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

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 (fissioned) 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*, democratizing verification of data-parallel programs for users without verification expertise. We support a small but powerful predefined set of properties—ranges, 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 → Functional languages**; **Parallel programming languages**.

Additional Key Words and Phrases: Pure functional language, data parallel, array programming, verification

1 Introduction

High-performance array languages (e.g., Futhark [23], Accelerate [44], Lift [17], DaCe [3], JAX [7]) and machine learning frameworks (e.g., TensorFlow [1], MLX [19], PyTorch [33]) 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 [24], F* [41], and Liquid Haskell [35, 47] 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 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). Moreover, small implementation changes may require fundamentally different proof strategies (if a proof is possible at all with the changes), 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

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

to annotate functions with pre- and postconditions using a small but powerful set of predefined properties: ranges, injectivity, bijectivity, monotonicity, and filtering/partitioning. P^2 is not a general-purpose theorem prover; supporting arbitrary properties and user proofs is an explicit non-goal. Instead, it leverages the fixed semantics of bulk-parallel operators to enable high automation verification for common array properties.

The key idea is to infer *index functions* from integer arrays in the source program—piecewise functions from indices to the elements at those indices, where elements are represented 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 data arrays 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 [4]).

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*, which translates each function into an index function (2) the *property layer*, which tracks and proves properties over the index functions, and (3) the *algebra layer*, which reasons about and dispatches low-level algebraic queries (generated by the property layer when proving properties) using a Fourier-Motzkin elimination-based solver. The three components are deeply interconnected to promote high automation. For example, the index function layer exploits bijectivity and monotonicity properties (established by the property layer) to produce meaningful index functions for scatter that enable the derivation of filtering/partitioning properties and flat expression of jagged arrays.

We make the following contributions:

- **Index functions:** A translation from bulk-parallel programs to guarded index functions enabling equational reasoning about data-parallel array programs and inference of segmented structure from flattened irregular nested parallel operations (Sections 3.1, 3.4 and 4.2).
- **Property reasoning:** A property system that automatically proves array properties, including injectivity and bijectivity for scatters, enabling reasoning about permutations, partitions and filters of data arrays (Sections 3.2, 4.1 and 4.3).
- **Algebraic solver:** An extension of Fourier–Motzkin elimination with tactics for overlapping sums, indexing, and mutually exclusive guards to automatically discharge (in)equalities over index-function guards (Sections 3.3 and 4.4).
- **Implementation:** An implementation of P^2 in the Futhark compiler, evaluated on a set of diverse data-parallel benchmarks that exercise the entire system, with P^2 automatically verifying all indexing, scatters, and annotated properties. The evaluation demonstrates utility for optimizing compilers by eliminating redundant bounds checks and array initializations (Section 5). The source code is available in the artifact and it will be open-sourced.

2 Source language and motivation

P^2 analyzes array programs expressed in a purely functional language using second-order array combinators (SOACs) [23], which include 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 and frameworks such as vectorized operations (using map), reductions and cumulative sums (using scan), scatters, and gathers (using map: map $(\lambda i.\, xs[i])\ idx$). The types

and semantics of map and scan are:

$$\mathsf{map} \;:\; (\alpha \to \beta) \to []\alpha \to []\beta \qquad \mathsf{scan} \;:\; (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to []\alpha$$
$$\mathsf{map}\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n] \qquad \mathsf{scan}\ f_\odot\ e_\odot\ [x_1, \ldots, x_n] = [x_1, \ldots, f_\odot\ (\ldots (f_\odot\ x_1\ x_2) \ldots)\ x_n]$$

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

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

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

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. Scatter writes these locations all at once—if different in-bounds values are written to the same location, the result is unspecified. This leads to the following precondition: for any two duplicate indices in idx, the values in x must be the same ($\forall j, k \in [0, |x|)\ .\ idx[j] = idx[k] \implies x[j] = x[k]$). This precondition is not checked statically in existing array languages and frameworks. In practice, it is typically satisfied because indices in idx are unique or because all values in x are equal (e.g., an array of zeros). Scatter also requires that $|idx| = |x|$. Its work is proportional to the number of updates, $|x|$.

Source language. The source language permits defining top-level functions with the syntax

$$\begin{array}{l}\textbf{def}\ \mathsf{f}\ (x_1 : \tau_1 \mid e^*_{\text{pre1}})\ (x_2 : \tau_2 \mid e^*_{\text{pre2}})\ \ldots\ (x_3 : \tau_3 \mid e^*_{\text{pre3}}) : \tau \mid \lambda y.\ e^*_{\text{post}} = \\ \quad \textbf{let}\ z_1 = e^*_1\ \textbf{in}\ \textbf{let}\ z_2 = e^*_2\ \textbf{in}\quad \ldots \quad \textbf{let}\ res = e^*_{res}\ \textbf{in}\ res\end{array}$$

where e^* is an expression that includes function and operator application, conditionals, let-bindings, and SOACs (see Appendix A for a grammar). The function body is in A-normal form [37]: it essentially consists 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 $(x : \tau_1 \mid e^*_{\text{pre1}})$ says that x has type τ_1 and is subject to precondition e^*_{pre1}, which is assumed when analyzing the function's body and is checked at the call site. The return type $\tau \mid \lambda y.\ e^*_{\text{post}}$ says that the function return has type τ and satisfies the postcondition e^*_{post}, which is proved by P^2. The pre- and postconditions are source-level expressions which notably include the properties in Fig. 8 (encoded as source-level Boolean functions). A special shape function $\mathrm{shp}(\cdot)$ returns array dimensions, with $|x| = \mathrm{shp}(x)_1$ denoting the size of x's first dimension; scalars are treated as unit-length arrays. The source language does not permit irregular (jagged) arrays or anonymous functions (except in SOACs). Instead, irregular arrays may be represented as a pairing of a data and shape array (see Section 3.4).

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. 1 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 (lines 8, 9). Finally, scatter reorders the data array all at once. Fig. 1 (right) shows a trace of the program.

Analyzing index arrays informs properties about programs as a whole: array primitives like scatters and gathers propagate properties on index arrays to arrays of any type (e.g., data arrays).

```
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 trace

```
>>> 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. 1. A source program implementing partition using SOACs.

```
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); }
```

```
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]);
} }
```

Fig. 2. partition_inds in Dafny: the left column shows the code; the right column shows one of the lemmas.

For example, *idx* is actually a permutation of the indices of *xs* ($0 \ldots |xs| - 1$). 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, our analysis is further able to prove that the permutation forms a partition according to p, as denoted by the postcondition $\lambda ys.\ \text{Part}\ ys\ xs\ (\lambda i.\ p\ xs[i])$.

2.1 Challenges of verifying partition in Dafny

This section is a case study on verifying partition using Dafny [24]—an industrial-strength verification framework (e.g., used at AWS [11]) with good support for reasoning about data-dependent array accesses. We built a library of SOACs in Dafny. For example, 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. The left column in Fig. 2 shows Dafny code for partition_inds, which constitutes the primary verification burden for partition. Dafny is able to verify the specified properties on `idx` (lines 3-8), which include the length of `idx` equals the length of `xs` (line 3), the indices of the elements that succeed/fail under the predicate form strictly monotonic sequences (line 5/6) and any succeeding index is smaller than any failed index (lines 7-8). Verifying the postcondition requires applying two user-defined lemmas (lines 19-21). To illustrate, the user-guided inductive proof of the `ComplementarySums` lemma is shown in the right column of Fig. 2 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 strategy. Dafny's verification process is brittle on SOAC-based programs—small changes to the implementation may require an entirely new proof strategy or may even make the proof intractable. We illustrate with two simple examples.

The first example rewrites `partition_inds` such that the scattered indices of the elements that fail under the predicate are computed with a reverse prefix sum:

```
var left_rev := seq(|left|, i requires 0 <= i < |left| => left[n-1-i]);
var n_left_at_and_after := scan((x,y) => x + y, 0, left_rev);
var indsF := map2((i,t) => i + t + 1, seq(|xs|, i => i), n_left_at_and_after);
```

In this form, proving the necessary index properties requires proving a query of the form:

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

which is challenging to solve because the quantified variables j and i occur in the bounds of their corresponding summed slices. 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.

The second example refers to the properties of exclusive prefix sum: $\mathrm{sum}^{exc}\ [a_1, \ldots, a_n] = [0,\ a_1,\ a_1 + a_2,\ \ldots,\ a_1 + \ldots + a_{n-1}]$. It can be implemented by shifting the array elements right and then applying an inclusive scan:

```
var inp_rot := map(i => if 1 <= i < |inp| then inp[i-1] else 0, seq(|inp|, i => i));
var inp_exc_scan := scan((x,y) => x + y, 0, inp_rot);
```

Dafny proves that each element o `inp_exc_scan` is a partial sum over `inp_rot` (via the three queries $|\texttt{inp}| > 0 \ \Rightarrow \texttt{inp\_rot}[0] = 0$, $0 < i < |\texttt{inp}| \Rightarrow \texttt{inp\_rot}[i] = \texttt{inp}[i-1]$, and $0 \leq i < |\texttt{inp}| \ \Rightarrow \texttt{inp\_exc\_scan}[i] = \sum_{k=0}^{i} \texttt{inp\_rot}[k]$) but it cannot prove that they are partial sums over the `inp` array: $0 \leq i < |\texttt{inp}| \Rightarrow \texttt{inp\_exc\_scan}[i] = \sum_{k=0}^{i-1} \texttt{inp}[k]$.

2.1.3 Dafny cannot reason about scatter. The left column of Fig. 3 shows the pre- and postconditions of our `scatter` implementation in Dafny, ensuring the semantics given in Eq. (1). The bottom side of the left column shows partition as presented in Fig. 1; 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 cannot 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 (shown on the right of Figure 3) such that the scatter (at line 6) produces the inverse permutation of `idx`, denoted $\sigma = idx^{-1}$, by scattering at indices in `idx` the value array $[0, \ldots, |xs|-1]$ (denoted *iota* at line 5). The result *ys* is obtained by gathering the elements of *xs* using the indices of σ (line 7). In this version, Dafny is able to verify the postconditions if σ is assumed to be an inverse of *idx* (line 11). Dafny is unable to verify this assumption, even when guided by intermediate

```
1  method scatter<T>(ys: seq<T>, idx: seq<int>, vs: seq<T>)
2                    returns (zs: seq<T>)
3    requires (|idx| == |vs|) && (injective(idx) || replicated(vs))
4  ✓ ensures (|ys| == |zs|) &&
5      (forall k :: 0≤k<|idx| && 0 ≤ idx[k] < |zs| ⇒ zs[idx[k]] == vs[k])
6  ✓ ensures forall i :: 0 ≤ i < |zs| ⇒
7      ((zs[i] == ys[i]) || (exists k :: 0 ≤ k < |idx| && idx[k] == i && zs[i] == vs[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 }
```

```
method partition( p: int -> bool, xs: seq<int>)
                  returns (ys: seq<int>) {
  var split, idx := partition_inds(p, xs);
  var dest := seq(|xs|, i requires 0≤i<|xs| => 0);
  var iota := seq(|xs|, i requires 0≤i<|xs| => i);
  var σ := scatter (dest, idx, iota);
  ys := seq(|xs|, i requires 0≤i<|xs| => xs[σ[i]]);

  × assert
    (forall i :: 0 ≤ i < |xs| ⇒ idx[σ[i]] == i);
  assume
    (forall i :: 0 ≤ i < |xs| ⇒ idx[σ[i]] == i);
  // These fail without the assumption above.
  ✓ assert (forall i :: 0 ≤ i < split ⇒ p(ys[i]));
  ✓ assert (forall i :: split ≤ i < |xs| ⇒ !p(ys[i]));
}
```

Fig. 3. The left column shows the implementation of `scatter` and `partition` in Dafny. The right column zooms in to identify the core challenge to verification, namely the *inverse* property. Lines marked with × and ✓ fail and succeed, respectively. Succeeding lines that come after **`assume`** would fail without that assumption.

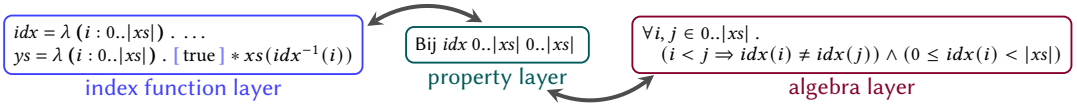
assertions establishing that $0 \leq i < |xs| \Rightarrow \sigma[idx[i]] == i$, which should allow Dafny to infer that σ's elements are unique. (Dafny is able to show that $0 \leq i < j < |xs| \Rightarrow \sigma[idx[i]] \neq \sigma[idx[j]]$, but can't show $0 \leq i < j < |xs| \Rightarrow \sigma[i] \neq \sigma[j]$.) 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 $\longrightarrow$ index functions $\times$ properties.

P^2 translates each function definition (**def**) to an index function and then verifies and infers properties over it. To scale the analysis, P^2 reuses inferred properties at call sites: formal parameters are substituted with actual arguments in the index function, and the postcondition is propagated into the caller's context.



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 work and interact to verify the partition program (see Figs. 1 and 4).

$\mathcal{P}$(**def** partition (p : f64 → bool) (xs : []f64) = ...)

$mask = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * p(xs(i))$

$left = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * p(xs(i))$

$right = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * (1 - p(xs(i)))$

$\vdots$

$n\_right = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * (i + 1 - \sum_{j=0}^{i}(p(xs(i))))$

$\vdots$

$idx = \lambda\,(i : 0..n)\,.\,\begin{cases}[\,p(xs(i))\,] * \sum_{j=0}^{i-1}(p(xs(j))) \\ [\,\neg p(xs(i))\,] * (i + \sum_{j=i+1}^{n-1}(p(xs(j))))\end{cases}$

$zeros = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * 0$

$ys = \lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * xs(idx^{-1}(i))$

Notation: $n = |xs|$.

$\mathcal{P}$(**let** n_right = scan ($\lambda x\ y.\ x + y$) 0 $right$)

1. $\lambda\,(i : 0..n)\,.\,\begin{cases}[\,i = 0\,] * right(i) \\ [\,i \neq 0\,] * (\circlearrowleft + right(i))\end{cases}$
2. $\lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * (right(0) + \sum_{j=1}^{i}(right(i)))$
3. $\lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * \sum_{j=0}^{i}(right(i))$
4. $\lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * \sum_{j=0}^{i}([\,\text{true}\,] * (1 - p(xs(i))))$
5. $\lambda\,(i : 0..n)\,.\,[\,\text{true} \wedge \text{true}\,] * \sum_{j=0}^{i}(1 - p(xs(i)))$
6. $\lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * (\sum_{j=0}^{i}(1) - \sum_{j=0}^{i}(p(xs(i))))$
7. $\lambda\,(i : 0..n)\,.\,[\,\text{true}\,] * (i + 1 - \sum_{j=0}^{i}(p(xs(i))))$

Fig. 4. P^2 transforms programs to index functions, enabling algebraic reasoning about properties.

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, $\lambda ys.\,\text{Part}\ ys\ xs\ (\lambda i.\ p\ xs[i])$, is treated after the body has been analyzed.

Translation. Function bodies are translated one let-binding at a time into index functions. Since bodies are A-normal, each let-binding applies a SOAC or other source construct to earlier bindings. For example, P^2 first translates the scan operation defining n_right into an index function (Fig. 4):

$$n\_right = \lambda\,(i : 0..|xs|)\,.\,[\,i = 0\,] * right(i) + [\,i \neq 0\,] * (\circlearrowleft + right(i))$$

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 $right(i)$ and $\circlearrowleft + right(i)$ that define the value at index i when their guard holds. The $\circlearrowleft$ symbol represents the recurrence introduced by scan; normalization transforms it into a prefix sum. 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 P^2 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. Rewrite rules normalize all expressions: n_right is normalized through the rewrite sequence shown in Fig. 4. Each step applies a rewrite rule (introduced in Sections 4.2 and 4.4): (1) Scan introduces a recurrence $\circlearrowleft$ based on scan's semantics; (2) RecSum (strength reduction) converts the recurrence to a closed-form sum; (3) JoinSums2 absorbs the base term into the sum; (4) Sub replaces $right(i)$ with its guarded expression; (5) Hoist hoists the nested guard $[\,\text{true}\,]$ to the outer guarded expression; (6) SplitSum splits the sum; and (7) SumConst eliminates a sum.

Together, substitution and hoisting of guarded expressions enables P^2 to automatically track positional dependencies backwards to the formal arguments of a function. Each index function shown in Fig. 4 has been produced by our system, and a user can inspect those functions to get an idea for what knowledge P^2 has inferred about the program. For example, *idx*'s index function says that if $p(xs(i))$ is true, index i maps to the count of preceding true elements; if false, index i maps to i plus the count of subsequent true elements.

Handling scatter. Scatter is the only source construct requiring multiple translation rules due to its determinism side condition in Eq. (1). For example, at line 12 in Fig. 1, the indices *idx* used to update the destination array *zeros* are a permutation of *zeros*' indices. Semantically, the result is $ys[idx[i]] = xs[i]$ for each index i. The rule below exploits this by verifying the equivalent premise that *idx* is a bijection over *zeros*' index function domain:

$$\textsc{Scatter2}\ \frac{\Gamma \vdash \text{Bij } idx\ 0..|xs|\ 0..|xs| \qquad \text{fresh } i}{\Gamma \vdash \text{scatter } zeros\ idx\ xs \longrightarrow \lambda\ (i : 0..|xs|)\ .\ [\,\text{true}\,] * xs(idx^{-1}(i))}$$

The bijection property Bij $idx\ 0..|xs|\ 0..|xs|$ says that $idx|_{idx^{-1}(0..|xs|)}$ bijectively maps to $0..|xs|$, where $idx|_{idx^{-1}(0..|xs|)}$ restricts the domain of *idx* to indices that map to $0..|xs|$ (i.e., ignoring elements outside this range in the underlying array). This restriction captures scatter's semantics, which ignores out-of-bounds values in the source array. To verify the bijection, P^2 queries the property layer (Section 3.2), which proves the property and reports back to the index function layer. The index function layer then exploits *idx*'s invertibility to produce *ys*' index function in Fig. 4. A more general scatter rule appears in Appendix B.1.1. When no rule applies to a scatter, we create an uninterpreted index function over the destination array's indices by indexing into a fresh name.

This final index function for *ys* is bound to partition in the environment—ready for verification against its postcondition—along with information about its parameters and preconditions.

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 (Sections 3.3 and 4.4), after which the bijection property is recorded into the environment.

After traversing the body and populating its environments with inferred facts, P^2 verifies the postcondition under the constructed context. For partition, verifying $\lambda ys.\ \text{Part}\ ys\ xs\ (\lambda i.\ p\ xs[i])$ proceeds in two steps. First, P^2 checks that *ys*'s index function has the 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

```
def filter (p : i64 → bool) (xs : []f64) : []f64 | λys. Filt ys xs (λi. p i)
  Analogous to partition in Fig. 1; uses scatter.
def get_smallest_edges (edges : []i64 | Range edges 0..|H|) (is : []i64 | Inj is (−∞)..∞ ∧ Equiv |edges| |is|)
                       (H : []i64) : []i64 | λedges′. Inj edges′ (−∞)..∞
  let edges′ = filter (λi. H[edges[i]] = is[i]) edges in edges′
```

Fig. 5. A simplified excerpt from a maximal matching algorithm.

establishing the corresponding inverse-partition property on *idx* (shown in Fig. 8) via the queries

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

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

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 P^2'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 P^2 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.

P^2 can be used to extend compilers with more sophisticated property-based reasoning, such as reasoning about partitions across conditionals (Appendix B.2.2) and verifying properties over uninterpreted expressions without direct propagation. To illustrate this latter point, consider the function get_smallest_edges in Fig. 5, which filters *edges* by indexing into an array H that stores the smallest index for each edge. Although the input *edges* may contain duplicates, we must verify that the filtered output $edges'$ has unique values (i.e., Inj $edges'$ $(-\infty)..\infty$). From filter's postcondition, $edges'$ is a filtered permutation of *edges* where elements satisfying $H(edges(i)) = is(i)$ are kept. P^2 applies the property rule FilterInj (Fig. 11) that generates a proof obligation that augments the injectivity proof obligation with the filter predicate. This yields the query

$$\forall i, j \in 0..|edges| \,.\, H(edges(i)) = is(i) \wedge H(edges(j)) = is(j) \wedge edges(i) = edges(j) \implies i = j$$

Enriching this query with transitive equalities derived from $edges(i) = edges(j)$, yields $H(edges(i)) = H(edges(j))$ and crucially $is(i) = is(j)$. Since *is* is injective (from the precondition), $is(i) = is(j)$ implies $i = j$, which proves the injectivity of $edges'$ through the injectivity of *is*. We implement this high-level reasoning example and use it to verify the maximal matching program in the Problem-Based Benchmark Suite [2] (Section 5). See Appendix C.3 for more details.

3.3 Algebra layer

P^2's algebra layer extends Fourier-Motzkin elimination [16, 49] with reasoning about sums, indexing and mutually exclusive boolean variables. Zooming in on the (partition) query above, P^2 substitutes applications of *idx* for its guarded expressions in Fig. 4, conjoining guards with the antecedent of

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

(b) Rewrites for a segmented sum using $f = y = \mathit{flags}$.

Fig. 6. P^2 infers and propagates flattened irregular structure.

the query. For example, for the third conjunct of the (partition) query, the system rewrites

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

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

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

This corresponds to Eq. (2)—the failing query in Section 2.1.2. P^2 checks the inequality on the RHS of the guard, to do so it 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:

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

Standard Fourier-Motzkin elimination fails to verify the inequality, yielding $i < j + 0$ when maximizing the LHS and minimizing the RHS under the ranges. Notice that the RHS term j is an upper bound for $\sum_{k=0}^{j-1}(p(xs(k)))$, which overlaps with the LHS sum $\sum_{k=0}^{i-1}(p(xs(k)))$. Hence, the RHS is an upper bound on those terms on the LHS and also includes $p(xs(i)) = 1$ since $j < i$. 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 $p(xs(j))$. P^2 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.

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

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 *flag* auxiliary array for the jagged array $[[x_1, x_2], [], [x_3, x_4, x_5], [x_6, x_7, x_8]]$ is $[1, 0, 3, 0, 0, 4, 0, 0]$ where each non-zero element denotes the start of a row (thereby encoding the shape). Flag arrays are used to lift bulk-parallel operations to irregular arrays. For

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

Fig. 7. Polynomial expression syntax parameterized by symbols s.

example, segmented scan [4]—which scans each jagged row—is implemented as a scan with lifted operator on the flag (f) and data arrays (y):

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

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. 6a computes a flag array. The inferred index function, shown below, represents irregular arrays through a 2D index domain where the inner dimension depend on the outer one:

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

The syntax $\times$ denotes that the two dimensions correspond to one flat dimension in the source program. This representation also captures empty rows: when $i_1 = 1$, then $0..s(i) = 0..0 = \emptyset$. 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: $\mathit{flags} = \text{map}\ (\lambda i.\ \text{map}\ (\lambda j.\ \textbf{if}\ j = 0\ \textbf{then}\ x(i)\ \textbf{else}\ 0)\ 0..s(i))\ 0..|s|$.

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, we infer this structure by matching 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 a flag array. The index function inferred for seg_sum in Eq. (4) is

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

from which—in the case of segment ids—P^2 derives the index function as $\lambda\ (i_1 : 0..|s| \times i_2 : 0..s(i_1))\ .\ [\,\text{true}\,] * (i_1 + 1)$ by substituting f and y for flag arrays. Figure 6b 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 C.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. 7). The index function and algebra layers both use this polynomial representation, but over different symbols s. Symbol and term order in expressions is 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 cases 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 and Z denote contiguous integer sets. For example, $Y = 0..n$ is the set of integers from 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 $Y \subseteq \text{dom}(x)$.

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 $x\ y$	$\lvert x\rvert = \lvert y\rvert \land \forall i \in 0..\lvert x\rvert\ .\ x(i) = y(i)$
Inj $x\ Y$	$\forall i, j \in 0..\lvert x\rvert\ .\ x(i) \in Y \land x(i) = x(j) \Rightarrow i = j$ $\lor\ \forall i, j \in 0..\lvert x\rvert\ .\ x(i) \in Y \land x(j) \in Y \land i \neq j \Rightarrow x(i) \neq x(j)$
Bij $x\ Y\ Z$	Inj $x\ Y\ \land\ Z \subseteq Y\ \land\ \lvert Z\rvert = \lvert\{x(i) \in Y \mid i \in 0..\lvert x\rvert\}\rvert$
InvFiltPart $x\ Z\ p_f\ (p_1, \ldots, p_n)$	Bij $x\ Z\ Z\ \land\ \lvert Z\rvert = \sum_{j \in 0..\lvert x\rvert} p_f(i)$ $\land\ \forall q, r \in \{1, \ldots, n\}\ .\ \forall i, j \in 0..\lvert x\rvert\ .$ $\big(q \neq r \Rightarrow \neg p_q(i) \lor \neg p_r(i)\big)$ $\land\ \big(i < j \land q \leq r \land p_q(i) \land p_r(j) \land p_f(i) \land p_f(j) \Rightarrow x(i) < x(j)\big)$ $\land\ \big(i < j \land q > r \land p_q(i) \land p_r(j) \land p_f(i) \land p_f(j) \Rightarrow x(i) > x(j)\big)$
FiltPart $y\ x\ p_f\ (p_1, \ldots, p_n)$	InvFiltPart $z\ (0..\sum_{j \in 0..\lvert x\rvert} p_f(i))\ p_f\ (p_1, \ldots, p_n)$ where $y \mapsto \lambda\,(i : 0..\sum_{j \in 0..\lvert x\rvert} p_f(i))\ .\ \text{true} \Rightarrow x(z^{-1}(i))$
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. 8. Array properties and the sufficient conditions to establish them.

$x|_{x^{-1}(Y)}$ 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_{Ixfn} for index functions, Γ_{Range} for ranges, and so on for each property in Fig. 8. Unbound variables map to $\emptyset$. 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. Range and Equiv properties are combined by a process described in Section 4.4. Environments are extended with predicates directly: $\Gamma, e_1 = e_2, 0 \leq i < j < n$ adds equivalences derived from $e_1 = e_2$ to Γ_{Equiv} and ranges from $0 \leq i < j < n$ to Γ_{Range}.

Query $(\Gamma,\ p)$ asks the question: Is p true under environment Γ? Our solver either returns Yes or Unknown (Section 4.4). When used in inference rule premises, the answer must be Yes.

$e_1[x/e_2]$ 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 8 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 $Y - Z$, enabling the user to specify, e.g., that all non-negative values ($Y = 0..\infty$) map bijectively to $Z = 0..n$ with Bij $x\ 0..\infty\ 0..n$.

Because filtering and partitioning compose commutatively, they're unified into one property. Predicates p are index functions from 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 $x\ Z\ p_f\ (p_1, \ldots, p_n)$

$$\begin{array}{rrcl} \text{Index fun.} & f & ::= & \lambda\,(D)\,.\,e \\ \text{Domain} & D & ::= & i : 0..e \end{array} \qquad \begin{array}{rrcl} \text{Symbol} & s & ::= & x \mid [p] * e \mid x(e) \mid x^{-1}(e) \mid \sum_{x=e}^{e}(e) \mid \circlearrowleft \mid p \\ \text{Predicate} & p & ::= & x \mid \text{true} \mid \text{false} \mid \neg p \mid p \wedge p \mid p \vee p \mid e \le e \mid x(e) \mid \circlearrowleft \end{array}$$

$$\begin{array}{rcl} C & ::= & \square \mid C + e \mid C \cdot t \mid [C] * e \mid [p] * C \mid x(C) \mid x^{-1}(C) \mid \sum_{x=C}^{e}(e) \mid \sum_{x=e}^{C}(e) \\ & \mid & \sum_{x=e}^{e}(C) \mid \neg C \mid C \wedge p \mid p \wedge C \mid C \vee p \mid p \vee C \mid C \le e \mid e \le C \end{array}$$

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

is the dual: it captures scatter/gather behavior where x is used as the index array. For example, scatter $z\ x\ y$ where $|z| = |Z|$ and map $(\lambda i.\, y[i])\ (x|_{x^{-1}(Z)})^{-1}$ are equivalent to filtering y by p_f and partitioning by $(p_1, \ldots, p_n)$. P^2 verifies: (1) p_f holds for exactly $|Z|$ indices in x, and (2) $x|_{x^{-1}(Z)}$ permutes Z such that each partition occupies a contiguous range: indices satisfying $p_f \wedge p_1$ map to $0, 1, \ldots, \sum_{i \in Z}(p_f(i) \ \wedge \ p_1(i)) - 1$; 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 in sequence. Filt $y\ x\ p$ and Part $y\ x\ p$ are aliases for filtering and 2-way partitioning.

4.2 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). Reduction contexts C define where a subexpression occurs in an expression (shown in Fig. 9). $C\langle e \rangle$ denotes the expression obtained by replacing $\square$ with e in the context C. For example, if $C = [p_1] * e_1 + [p_2] * x_1(\square)$, then $C\langle e_2 \rangle$ is $C = [p_1] * e_1 + [p_2] * x_1(e_2)$.

We may omit writing trivial guards and coefficients: $[\text{true}] * t = t$ and $[p] * 1 = [p]$. Guard multiplication yields logical conjunction: $[p_1] * 1 \cdot [p_2] * e_1 = [p_1 \wedge p_2] * e_1$. Hence each term in an expression has at most one guard. Together with expression multiplication, this combines guards:

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

such that the partition of the domain is maintained. Scalars are single element arrays: $\lambda\,()\,.\,g$ abbreviates $\lambda\,(i : 0..1)\,.\,g$. Comparisons are syntactic sugar: $0 \le e_1 < e_2$ is $(0 \le e_1) \wedge (e_1 + 1 \le e_2)$. Inference rules match to expressions and index functions via unification with bound variables [39].

4.2.1 Source language conversion. Figure 10 gives rules for converting the source language to index functions. The judgment $\Gamma \vdash e^* \longrightarrow f$ states that source expression e^* has index function f under environment Γ. Arrays are assumed to be one-dimensional (this restriction is lifted in Section 4.2.3). Idx converts array indexing with bounds checking. Var converts variables to index functions. 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 $\circlearrowleft$ (illustrated in the derivation of *n_left* in Fig. 4.) 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. Each rule implicitly returns a new environment; the proper judgment is $\Gamma \vdash e^* \longrightarrow (\Gamma, f)$ where the returned environment has been extended with any properties proven in the rule premises or inferred in Let using the rules in Section 4.3. The complete conversion rules are in Appendix B.1.1. Untranslatable expressions are treated as uninterpreted functions by introducing an index function that indexes a fresh name over some domain (possibly of unknown size).

$$\textsc{Let}\ \frac{\Gamma \vdash e_1^* \rightsquigarrow f_1 \quad \Gamma, x \mapsto f_1 \vdash e_2^* \rightsquigarrow f_2}{\Gamma \vdash \textbf{let}\ x = e_1^*\ \textbf{in}\ e_2^* \rightsquigarrow f_2} \qquad \textsc{Idx}\ \frac{\Gamma \vdash e^* \rightsquigarrow \lambda\,() \,.\, e \quad \text{Query}\ (\Gamma,\ 0 \le e < |x|)}{\Gamma \vdash x[e^*] \rightsquigarrow \lambda\,() \,.\, x(e)} \qquad \boxed{\Gamma \vdash e^* \rightsquigarrow f}$$

$$\textsc{Var}\ \frac{\text{fresh } i}{\Gamma \vdash x \rightsquigarrow \lambda\,(i : 0..|x|) \,.\, x(i)}$$

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

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

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

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

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

4.2.2 Normalization. Figure 10 shows normalizing rewrite rules that are applied to a fixed point after each source language conversion step. RecSum rewrites recurrences introduced by scan into closed-form sums, as seen in rewrite (2) of *n_left* 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 sorrounding 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 HoistGuards enable P^2 to track positional dependencies backwards to the formal arguments of a top-level definition. Appendix B.1.2 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.

4.2.3 Multi-dimensional arrays. The source language allows multi-dimensional arrays with reshaping via flatten : $[][]\alpha \to []\alpha$, 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 an index expression in terms of a flat domain D's index variables. Rules Flatten and PropFlatten (Appendix B.1.4) use these to syntactically preserve and propagate shape information about flattened arrays. Existing rules (Fig. 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.

$$\textsc{PreserveInj}\ \frac{\Gamma_{\text{FiltPart}}(x_1) = (x_2, f_1, (f_2, \ldots, f_n)) \qquad \Gamma \vdash \text{Inj } x_2\ Y}{\Gamma \vdash \text{Inj } x_1\ Y} \qquad \textsc{InjSubset}\ \frac{\Gamma_{\text{Inj}}(x) = Y_2 \qquad \text{Query } (\Gamma,\ Y_1 \subseteq Y_2)}{\Gamma \vdash \text{Inj } x\ Y_1} \qquad \textsc{InjMono}_<\ \frac{\Gamma \vdash \text{Mono } x|_{x^{-1}(Y)}\ <}{\Gamma \vdash \text{Inj } x\ Y}$$

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

$$\textsc{InjConcat}\ \frac{\Gamma(x_1) = f_1 \mathbin{++} f_2 \qquad \text{fresh } x_2, x_3 \qquad \Gamma, x_2 \mapsto f_1, x_3 \mapsto f_2 \vdash \text{Inj } x_2\ Y \qquad \Gamma, x_2 \mapsto f_1, x_3 \mapsto f_2 \vdash \text{Inj } x_3\ Y \qquad \Gamma, x_2 \mapsto f_1, x_3 \mapsto f_2 \vdash \text{DisjointImg } x_2\ x_3}{\Gamma \vdash \text{Inj } x_1\ Y}$$

Fig. 11. Verifying properties using other properties (selected rules.) Each rule implicitly returns a new environment recording properties proven; the proper judgment is $\Gamma \vdash P \longrightarrow \Gamma$ where P is a property. The proof obligation for DisjointImg $x\ y$ is $\forall i \in \text{dom}(x)\ .\ \forall j \in \text{dom}(y)\ .\ x(i) \neq y(j)$.

As explained in Section 3.4, we infer structure from flattened programs via scatter. Rule Scatter3 covers the case where in-bounds written indices (x_{idx}) are strictly monotonically increasing,[1] producing a flat jagged array representation where rows start at the written indices. Importantly, the representation supports empty rows (corresponding to out-of-bound indices when $[\,\neg p\,]$ implies that the row is empty: $e_{row}[i_1/i_1+1] - e_{row} = 0$). The number of rows equals the index array length; otherwise, inference would be statically intractable. Judgments $\Gamma \vdash e^* \longrightarrow \text{Safe}$ (Appendix B.1.3) verify that scatter matches the deterministic semantics in Eq. (1).

Scatter3

$$\frac{\begin{array}{c}\Gamma \vdash \text{scatter } x_{dst}\ x_{idx}\ x_{src} \longrightarrow \text{Safe} \qquad \text{fresh } x_\bot, k, l, i_2 \\ \Gamma \vdash \lambda\ (i_1 : 0..|x_{idx}|)\ .\ \begin{cases}[\,x_{idx}(i) \in 0..|x_{dst}|\,] * x_{idx}(i) \\ [\,x_{idx}(i) \notin 0..|x_{dst}|\,] * x_\bot\end{cases} \rightsquigarrow \lambda\ (i_1 : 0..|x_{idx}|)\ .\ \begin{cases}[\,p\,] * e_{row} \\ [\,\neg p\,] * x_\bot\end{cases} \\ \text{Query } (\Gamma,\ e_{row}[i_1/0] = 0) \\ \text{Query } (\Gamma,\ e_{row}[i_1/|x_{idx}|] = |x_{dst}|) \qquad \text{Query } ((\Gamma, 0 \le j < k < |x_{idx}|),\ 0 \le e_{row}[i_1/j] \le e_{row}[i_1/k] \le |x_{dst}|)\end{array}}{\Gamma \vdash \text{scatter } x_{dst}\ x_{idx}\ x_{src} \longrightarrow \lambda\ (i_1 : 0..|x_{idx}| \times i_2 : 0..(e_{row}[i_1/i_1+1] - e_{row}))\ .\ \begin{cases}[\,i_2 = 0 \wedge p\,] * x_{src}(i_1) \\ [\,i_2 \neq 0 \vee \neg p\,] * x_{dst}(e_{row} + i_2)\end{cases}}$$

4.2.4 Concatenation. Concatenation of arrays x and y manifests to an index function:

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

We record a dual form $z = f_x \mathbin{++} f_y$ tracking z as a concatenation of x and y's index functions f_x and f_y. The manifested form handles indexing $z(e)$, while some properties are provable only on the concatenated form (Sections 4.3 and 5.1.2). 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 prepending/appending index functions covering elements before the first write ($0..e_{row}[i_1/0]$) and elements after the last write ($e_{row}[i_1/|x_{idx}|]..|x_{dest}|$), where empty ranges eliminate unnecessary concatenations. This makes the irregular index function invariant to whether e_{row} uses inclusive or exclusive scan.

4.3 Property layer

Properties are verified either by expanding the proof obligations listed in Fig. 8 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).

[1] Checked via scatter safety (implying injectivity of in-bounds indices) and the last query (establishing non-strict monotonicity of e_{row}).

Set of variables X
Array $a ::= x \mid \vee_\perp X$
Symbol $s ::= x \mid a[e] \mid \sum a[e:e]$

Legal range example

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

Illegal range example

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

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

```
1  let e' = Apply PeelOnRange to Simplify(Δ, e)
2  if e' is a constant n ∈ ℤ and n ≤ 0 then return Yes else return Unknown
3  let s = PickSym(Δ, e')
4  factorize e' by s yielding the form s · e1 + e2
5  lookup Δ_Range(s) = min L ≤ n · s ≤ max U
6  for each (l, u) ∈ L × U do
7    if Solve(Δ, −e1 ≤ 0) then if Solve(Δ, u · e1 + n · e2 ≤ 0) then return Yes
8    if Solve(Δ, e1 ≤ 0) then if Solve(Δ, l · e1 + n · e2 ≤ 0) then return Yes
9  return Unknown
```

Fig. 12. Algebraic language syntax and solver algorithm.

Low-level algebraic reasoning. Proof obligations are reified as solver queries as explained in Sections 3.2 and 3.3. The proof obligations of Range, Equiv, Mono, Inj, and Bij generalize to n dimensions by replacing i and j for vectors $\vec{i}$ and $\vec{j}$ of size n and changing ordering on these to be lexical. For example, the proof obligation for Mono $x \prec$ expands to $2^n - 1$ distinct queries:

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

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 11 shows the property rules. Γ_{Part} and Γ_{Filt} are aliases for $\Gamma_{\mathsf{FiltPart}}$ with the filter and partitioning predicates being $\lambda i.\,\mathrm{true}$, respectively. Premises like $\Gamma \vdash \mathsf{Inj}\ x\ Y$ may use either level of reasoning. PreserveInj propagates injectivity over filter/partitioning. InjSubset concludes that $x|_{x^{-1}(Y_1)}$ is injective if $Y_1 \subseteq Y_2$ and $x|_{x^{-1}(Y_2)}$ is injective. FilterInj proves injectivity of x_1 via x_2, which x_1 filters/partitions. The filter predicate is used to refine the values of x_2, possibly ignoring those that would violate injectivity: $\forall i, j \in |x_2| \,.\, p(i) \wedge p(j) \wedge x_2(i) \in Y \wedge x_2(j) \in Y \wedge x(i) = x(j) \Rightarrow i = j$. We reify this as an index function to leverage both reasoning levels rather than directly invoking the solver. These three rules work even when x_1 is uninterpreted (a formal argument or untranslatable source expression). InjMono$_<$ proves injectivity via strict monotonicity (recording both). InjConcat verifies injectivity via concatenated functions, requiring that the functions' images are disjoint. Additional rules in Appendix B.2 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 source code (e.g., combining the ranges of each branch in an conditional).

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. 8 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. (Shown in Appendix B.2.1.) The For property composes to express properties over inner dimensions. Flattened domains are handled by matching the outermost iterator in the flat dimension.

4.4 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. 12). 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 $\vee_\perp\{x_1, \ldots, x_n\}$. $\vee_\perp\{x_1, \ldots, x_n\}[i]$ is equivalent to $x_1[i] \vee \cdots \vee x_n[i]$ since $x_1, \ldots, x_n$ are mutually exclusive.

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) $\Delta_{\mathsf{Untrans}}$ 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 depend on any of the symbols appearing in the transitive closure of its ranges and equivalences. A symbol depends on another symbol if its *leading variable* (Appendix B.3.1) is in the free variables of the other symbol. 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 $x \notin \mathrm{fv}(i)$. 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.

4.4.1 Solver algorithm. Inequalities are reduced to form $e \leq 0$ and solved by the adaptation of Fourier-Motzkin elimination [16, 49] presented in Algorithm 1. It starts by simplifying e as presented in Section 4.4.2; 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.

4.4.2 Simplification strategy. Our simplification strategy has three steps. Step 1 extends sum slices to include terms that have equivalences in Δ_{Equiv}, which enables Step 2 to simplify across sum-sum or sum-element terms. Step 3 splits sums to peel off elements that have known equivalences. The rules used in each stage and the algorithm that applies them appear in Appendix B.3.2.

Step 1 applies rules Equiv1–2, EmptySum and ExtendSum. Equiv1 substitutes symbols bound in Δ_{Equiv} with their equivalent rewrites. Equiv2 replaces $\vee_\perp X[e]$ with 1 whenever $\exists x \in X$ such that $\Delta_{\mathsf{Equiv}}(x[e]) = 1$. EmptySum replaces sums of provably empty slices with 0. ExtendSum extends a sum slice to include a start/end element bound in Δ_{Equiv}, e.g., $\sum a[e_1 : e_2] \to \sum a[e_1 - 1 : e_2] - e_3$ when the slice is nonempty: $\Delta_{\mathsf{Equiv}}(a[e_1 - 1]) = e_3$ and $e_1 \leq e_2 + 1$.

Step 2 joins sums that are disjoint (JoinSums1–4 in Appendix B.3) and eliminates and extracts overlaps between sums (ElimSumOverlap and ExtractSumOverlap). For example, JoinSums4 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).

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

JoinSums1 rewrites two contiguous sum slices into one. JoinSums2 (JoinSums3) extend a sum slice to include the next (previous) element of the array. ElimSumOverlap eliminates the common part

1. **Query**

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

2. **Build context**

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

3. **Lower**

$$\begin{aligned} &\mathrm{Lower}(idx(i_1) < idx(i_2)) \\ &= \mathrm{Lower}([\,p(xs(i_1)) \wedge p(xs(i_2))\,] * (\textstyle\sum_{j=0}^{i_1-1}(p(xs(j))) < \sum_{j=0}^{i_2-1}(p(xs(j)))) \\ &\qquad + [\,p(xs(i_1)) \wedge \neg p(xs(i_2))\,] * (\textstyle\sum_{j=0}^{i_1-1}(p(xs(j))) < i_2 + \sum_{j=i_2+1}^{n-1}(p(xs(j)))) + \ldots) \\ &= (\Delta,\ \textstyle\sum \vee_{\perp}\{x_1\}[0 : i_1 - 1] < i_2 + \sum \vee_{\perp}\{x_1\}[i_2 + 1 : x_{|xs|} - 1]) \end{aligned}$$

4. **Solve**

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

Fig. 13. Lowering the query discussed in Section 3.3.

of two overlapping sum slices that are subtracted from each other. ExtractSumOverlap extends this treatment of sum-slice subtraction to mutually-exclusive boolean arrays ($\vee_{\perp} X$).

Step 3 peels off elements from sums that have known equivalences (or more specialized ranges) by applying EmptySum followed by a fixpoint application of rules SingletonSum, PeelEquiv1-2, EmptyDisjunction. They 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.

4.4.3 Lowering. Lowering a query to the algebra layer has two stages (Fig. 13). First, the algebraic context Δ is built from the environment Γ and the index function associated with the query. In Fig. 13, 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.1.2), or, (2) assuming p by extending Δ to Δ' with any ranges and equivalences in p, $\mathrm{Solve}(\Delta', e)$ must return Yes. The bottom of Fig. 13 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 supporting nD regular arrays and 2D flat jagged arrays. We support additional source constructs: tuples, (un)zip, concat, analysis over loops and histograms [22], 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.

5.1 Experimental evaluation

We demonstrate P^2's ability to statically verify properties of bulk-parallel operations in 10 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 adhere to deterministic semantics (Sections 3.2, 5.1.1 and 5.1.2), and obviating dynamic bounds checks in indexing and gathers (all subsections).

Experimental overview. The benchmarks include Futhark implementations of real-world algorithms, such as the maximal matching graph algorithm from Problem Based Benchmark Suite [2]

Program	Properties & annotations	Safe	#S	#A	Check time	% of Compile time
max_match	Range, Equiv, Inj, FP	✓	6	14	1.6s	65%
MIS	Range	✓	3	35	7.9s	95%
FFT	Inj	✓	1	1	0.2s	16%
primes	Range, FP	✓	2	12	2.1s	82%
kmeans_ker	Range	✓	0	3	0.1s	13%
partition	Equiv, FP	✓	1	1	0.2s	31%
partition3	Equiv, FP	✓	1	2	0.8s	64%
seg_partition	Range, Equiv, FP	✓	1	3	2.2s	83%
filter	Equiv, FP	✓	1	3	0.1s	21%
filter_irreg	Range, Equiv, InvFP	✓	1	3	0.8s	63%

Program & Data	Dyn. (*ms*)	Speedup Static	Speedup +Opt
kmeans_ker			
movielens	280	2.2×	
nytimes	315	1.9×	
scrna	861	2.2×	
partition2			
50M	12	4.4×	1.08×
100M	38	7.0×	1.08×
200M	135	12.2×	1.05×

Fig. 14. Left: Summary of evaluated programs. FP abbreviates FiltPart. Safe indicates whether all indexing and scatters are verified. #S and #A denote scatters and annotations. Check time measures P^2's runtime (Apple M4 chip). Right: NVIDIA A100 performance with dynamic checks (Dyn.) as baseline. Static shows speedup over dynamic checks. +Opt additionally removes scattered array initialization (speedup over Static).

(max_match) as well as the maximal independent set algorithm (MIS), the Cooley-Tukey FFT algorithm [12] (FFT), sparse k-means [38] (kmeans_ker), and an optimal work-depth implementation of prime sieve [5] (primes). We also evaluate widely-used kernels such as filter, two- and three-way partitioning (partition3) and segmented filters (filter_seg) and partitions (seg_partition). For example, partition and seg_partition are key components of radixsort and quicksort. To our knowledge, none of these benchmarks have been verified in the data-parallel context. The most challenging to verify are the ones that flatten irregular nested parallelism: seg_partition and primes.

Verification checks that array indexing is within bounds and that scatters are safe, in addition to other specified properties. Figure 14 (left) summarizes the evaluation, including the number of scatters and the kind of properties for each benchmark. Check times span from under 1 second up to 8 seconds, increasing with complex index functions requiring many normalization rewrites (e.g., seg_partition) or many annotations (e.g., MIS). Figure 14 (right) compares dynamic versus static verification on an NVIDIA A100 GPU. Futhark inserts dynamic bounds checks in CUDA kernels [21]. For kmeans_ker, static verification achieves roughly 2× speedup on datasets movielens [20], nytimes, and scrna [29] (using parameters from [38]) by eliminating dynamic checks. Futhark does not dynamically verify scatter determinism in Eq. (1). For partition2, we manually implement dynamic scatter checks using reduce-by-index [22] to match the program's asymptotic work. 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 leaving the scattered array uninitialized—safe because P^2 automatically proves all locations are overwritten—yielding an additional 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 5.1.1, 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:

def seg_partition (s : []i64 | Range s 0..∞) (x : []f64 | Equiv $|x|$ (sum s)) (p : []bool | Equiv $|x|$ $|p|$)
: []f64 | λy. For (k : 0..$|s|$) (Part y x (λi. $p[i]$))

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. 6a):

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

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

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 4.4.2. The sum-slice simplifications are critical here because boundaries are themselves sums. See Appendix C.2 for the implementation and an illustrative lowered query with solving steps.

```
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 is = (flatten iss1) ++ (flatten iss2)
    let vs = ... in scatter x is vs
```

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

5.1.2 FFT. The Cooley-Tukey FFT algorithm produces scatter indices *is* comprising strictly monotonic segments (e.g., $[0, 1, 4, 5, \ldots] ++ [2, 3, 6, 7, \ldots]$). To verify scatter safety, we must prove that *is* is injective. Concatenation manifests the term $\boxed{-2^{n+q}}$ 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, P^2 further exploits the dual unmanifested form:

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

Using InjConcat (Fig. 11), we prove 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 and concluding remarks

Liquid types and Liquid Haskell. Liquid types are refinement types [35] automatically discharged by SMT solvers [46]. Refinement reflection [47] 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, Liquid Haskell experts[2] explained that automatically verifying that `let is = map (\c-> if c then 1 else 0) cs in zipWith (\c i-> if c then i-1 else 1) cs is` produces positive integers requires fusion to maintain positional correlation between `cs` and `is`—contrary to the bulk-parallel programming style. It can be proved by manually defining and applying a user lemma, which is unintuitive for non-experts. 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* [41] 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. Pulse [42], embedded in F*, uses concurrent separation logic [9] to reason about concurrent mutable state. For example, verifying

[2]We are thankful for their help and will thank them in the acknowledgments (if permitted).

in-place partitioning for quicksort checks buffers against a pivot and ensures non-overlap, but doesn't extend to data-parallel contexts using bulk operations and scatter.

Linear Array Logics. Dependent ML [50] and ATS [51] restrict dependent values to limited languages for decidability. Dependent ML enables static array bounds checking via linear constraints [52]. ATS allows explicit proof terms but requires intertwining proof and program. Daca et al.'s [13] logic for counting and partitioning, Bradley et al.'s [8] for index ranges and sortedness using Presburger arithmetic, and Qube [43] for array indexing and shape matching are restricted to linear indexing. We target programs with non-linear indexing via gather/scatter (see Section 5).

Dependence analysis. P^2 is inspired by work in automatic loop optimization where suitable representations for access patterns [27, 36] are key to scaling interprocedural analysis, either statically [18, 48] or via static-dynamic combinations [30, 36] for non-affine code. Dynamic analyses use transformations like program slicing and hoisting to extract and check sufficient conditions at runtime for statically irreducible queries, often establishing array properties such as permutation [15, 40], injectivity [14, 36], and monotonicity [31, 32]. By establishing array properties early, our approach could simplify dependence analyses and eliminate runtime overheads.

Verified compiler transformations. Work on verifying scheduling DSLs like Halide [34] includes improvements to its term rewriting system [28], validation of affine specifications [10], and HaliVer [45], which verifies specifications using linear indexing and low-level generated code via permission-based separation logic [6] for memory safety. Bounded translation validation tools like Alive2 [25, 26] verify LLVM code transformations. These directions are complementary to our work.

Conclusions. To our knowledge, P^2 is the first effort to fully automatically verify integer array properties—such as monotonicity, bijectivity, filtering, and partitioning—in a data-parallel context that supports non-linear indexing produced by prefix sums, scatters and gathers. Our evaluation demonstrates verification of ten challenging (previously unreported) data-parallel benchmarks, including programs resulting from flattening iregular-nested parallelism.

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] 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

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

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

[6] 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

[7] 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*.

[8] 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.

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

[10] 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

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

[12] 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

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

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

[15] 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

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

[17] 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

[18] 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.

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

[20] 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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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.

[30] 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

[31] 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.

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

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

[38] 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

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

[40] 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

[41] 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.

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

[43] 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).

[44] 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.

[45] 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.

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

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

[52] 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