BFS	Range, Mono, Inj, FP	✓	1	26	1.3 s	87 %
quickhull	Range, InvFP, FP,	✓	8	21	8.1 s	98 %
FFT	Range, Inj	✓	1	1	0.2 s	16 %
primes	Range, FP	✓	2	12	0.5 s	58 %
lavaMD	Range	✓	0	6	1.9 s	82 %
srad	Range	✓	0	12	0.9 s	68 %
kmeans_ker	Range	✓	0	3	0.1 s	13 %
partition	Equiv, FP	✓	1	1	0.2 s	31 %
partition3	Equiv, FP	✓	1	2	0.8 s	64 %
seg_partition	Range, Equiv, FP	✓	1	3	1.1 s	76 %
filter	Equiv, FP	✓	1	3	0.1 s	21 %
filter_irreg	Range, Equiv, InvFP	✓	1	3	0.8 s	63 %

(b) NVIDIA A100 performance with dynamic checks (DYN.) as baseline. STATIC shows speedup over dynamic checks. +OPT additionally removes scattered array initialization.

PROGRAM & DATA	DYN. (*ms*)	SPEEDUP STATIC	SPEEDUP +OPT
kmeans_ker			
movielens	280	2.2×	
nytimes	315	1.9×	
scrna	861	2.2×	
partition			
50M	12	4.4×	1.08×
100M	38	7.0×	1.08×
200M	135	12.2×	1.05×

5.1 Experimental Evaluation

We evaluate P²'s ability to statically verify memory safety on real-world algorithms from PBBS [2] (the maximal matching, maximal independent set and breadth-first search graph algorithms and flattened nested parallel *quickhull*) and Rodinia [13] (*srad* and *lavaMD*), alongside the Cooley-Tukey FFT algorithm [17] (*FFT*), sparse k-means [59] (*kmeans_ker*), and an optimal work-depth implementation of a flattened nested parallel prime sieve [7] (*primes*).

We modify existing Futhark implementations to structure-of-arrays and A-normal form and extract loop bodies into function definitions to match our source language. In *quickhull*, we also introduce an explicit if-statement to prove a range property; this modification could be be obviated by propagating properties on slices of irregular arrays. Required annotations primarily specify: (1) *index ranges* for safe indirect indexing, and (2) *injectivity* for scatter safety. *MIS* expands and contracts flat jagged arrays. Safe indirect indexing requires a non-trivial flat-segment-dependent property: For $(i : 0..|y|)$ (Range y $0..\mathit{offsets}(i+1) - \mathit{offsets}(i) + 1$). Remaining annotations repeat 6 simple annotations that are derived from the benchmark's description of the input data. *max_match* iteratively removes graph edges using scatters. Safe scattering requires maintaining an Inj property on the initial edge index array as a loop invariant, repeating 3 unique annotations.

We also verify functional correctness for widely-used kernels—filter, alongside two-way, three-way, and segmented partitioning (*seg_partition*)—which appear throughout our benchmarks and are core to, e.g., radixsort and quicksort. To our knowledge, verification of memory safety for the PBBS benchmarks and prime sieve, and functional correctness for the widely-used data-parallel kernels (except for filter in [57]) has not been reported before. Table 1a summarizes the evaluation. Check times range from under 1 second to 8 seconds, increasing with complex index functions requiring many normalization rewrites (e.g., *seg_partition* and *quickhull*). Table 1b compares dynamic versus static verification on an NVIDIA A100 GPU. Futhark inserts dynamic bounds checks in CUDA kernels [31]. For *kmeans_ker*, static verification achieves roughly 2× speedup on datasets movielens [30], nytimes, and scrna [47] (using parameters from [59]) by eliminating dynamic checks. We implement dynamic scatter-determinism checks using reduce-by-index [32] to match the program's asymptotic work and compare against static verification. Static verification achieves 4–12× speedup

on random float arrays of 50–200 million elements. Dynamic checking uses atomic updates that thrash the L2 cache, exacerbating the overhead. The static version is further optimized (+OPT) by replacing scatter's destination array with an uninitialized array—safe because P^2 automatically proves all locations are overwritten—yielding a 5–8 % speedup.

We select two of the above benchmarks to illustrate P^2's capabilities: solving queries over complex nested expressions in Section 5.1.1, and, in Section 5.1.2, reasoning over concatenated forms.

5.1.1 seg_partition. This is a segmented version of *partition* that partitions each row of a flattened irregular array (Section 3.4). The postcondition asserts that p partitions each row:

$$\textbf{def}\ seg_partition\ (s : []\text{i64} \mid \text{Range}\ s\ 0..\infty)\ (x : []\text{f64} \mid \text{Equiv}\ |x|\ (\text{Sum}\ s))\ (p : []\text{bool} \mid \text{Equiv}\ |x|\ |p|)$$
$$: []\text{f64} \mid \lambda y.\ \text{For}\ (k : 0..|s|)\ (\text{Part}\ y\ x\ (\lambda i.\ p[i]))$$

The implementation lifts *partition* using segmented operations like *seg_ids* (Section 3.4). The inferred index functions have a flattened domain propagated from *flags* (Fig. 5a):

$$idx = \lambda\ (i_1 : 0..|s| \times i_2 : 0..s(i_1))\ .\ \begin{cases} [p(\sum_{j_1=0}^{i_1-1}(s(j_1)) + i_2)] * \sum_{j_1=0}^{i_1-1}(s(j_1)) + \sum_{j_1=\sum_{j_2=0}^{i_1-1}(s(j_2))}^{\sum_{j_1=0}^{i_1-1}(s(j_1))+i_2-1}(p(j_1)) \\ [\neg p(\sum_{j_1=0}^{i_1-1}(s(j_1)) + i_2)] * \sum_{j_1=0}^{i_1-1}(s(j_1)) + i_2 + \sum_{j_1=\sum_{j_2=0}^{i_1-1}(s(j_2))+i_2+1}^{\sum_{j_1=0}^{i_1}(s(j_1))-1}(p(j_1)) \end{cases}$$

$$seg_partition = \lambda\ (i_1 : 0..|s| \times i_2 : 0..s(i_1))\ .\ x(idx^{-1}(\textstyle\sum_{j=0}^{i_1-1}(s(j)) + i_2))$$

Verifying the postcondition requires proving partition properties like bijectivity over each row's restricted domain, e.g., Bij $idx\ \sum_{j=0}^{i_1-1}(s(j))..\sum_{j=0}^{i_1}(s(j))$. The solver applies the three-step method from Section 4.5.2. The sum-slice simplifications are critical here because boundaries are themselves sums. See Appendix D.2 for the implementation and an illustrative lowered query with solving steps.

5.1.2 FFT. The Cooley-Tukey FFT algorithm produces scatter indices *idx* comprising strictly monotonic segments via concatenation (e.g., $[0, 1, 4, 5, \dots]$ ++ $[2, 3, 6, 7, \dots]$):

```
def fft (n : i64 | Range n 1..∞) (...) (x : []f32 | Equiv |x| 2^n) =
  loop x for q < n do
    let iss1 = map (λk. map (λj. k * 2^(q+1) + j) 0..2^q) 0..2^(n-q-1)
    let iss2 = map (λk. map (λj. k * 2^(q+1) + j + 2^q) 0..2^q) 0..2^(n-q-1)
    let idx = (flatten iss1) ++ (flatten iss2)
    let vs = ...  in scatter x idx vs
```

$$idx = \lambda\ (i_1 : 0..2^{n-q} \times i_2 : 0..2^q)\ .\ \begin{cases} [i_1 < 2^{n-q-1}] * i_1 \cdot 2^{q+1} + i_2 \\ [i_1 \geq 2^{n-q-1}] * i_1 \cdot 2^{q+1} + i_2 + 2^q - 2^{n+q} \end{cases}$$

To verify scatter safety, we must prove that *idx* is injective. Concatenation yields the term -2^{n+q} in the second guarded expression above, which makes disjointness of the guards unprovable for our solver, even when exploiting the inferred 2D structure in the flattened domain. However, we also record the symbolic form of *idx* for concatenations, which can be used to verify injectivity:

$$idx = \lambda\ (i_1 : 0..2^{n-q-1} \times i_2 : 0..2^q)\ .\ i_1 \cdot 2^{q+1} + i_2\ \texttt{++}\ \lambda\ (i_1 : 0..2^{n-q-1} \times i_2 : 0..2^q)\ .\ i_1 \cdot 2^{q+1} + i_2 + 2^q$$

Using INJCONCAT (Fig. 12), P^2 proves that each constituent index function is injective and their images are disjoint. For $(i_1, i_2),\ (j_1, j_2) \in 0..2^{n-q-1} \times 0..2^q$, the solver verifies that $i_1 \cdot 2^{q+1} + i_2 < j_1 \cdot 2^{q+1} + j_2 + 2^q$ when $(i_1, i_2) \leq (j_1, j_2)$ lexicographically, and $i_1 \cdot 2^{q+1} + i_2 > j_1 \cdot 2^{q+1} + j_2 + 2^q$ otherwise.

6 Related Work

Liquid Types and Liquid Haskell. Liquid types are refinement types [54] automatically discharged by SMT solvers [71]. Refinement reflection [72] allows source functions in refinements and automates definition unfolding, but still requires manual proofs for our class of programs (like Dafny in Section 2.1). For instance, automatically verifying that the below code produces positive integers

seems to require fusion to maintain positional correlation between cs and is—contrary to the data-parallel programming style, but it can be proved by manually defining and applying a lemma.

```
let is = map (\c-> if c then 1 else 0) cs in zipWith (\c i-> if c then i-1 else 1) cs is
```

We could not verify *partition*'s data-parallel implementation, and verifying three-way and flat-parallel batch partitioning (*seg_partition* in Section 5.1.1) are harder still. In contrast, P^2 uses index functions to automatically infer positional correlations and prove array properties without structural constraints or proof writing.

F and Pulse.* F* [65] is a proof assistant with dependent and refinement types, combining SMT automation with interactive proving. It automates term reasoning via reductions but still requires manual proofs for properties our system handles automatically. PulseCore [21] verifies task-parallel quicksort via sequential partitioning using Pulse [66], which uses concurrent separation logic [12]. Unlike P^2, it cannot verify implementations that use data-parallel segmented partitioning.

Linear Array Logics. Dependent ML [76] and ATS [77] restrict dependent values to limited languages for decidability. Dependent ML enables static array bounds checking via linear constraints [78]. ATS allows explicit proof terms but requires intertwining proof and program. Daca et al.'s [18] logic for counting and partitioning, Bradley et al.'s [11] for index ranges and sortedness using Presburger arithmetic, and Qube [68] for array indexing and shape matching are restricted to linear indexing. An Agda DSL [74] for reverse-mode differentiation of affine functions focuses on verifying tensor indexing. We target programs with non-linear indexing via gather/scatter/scan.

Verifying Parallel Programs and Compiler Transformations. The VerCors [8] toolset has been used to verify CUDA implementations of prefix sum and *filter* operations [57]. The latter requires establishing four intermediate properties expressed via a non-trivial pure function, whereas P^2 requires a straightforward postcondition. Work on verifying scheduling DSLs [27] like Halide [52] includes improvements to its term rewriting system [46], validation of affine specifications [14], and HaliVer [70], which builds on VerCors to verify the low-level generated code via permission-based separation logic [9]. Bounded translation validation tools like Alive2 [39, 40] verify LLVM transformations.

Descend [35] targets GPU memory safety and data-race freedom by requiring programmers to decompose arrays (using structured layouts) and to assign entire dimensions to threads for partitioned memory access. This permits verification of regular access patterns, but not graph algorithms. TensorRight [3] verifies shape-polymorphic rewrites from XLA's algebraic simplifier [15], but similarly assumes structured layouts (e.g., reshapes are not supported). ATL [38] verifies affine schedules that involve quasi-affine indexing (except ceiling division). Work in [22] extracts pure bulk operations (e.g., map) from low-level code by modelling variables/allocations as loop-indexed sequences. P^2 complements these by supporting irregular index arrays in pure data-parallel settings.

Dependence Analysis. P^2 is inspired by loop optimizations where suitable access-pattern representations [43, 55] are key to scaling interprocedural analysis, statically [28, 50, 73] or via static-dynamic combinations [43, 48] for non-affine code. Inspector-executor [58] instances are used to prove parallelism [43, 49], extract parallel wavefronts [53, 79] and improve locality [20, 63], often by establishing index-array properties (e.g., injectivity, bijectivity, monotonicity). Notably, optimizing data layout (movement) is key to efficient GPU execution [25, 45, 67]. SPF [42, 62, 64] provides a principled way to integrate inspector executors into polyhedral analysis by modeling index-array results as uninterpreted functions annotated with *properties*. These are crucial to ensure analysis soundness and optimize inspector code [42]. We support most SPF properties [41, 62]—including periodic ones and permutation inverses—except for co-monotonicity and triangularity, which are feasible extensions. Statically verifying these index-array properties would simplify analyses, verify inspector code, and reduce runtime overheads.

Acknowledgments

We are very grateful to our reviewers for their extensive and helpful feedback, to Troels Henriksen and Lubin Bailly for extending the Futhark compiler to support pre- and postcondition parsing, to Martin Elsman for his advice on how to present expression contexts, and to Ranjit Jhala and Alessio Ferrarini for their expert insights into verifying data-parallel style programs in Liquid Haskell. Generative AI was used to help shorten paragraphs in this paper. Research reported in this publication was partially supported by the Novo Nordisk Foundation through award NNF24OC0090447.

Data-Availability Statement

An artifact of the P^2 prototype is available on Zenodo [34].

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] 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. doi:10.1145/3503221.3508422

[3] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. 2025. TensorRight: Automated Verification of Tensor Graph Rewrites. *Proc. ACM Program. Lang.* 9, POPL, Article 29 (Jan. 2025), 32 pages. doi:10.1145/3704865

[4] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. doi:10.1109/PACT.2015.17

[5] 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. doi:10.1145/3295500.3356173

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

[7] Guy E Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[8] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods*, Nadia Polikarpova and Steve Schneider (Eds.). Springer International Publishing, Cham, 102–110.

[9] 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. doi:10.1145/1040305.1040327

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. 2018. *JAX: composable transformations of Python+NumPy programs*.

[11] 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.

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

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

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

[15] OpenXLA Contributors. 2024. XLA-HLO Operation Semantics. https://openxla.org/xla/operation_semantics.

[16] Byron Cook. 2018. Formal reasoning about the security of amazon web services. In *International Conference on Computer Aided Verification*. Springer, 38–47.

[17] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301. http://www.jstor.org/stable/2003354

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

[19] S. de Gouw, F.S. de Boer, and R. et al. Bubel. 2019. Verifying OpenJDK's Sort Method for Generic Collections. *J Autom Reasoning* 62 (2019). https://doi.org/10.1007/s10817-017-9426-4

[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. doi:10.1145/301618.301670

[21] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI, Article 208 (June 2025), 24 pages. doi:10.1145/3729311

[22] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. 2019. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 157 (Oct. 2019), 30 pages. doi:10.1145/3360583

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

[24] Fabian Gieseke, Sabina Rosca, Troels Henriksen, Jan Verbesselt, and Cosmin E. Oancea. 2020. Massively-parallel change detection for satellite time series data with missing values. In *Proceedings - 2020 IEEE 36th International Conference on Data Engineering, ICDE 2020*. IEEE, 385–396. doi:10.1109/ICDE48307.2020.00040 36th IEEE International Conference on Data Engineering, ICDE 2020 ; Conference date: 20-04-2020 Through 24-04-2020.

[25] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3410463.3414632

[26] 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. doi:10.1145/3168824

[27] 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). doi:10.1145/3743135

[28] 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.

[29] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. *MLX: Efficient and flexible machine learning on Apple silicon*. https://github.com/ml-explore

[30] 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. doi:10.1145/2827872

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

[32] 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.

[33] 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.

[34] Nikolaj Hey Hinnerskov, Robert Schenck, and Cosmin Oancea. 2026. *Artifact for the paper "Verifying Array Properties in Pure Data-Parallel Programs"*. doi:10.5281/zenodo.19610644

[35] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 181 (June 2024), 24 pages. doi:10.1145/3656411

[36] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[37] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Verified Scheduling Artifact* 6, POPL (Jan. 2022), 55:1–55:28. doi:10.1145/3498717

[38] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (Jan. 2022), 28 pages. doi:10.1145/3498717

[39] 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. doi:10.1145/3453483.3454030

[40] 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. doi:10.1145/3166064

[41] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Extending index-array properties for data dependence analysis. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 78–93.

[42] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 594–609. doi:10.1145/3314221.3314646

[43] 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. doi:10.1145/301104.301112

[44] 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.

[45] 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. doi:10.1109/SC41404.2022.00036

[46] 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. doi:10.1145/3428234

[47] 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.

[48] Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical Inference Techniques for Loop Parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 509–520. doi:10.1145/2254064.2254124

[49] 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.

[50] 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

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[52] 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. doi:10.1145/2491956.2462176

[53] Lawrence Rauchwerger. 1998. Run-time parallelization: Its time has come. *Parallel Comput.* 24, 3 (1998), 527–556. doi:10.1016/S0167-8191(98)00024-6

[54] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. 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. doi:10.1145/1375581.1375602

[55] 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. doi:10.1145/514191.514229

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

[57] Mohsen Safari and Marieke Huisman. 2022. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoretical Computer Science* 912 (2022), 81–98. doi:10.1016/j.tcs.2022.02.027

[58] J.H. Saltz, R. Mirchandaney, and K. Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.* 40, 5 (1991), 603–612. doi:10.1109/12.88484

[59] 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. doi:10.1109/SC41404.2022.00063

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

[61] Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. *SIGPLAN Not.* 44, 6 (June 2009), 223–234. doi:10.1145/1543135.1542501

[62] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[63] 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. doi:10.1145/1065895.1065899

[64] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57. doi:10.1016/j.parco.2016.02.004

[65] 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.

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

[67] Amir Mohammad Tavakkoli, Cosmin E. Oancea, and Mary Hall. 2026. LEGO: A Layout Expression Language for Code Generation of Hierarchical Mapping . In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 228–241. doi:10.1109/CGO68049.2026.11394846

[68] 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. doi:10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[69] 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.

[70] 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.

[71] 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. doi:10.1145/2692915.2628161

[72] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. 2 (2018), 1–31. Issue POPL. doi:10.1145/3158141

[73] 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. doi:10.1145/2400682.2400713

[74] Artjoms Šinkarovs and Troels Henriksen. 2025. Correctness Meets Performance: From Agda to Futhark. *Proc. ACM Program. Lang.* 9, ICFP, Article 255 (Aug. 2025), 30 pages. doi:10.1145/3747524

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

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

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

[78] 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. doi:10.1145/277650.277732

[79] Xiaotong Zhuang, Alexandre E. Eichenberger, Yangchun Luo, Kevin O'Brien, and Kathryn O'Brien. 2009. Exploiting Parallelism with Dependence-Aware Scheduling. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 193–202. doi:10.1109/PACT.2009.10

Received 2025-11-12; accepted 2026-04-03