UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Static Verification of Array Properties for Segmented Arrays

Nikolaj Ingemann Gade (qhp695) & Tudor-Ovidiu Pal (szh252)

Advisor: Cosmin Eugen Oancea
Co-advisor: Nikolaj Hey Hinnerskov

Submitted: May 31, 2026

Abstract

Data-parallel programs that work with irregular (jagged) data flatten it into a single flat array, paired with a shape array recording each segment's length. This layout suits parallel hardware, but it erases the segment boundaries from the data and leaves them implicit in the shape array. A static verifier that wants to prove per-segment properties, such as that an index array stays in bounds, sends no two elements to the same position, or describes a partition, must first recover that hidden structure from the flat representation.

PropProp, a static analysis in the Futhark compiler, verifies such array-content properties by translating programs into symbolic index functions. Its framework already represents segmented data through a flattened domain that exposes the segment iterator and per-segment offset directly, but its implementation produced this representation only for regular flattening, where every segment has the same length. Irregular segments, whose lengths vary, were instead carried in an older `Cat` encoding that the rules for recognizing per-segment computations and indexing could not consume directly.

This thesis closes that gap with four implementation-level contributions to PropProp. The first generalizes the PropFlatten rule, implemented as `propFlattenOnce`, to the irregular case, computing each segment's start as a running sum of the preceding segment lengths rather than as a single multiplication. The second migrates the segmented scatter rule, `scatterSc2`, to produce the explicit flattened representation directly; since scatter is the main point where segmented structure first enters the analysis, this also leaves the `Cat` form unproduced throughout the system. The third makes substitution work across flattened irregular domains, so that once such a domain has been propagated, the analysis can substitute through applications over it and simplify the result. The fourth adds support for row-wise inverse filtering/partitioning properties over flattened domains: the For-InvFiltPart-Flat rule reduces such a per-segment property to an ordinary one-dimensional InvFiltPart proof. All four changes are checked structurally, both in isolation and through program-level regression tests.

Contents

1 Introduction

The earlier a problem in a program is found, the cheaper it is to fix. The earliest any tool can find one is during compilation, the step that creates the final program. Compilation is usually quick, while running the finished program can take minutes or hours on a large input, so a problem caught during compilation is caught long before any of that running time is spent. This thesis is about a way of finding problems during compilation, called *static verification*. Instead of running the program, static verification examines the program itself and shows that certain kinds of failure cannot happen on any input at all.

Verifying a program this way gets harder when the program is built to do many things at once. Graphics processors and similar hardware are fast because they apply the same operation to thousands of elements in parallel. A single such parallel operation needs all of its input elements sitting together, so when the data is naturally divided into segments of different sizes, the program flattens it: every element of every segment is laid end to end in one flat array, and a second array, the shape array, records how many elements each segment contains. The flat array is what the parallel operation runs over, and the shape array is all that remains of the segmentation. This is efficient, but it removes information that a static verifier would need to prove certain things. The boundaries between segments are gone from the data and survive only in the shape array, so before the verifier can prove anything segment by segment, it has to rebuild that structure.

Suppose we want to sort a class of students so that everyone who passed comes first, followed by everyone who failed, with each student keeping their original order within the pass/fail group. On its own this is simple. The difficulty appears when there are several classes and we want to process them together. To do that in parallel, every student from every class is placed in one long list, with a shape array recording how many students are in each class. Figure 1 shows such a list for three classes of sizes two, three, and one. We want to split up the students within each class, processing all classes at the same time. However, the program sees only the single list and the shape array. The class boundaries are not present in the list itself, and the verifier must recover them from the shape array.

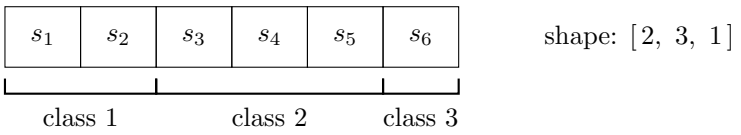

Figure 1: A list of six students s_1 through s_6, drawn from three classes of sizes two, three, and one. The division into classes survives only in the separate shape array. The single list is what the hardware processes in parallel, and the shape array is what remembers the structure.

Recovering that structure also matters for the guarantees we most want before the

program runs. A data-parallel program often produces its output with an operation called `scatter`. A `scatter` computes a position for each element of its input and writes all of them into the output in parallel. For the result to be trustworthy, those computed positions have to fit certain criteria. The main requirement is that no two elements are sent to the same position. Otherwise one would silently overwrite the other, and the output would depend on which write happened to win. It's also nice to know if every position lies inside the output, so that no element is quietly dropped. These requirements are about which positions the program chose, which is to say the contents of the index array it builds. Guarantees about the contents of the arrays a program computes (rather than their sizes) are what we call *array properties*.

Several tools for proving such properties already exist. PropProp [9], for example, is a static analysis inside the Futhark compiler [8]. Rather than running a program, it constructs a symbolic description of the contents of each array and proves a fixed set of properties over those descriptions, including that an index array stays in bounds, that it never places two elements at the same position, and that it describes a clean partition. PropProp already handles flattened data when every segment has the same size. The case it could not handle is when the segments of a flattened array have different sizes, like the classes of sizes two, three, and one above. On paper the framework covered this case, but the implementation did not. For irregular segments it fell back on an older encoding of the segment structure that the rest of the analysis could not use as directly.

This thesis makes the implementation handle the irregular case directly, in the same way it already handles the regular one.

1.1 Contributions

The thesis makes four implementation-level contributions to PropProp.

The first lets the tool handle segments of different sizes, not just segments that are all the same size. In the terms of the framework, it generalizes the PropFlatten rule, implemented as the function `propFlattenOnce`, to the irregular case, where a segment's length can depend on which segment it is. The regular case finds the starting position of a segment with a single multiplication and checks total size with a single equality. The irregular case instead computes each segment's start as a running total of the lengths of the segments before it, and checks the total size against that same running sum across all segments. Otherwise an irregular intermediate stays in the old indirect encoding, which the rules for per-segment computation and indexing cannot consume.

The second changes how the tool records segmented structure internally, so that the rest of the analysis can use it directly instead of reconstructing it each time. It migrates the segmented scatter rule, `scatterSc2`, so that the rule produces the explicit flattened representation used elsewhere in the analysis rather than the older `Cat`-based one. Scatter is the main point where segmented structure first enters the analysis, so this is also the change that leaves the `Cat` form unproduced everywhere in the system. We do not remove `Cat` from the underlying data type. It stays defined, and some legacy utilities still match on it. What changes is that the analysis no longer generates it.

The third makes substitution work across flattened irregular domains, so that once a

flattened domain has been propagated, the analysis can substitute through applications over that domain and simplify the result. It centers on `substituteOnce`, which inlines the body of one index function into another. When a call site supplies a single flat argument for a dimension that the index function exposes as a flattened domain, the helper `from1Dto2DM` splits that flat argument into an outer segment coordinate and an inner local offset, with a new branch for the irregular case where the row start is a prefix sum rather than a multiplication. Substitution introduces symbolic segment indices of the form $\%_D(e_{idx})$, which `solveIdx1` then resolves to a concrete segment iterator by proving that the flat index lies within the current row, including a one-dimensional case needed when the surrounding index function is not itself flattened.

The fourth lets the tool prove the partitioning property segment by segment rather than over the whole flat array at once. The analysis previously carried only the older `FiltPartInv` property, which has no explicit image range, so it could not say which range each segment's indices should fall in. The formal `InvFiltPart` property adds such a range, and wrapping it in `For` lets the property be required separately for each segment, over that segment's own flat interval. The new rule For-InvFiltPart-Flat, implemented in `matchProof`, reduces this to an ordinary one-dimensional `InvFiltPart` proof: it builds a one-segment index function, rewrites the predicates so they refer to positions inside that segment, and reuses the existing one-dimensional `InvFiltPart` prover. The payoff is that `part2indicesL` can state its postcondition directly as a per-segment property, even though the program still stores all segments in one flat array.

All four changes are checked structurally rather than by measuring performance. The tests confirm that the rewrites behave as the rules specify, that the migrated scatter rule no longer emits the old representation and instead exposes the segmented structure in the new form, and that substitution across flattened domains resolves to the expected segment coordinates, both on their own and when reached through the normal analysis of complete programs.

1.2 Thesis Outline

The rest of the thesis follows the partitioning example from the easy setting to the hard one. Section 2 introduces Futhark and PropProp and shows how segmented data is stored as one flat array alongside a separate shape array. Section 3 defines the small language fragment the analysis works on. Section 4 walks through the single-segment version of partitioning, where the core machinery is easiest to see. Section 5 lifts that machinery to multiple segments at once and derives the segmented partition, the multi-class version of the same example, showing how the hidden structure is recovered one step at a time. Section 6 presents the four changes as they appear in the compiler, and Section 7 reports how each was tested. Section 8 surveys related work, and Section 9 concludes with a summary and future work.

2 Background

2.1 Futhark

Futhark [8] is a statically typed, purely functional data-parallel programming language that targets highly-parallel hardware, especially GPUs. Its programming model is centered around bulk operations over arrays, such as `map`, `scan`, `reduce`, and `scatter`. These operations make parallelism explicit while keeping the source language functional and side-effect free.

This style is important for verification because many programs compute arrays of indices using a sequence of bulk-parallel operations and then use those indices for indirect reads or writes. For example, a program may compute a permutation, use it to gather values from another array, or pass it to `scatter` to place values in their final positions. The correctness of such programs often depends not only on array sizes, but also on the values stored in the index arrays.

2.1.1 The Size Type System

Futhark has a size-dependent type system [6] that tracks array sizes at the type level. Function signatures can express that two arrays have the same length, or that an operation such as `map` preserves the length of its input.

The size system is deliberately restricted. It tracks shape information in a way that keeps type inference practical, but it does not attempt to prove arbitrary arithmetic facts about computed sizes. When a computation produces an array whose size cannot be expressed precisely in the size language, such as the result of a `filter`, the type system may assign it an existential size. This allows the program to remain typeable, but the type checker no longer knows more detailed relationships between that size and other program values.

Futhark also provides explicit size coercions using `:>`. These coercions let the programmer state that a computed array has a particular size, and the compiler inserts a dynamic check when the equality cannot be established statically. Thus, the size type system is very useful for tracking regular shape information, but more complex relationships involving computed arrays may still require runtime checks.

2.1.2 The Verification Gap

Shape information alone does not rule out many important errors in data-parallel programs. For example, it does not show that every element of an index array is within the bounds of another array, that an index array is injective, or that a `scatter` is deterministic. These are properties of array contents rather than array shapes.

PropProp [9] addresses this gap by adding a compiler-integrated layer of static reasoning about selected array properties. It supports a fixed set of properties, including range, equivalence, injectivity, bijectivity, monotonicity, filtering, and partitioning. The restriction to a fixed property language is intentional: PropProp is not a general-purpose

theorem prover, but a specialized analysis for common verification tasks in data-parallel array programs.

2.1.3 Segmented Arrays

Futhark does not provide irregular, or jagged, arrays as a first-class type. Arrays are regular: all rows of a nested array must have compatible shapes. Programs that work with irregular data therefore use a segmented representation, where the data is stored in one flat array and a separate shape array records the length of each logical segment.

For example, a logical jagged array with segment lengths $\texttt{shape} = [2, 0, 3]$ is represented by a flat data array whose first two elements belong to segment 0, and the next three elements belong to segment 2, because segment 1 is empty, as its shape is 0. A logical position (k, j), where k is the segment number and j is the local index inside that segment, corresponds to the flat position $\sum_{l=0}^{k-1} \texttt{shape}[l] + j$.

This encoding is efficient and fits Futhark's regular array model, but it makes static reasoning harder. The program operates on flat arrays, while the logical structure is hidden in auxiliary arrays such as shapes, offsets, or flags. A verifier must therefore be able to relate flat positions back to their segmented meaning. This issue is central to the segmented and flattened programs studied later in this thesis.

2.2 PropProp

PropProp is a static analysis implemented in the Futhark compiler. Its purpose is to verify selected properties of array programs automatically, especially properties involving index arrays, indirect indexing, and `scatter`. Instead of asking the user to provide arbitrary proofs, PropProp works with a fixed set of predefined properties and exploits the semantics of Futhark's bulk-parallel operators.

The supported properties include range, equivalence, injectivity, bijectivity, monotonicity, filtering, and partitioning. These properties are expressive enough to describe many of the facts needed for safe and efficient data-parallel programs. For example, a range property can show that all indices are within bounds, an injectivity property can show that a `scatter` has no conflicting writes, and a partitioning property can describe how an output array rearranges the elements of an input array.

PropProp transforms a type-checked Futhark program into symbolic descriptions of array contents and then proves properties over those descriptions. The central symbolic object is the index function. An index function describes, for each valid index of an array, what value is stored at that position. The next chapters introduce index functions. For now, the important point is that index functions let PropProp reason about array values algebraically rather than by following individual runtime executions.

PropProp analyses a program in a forward direction. Function preconditions are assumed while analyzing the function body and checked when the function is called. Function postconditions are verified during the analysis of a function definition. If a property cannot be proved, PropProp does not conclude that the property is false; it

returns *unknown*. For required checks, such as postconditions, bounds checks, or scatter-safety checks, an unknown result means that the analysis cannot statically justify the program at that point.

2.2.1 Architecture

PropProp is organized into three interacting layers, shown in Figure 2.

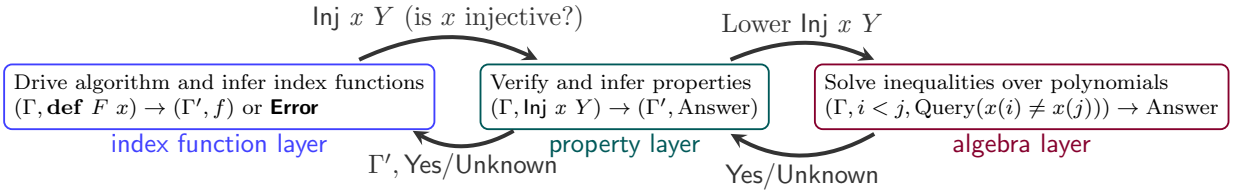

Figure 2: High-level organization of PropProp. The index function layer constructs symbolic descriptions of array contents, the property layer proves and propagates array properties, and the algebra layer discharges the low-level arithmetic queries generated during verification. This figure is taken from the PropProp paper. [9]

The index function layer is responsible for constructing symbolic descriptions of array contents. It traverses the program one binding at a time and assigns index functions to intermediate arrays. This works well because Futhark programs are represented in a form where each binding depends only on earlier bindings. For simple bulk operations, the connection between input and output positions is direct. For example, a `map` preserves the domain of its input and transforms each element independently. For `scan`, the layer must represent the dependency on previous elements, which is later normalized into a symbolic expression involving sums. For `scatter`, the layer often needs additional information before it can construct a precise index function: if the index array is known to be bijective or injective on the relevant range, then the result of the scatter can be described much more precisely.

The property layer records and proves properties over the index functions produced by the first layer. It maintains an environment of known facts, such as range information, injectivity, monotonicity, or postconditions propagated from function calls. The property layer uses two styles of reasoning. High-level reasoning works directly with known properties. For example, it can use the fact that certain operations preserve range or injectivity properties. Low-level reasoning reduces a property to algebraic proof obligations. For instance, proving that an index array is injective can be reduced to showing that two different source positions cannot map to the same target position. These generated proof obligations are then sent to the algebra layer.

The algebra layer discharges the low-level arithmetic queries generated by the property layer and by bounds checks in the index function layer. These queries are not arbitrary mathematical formulas; they are expressed in the restricted symbolic language used by PropProp. This language includes integer expressions, array-indexing symbols, guards from conditionals, and sums over slices of arrays. The solver is based on Fourier–Motzkin elimination, extended with rewriting techniques for expressions that commonly appear in data-parallel programs, such as overlapping sums and mutually exclusive guarded cases.

2.2.2 The Algorithm

PropProp analyses a program by walking through it in source order and gradually building an environment Γ of known information. This environment contains the index functions inferred for program variables, the properties that have already been proved, summaries for analyzed functions, and facts that can later be used when solving queries. The algorithm is therefore forward: each new binding may add information that later bindings can use.

The analysis starts at the level of function definitions. For each function, PropProp first translates the source-level pre- and postconditions into the property language used by the analysis. The preconditions are assumed while the body is analyzed, because they describe what must hold for the formal arguments. The formal arguments themselves are represented symbolically, so the analysis can reason about the function without knowing the concrete arrays that will be passed at runtime.

The body of a function is then processed one let-binding at a time. For a binding

$$\textbf{let } x = E_0 \textbf{ in } E,$$

the right-hand side E_0 is converted into one or more index functions. The result is normalized by rewriting, so that later reasoning sees a simplified symbolic description rather than the immediate raw result of the conversion. PropProp also tries to infer properties from the binding. For example, if E_0 is a function call, the callee's postcondition can be attached to the result variable x. The inferred index function and any inferred properties are then added to Γ, and the analysis continues with the remaining body E.

Function calls use the summaries stored in the environment. When a function is called, PropProp checks that the actual arguments satisfy the callee's preconditions. If the callee has a useful index-function summary, that summary is reused by substituting the actual arguments for the formal parameters. If no such summary is available, the result can still be represented symbolically as an uninterpreted function, while the callee's postconditions may still provide useful properties about the result.

After the function body has been analyzed, PropProp verifies the function postcondition using the information accumulated in Γ. Some properties can be proved directly from known facts in the environment. Others are reduced to lower-level algebraic queries, which are sent to the algebra layer. If the postcondition is proved, the function summary is recorded so later calls can reuse it. If a required postcondition, bounds check, or scatter-safety check cannot be proved, the analysis reports failure at that point.

The notation used for the algorithm distinguishes between conversion, rewriting, and querying. A conversion step constructs an index function from a source construct. A rewriting step simplifies an index-function expression or an index function until it reaches a stable form. A query step asks whether a property-derived condition can be proved. The symbol $\rightsquigarrow$ is used for the combined operation: perform the relevant conversion or query preparation, and then rewrite the result to a fixed point. Figure 3 summarizes the main relations.

Relation	**Meaning**
$\Gamma \vdash e \rightsquigarrow e'$	Rewrite an internal symbolic expression e until no more rewrite rules apply.
$\Gamma \vdash f \rightsquigarrow f'$	Rewrite an index function f to a stable form, usually by simplifying its body and sometimes its domain.
$\Gamma \vdash B \rightsquigarrow f'$	Convert a simple source-level base expression B, such as a variable, constant, or array lookup, into an index function and then rewrite it.
$\Gamma \vdash E_0 \rightsquigarrow (\Gamma', f'_1, \ldots, f'_n)$	Convert the right-hand side of a let-binding into one or more index functions, possibly extending the environment, and rewrite the results.
$\Gamma \vdash E \rightsquigarrow (\Gamma', f'_1, \ldots, f'_n)$	Analyze the remaining body of a function, including its sequence of let-bindings, and return the index functions for the final result.
$\Gamma \vdash \mathsf{Query}_x(e) \rightsquigarrow_Q A$	Rewrite the expression inside a query and ask the solver whether it can be proved. The answer A is either Yes or Unknown.

Figure 3: High-level view of the main algorithmic relations in PropProp. The relations describe how source constructs are converted to index functions, how symbolic expressions are normalized, and how proof obligations are sent to the solver.

This organization explains why the three layers cannot be treated as independent components. The index function layer constructs the symbolic descriptions that properties are stated over. The property layer records and proves facts about those descriptions, and it is sometimes consulted while index functions are being built. The algebra layer discharges the lower-level queries generated by property verification.

This thesis builds on PropProp, but focuses on programs that use segmented and flattened representations of irregular data. In such programs, the source code manipulates flat arrays, while the verifier needs to recover and preserve hidden segmented structure. The following chapters first introduce the one-dimensional framework in a self-contained way, and then extend the discussion to flattened and segmented domains.

3 Source Language

The previous section described PropProp as a forward analysis over source programs. This section defines the source language fragment that the analysis operates on. The purpose is not to repeat the algorithm, but to make clear what kinds of expressions, properties, and function definitions appear in the formal rules used later in the thesis.

The language is a simplified core language rather than the full Futhark language. It contains the constructs needed by the analysis: arrays, bulk-parallel operators, function calls, conditionals, loops, and property annotations.

3.1 Syntax

Figure 4 gives the grammar used in the rest of the thesis. Variables are written x, y, z, function names are written F, and integer constants are written n. Types include base types and arrays. Base expressions contain variables, constants, array lengths, arithmetic and boolean operations, and array indexing. Bound expressions E_0 are the right-hand sides of let-bindings. These include base expressions, sequences, `map`, `scan`, `scatter`, function application, conditionals, and loops. Full expressions E are let-chains ending in one or more result variables.

3.2 Properties and Annotations

Properties are written in the source language as annotations, but the analysis reasons about them as facts over index functions. The property grammar includes range, equivalence, injectivity, bijectivity, monotonicity, filtering, and partitioning properties. These are the same kinds of properties described at a high level in the background chapter.

The source-level property expressions E_π are deliberately more restricted than arbitrary source expressions. They include base expressions and explicit sums over ranges. This restriction is important because properties are eventually lowered to algebraic queries. For example, a range property can be used to prove that an array lookup is in bounds, while an injectivity or bijectivity property can be used to reason about a `scatter`.

The following chapters use two examples of such annotations. The one-dimensional framework chapter uses `part2indices`, whose postcondition is expressed using `InvFiltPart`. Later, the segmented framework chapter uses examples where the same kinds of properties must be interpreted over flat arrays that encode segmented structure.

3.3 Relation to Futhark

The source language in Figure 4 is not meant to include every feature of Futhark. It is the fragment needed to describe the analysis. In particular, it focuses on pure array computations built from bulk-parallel operators and on the annotations that PropProp can verify.

$$
\begin{array}{rcll}
\beta & ::= & \texttt{i64} \mid \texttt{f64} \mid \texttt{bool} \mid \cdots & \text{base types} \\
\tau & ::= & \beta \mid []\tau & \text{types} \\
Op & ::= & < \mid \leq \mid > \mid \geq \mid + \mid - \mid * \mid = \mid \neq \mid \wedge \mid \vee & \text{operators} \\
B & ::= & x & \text{variable} \\
 & \mid & n & \text{constant} \\
 & \mid & |x| & \text{array length} \\
 & \mid & 2^B & \text{power of two} \\
 & \mid & B\ Op\ B & \text{binary operation} \\
 & \mid & x[B_1, \ldots, B_n] & \text{array indexing} \\
E_\pi & ::= & B \mid \mathsf{Sum}\ x[E_\pi : E_\pi] & \text{property expressions} \\
\pi & ::= & \mathsf{Range}\ x\ E_\pi\ E_\pi & \text{range} \\
 & \mid & \mathsf{Mono}\ x\ \prec & \text{monotonicity} \\
 & \mid & \mathsf{Equiv}\ x\ E_\pi & \text{equivalence} \\
 & \mid & \mathsf{Inj}\ x\ (E_\pi, E_\pi) & \text{injectivity} \\
 & \mid & \mathsf{Bij}\ x\ (E_\pi, E_\pi)\ (E_\pi, E_\pi) & \text{bijectivity} \\
 & \mid & \mathsf{InvFiltPart}\ x\ (E_\pi, E_\pi)\ (\lambda x.\ E_\pi)\ \overline{(\lambda x.\ E_\pi)} & \text{inverse filtering/partitioning} \\
 & \mid & \mathsf{FiltPart}\ x\ x\ (\lambda x.\ E_\pi)\ \overline{(\lambda x.\ E_\pi)} & \text{filtering/partitioning} \\
 & \mid & \mathsf{Filt}\ x\ x\ (\lambda x.\ E_\pi) & \text{filtering} \\
 & \mid & \mathsf{Part}\ x\ x\ (\lambda x.\ E_\pi) & \text{partitioning} \\
 & \mid & \mathsf{For}\ (i : E_\pi..E_\pi)\ \pi & \text{bounded property} \\
E_0 & ::= & B & \text{base expression} \\
 & \mid & B..B & \text{sequence} \\
 & \mid & \texttt{map}\ (\lambda \overline{x}.\ E)\ \overline{x} & \text{map} \\
 & \mid & \texttt{scan}\ (\lambda \overline{x_1}\ \overline{x_2}.\ E)\ \overline{B}\ \overline{x} & \text{scan} \\
 & \mid & \texttt{scatter}\ x_{dst}\ x_{idx}\ x_{val} & \text{scatter} \\
 & \mid & F\ \overline{x} & \text{function application} \\
 & \mid & \texttt{if}\ B\ \texttt{then}\ E\ \texttt{else}\ E & \text{conditional} \\
 & \mid & \texttt{loop}\ \overline{x} = \overline{x_{init}}\ \texttt{while}\ x\ \texttt{do}\ F\ \overline{x} & \text{while loop} \\
 & \mid & \texttt{loop}\ \overline{x} = \overline{x_{init}}\ \texttt{for}\ x < B\ \texttt{do}\ F\ \overline{x} & \text{for loop} \\
E & ::= & \texttt{let}\ \overline{x} = E_0\ \texttt{in}\ E \mid \overline{x} & \text{expressions} \\
Fun & ::= & \texttt{def}\ F\ \overline{(x : \tau \mid \overline{\pi})} : (\tau \mid \lambda \overline{x}.\ \overline{\pi}) = E & \text{function definition} \\
Prog & ::= & \epsilon \mid Fun\ Prog & \text{programs}
\end{array}
$$

Figure 4: Source language fragment used by the PropProp analysis. Taken from [9].

4 Framework Overview for One-Dimensional Index Functions

This section introduces the formal framework in the simple one-dimensional setting. Its purpose is to explain the basic notation, grammar, and inference style used by PropProp before introducing the segmented case. We focus only on the fragment needed for ordinary one-dimensional index functions, since this is the easiest setting in which to understand how source programs are translated into symbolic array descriptions.

4.1 Notation and grammar for the one-dimensional case

This subsection fixes the notation used in the rest of the section. The main formal object is the *index function*, which symbolically describes the contents of an array by mapping valid indices to guarded expressions.

Core notation. We begin with the most frequently used symbols, since these will reappear in both the formal rules and the running examples later in the chapter. Table 1 gives the meaning of the main symbols we use.

Notation	Meaning
f, g, h	index functions
$\text{shape}(f)$	domain of index function f
$\text{body}(f)$	guarded expression of f
i, j	iterator variables
e	symbolic expression
p	predicate / guard
`Iota` e	integer range $0, \ldots, e-1$
$fv(e)$	free variables of expression e
$e[x := e_y]$	expression obtained by substituting symbol x with expression e_y in the target expression e

Table 1: Notation used in the one-dimensional framework overview.

Grammar. For the one-dimensional case, we only need a small fragment of the full formalism. Figure 5 shows the grammar for the one-dimensional index functions used in this section.

$$
\begin{aligned}
p &::= \texttt{true} \mid \texttt{false} \mid e \le e \mid \neg p \mid p \wedge p \mid p \vee p \mid \cdots \\
t &::= n \mid s \mid s \cdot t \\
e &::= t \mid t + e \\
D &::= i : 0..e \\
a &::= [p] * e \\
G &::= a \mid a + G \\
f &::= \lambda(D).\, G
\end{aligned}
$$

Figure 5: Grammar for one-dimensional index functions

An index function therefore has the form $\lambda(D).\, e$, where D is its domain and e is a guarded expression. Intuitively, the domain describes which iterator values are valid, and the body describes the symbolic value stored at those iterator values.

Source-language categories and judgements. The inference rules in this chapter use the source-language categories introduced in Section 3. We use B for base expressions, such as variables, constants, arithmetic expressions, and array indexing. These are expressions that can be converted directly into scalar or array index functions. We use E_0 for the right-hand side of a let-binding, such as a `map`, `scan`, `scatter`, function call, conditional, or loop. Full expressions E are let-chains ending in one or more result variables.

The rules are written under an environment Γ. As introduced in Section 2.2.2, Γ stores the information known at the current program point: inferred index functions, proven properties, iterator ranges, predicates assumed in scope, and function summaries. The notation $\Gamma \vdash B \rightsquigarrow f$ means that the base expression B is converted into an index function and normalized. Similarly, $\Gamma \vdash E_0 \rightsquigarrow (\Gamma', f_1, \ldots, f_n)$ means that a let-binding right-hand side is converted into one or more index functions, possibly extending the environment. Finally, $\Gamma \vdash e \rightsquigarrow e'$ is used for rewriting an internal symbolic expression to a normalized form.

The inference rules also use the arrow $\to$ in their conclusions. This denotes a single rewrite step produced by a rule, as opposed to $\rightsquigarrow$, which denotes rewriting to a fixed point (the relevant rule applied as many times as necessary, together with the surrounding normalization rewrites).

In this chapter we only use the parts of these judgments needed for the one-dimensional examples. The segmented chapter extends the same notation with flattened domains, but the basic reading of the judgments remains the same: source expressions are converted into index functions, and the resulting symbolic expressions are rewritten into a form suitable for later property reasoning.

Domains and guarded expressions. The notation $i : 0..e$ denotes the integer range $0, \ldots, e-1$, which corresponds to the implementation notion `Iota` e. A one-dimensional

array is therefore described by a single iterator domain.

The body of an index function is a guarded expression. A term of the form $[p] * e$ means that the expression contributes the value e when the predicate p holds, and contributes nothing otherwise. A full body G is a sum of such guarded alternatives. The guards are assumed to describe disjoint cases over the domain, so for a given valid index at most one guard should hold. In the usual case they also cover the whole domain, meaning that every valid index is described by one of the guarded alternatives. In this way, the body can describe piecewise array contents by splitting the domain into cases.

Examples. As a simple example, the array `iota n`, which is written as $[0, 1, \ldots, n-1]$ is represented by the index function

$$\lambda(i : 0..n).\, i.$$

The domain says that i ranges over $0, \ldots, n-1$, and the body says that the value stored at position i is simply i itself.

A slightly richer example is a conditional array such as

$$\texttt{map}\ (\lambda x.\ \texttt{if}\ x > 0\ \texttt{then}\ 1\ \texttt{else}\ 0)\ xs.$$

Its index function can be written as

$$\lambda(i : 0..n).\, [xs(i) > 0] * 1 + [xs(i) \leq 0] * 0.$$

This example illustrates that index functions describe not only array shapes, but also symbolic array contents. Index functions are built by a single forward traversal of the Futhark AST. Each source construct has a corresponding rule that computes an index function for the whole expression from the index functions of its sub-expressions. These rules, together with a set of normalization rewrites, are what the rest of PropProp and the extensions in this thesis operate on.

Dependence and rewriting. We write $fv(e)$ for the free variables of an expression e, that is, the variables that occur in e and are not bound locally. This is used to express whether one variable depends on another. For example, if $i_1 \notin fv(e_2)$, then e_2 does not depend on i_1.

Several later sections present transformations using inference-rule notation. These rules should be read in the standard way: if the premises above the line hold, then the conclusion below the line may be derived. In this thesis, the premises usually describe when an index-function transformation is valid, and the conclusion describes the rewritten index function or expression that results. This is referred to as a "rewrite rule".

Finally, a substitution such as $e[x := e_y]$ means that every free occurrence of the symbol x in the target expression e is replaced by the expression e_y.

With this notation in place, we can now introduce index functions in the simple one-dimensional case before turning to the segmented representation that is the main topic of this thesis.

4.2 A running one-dimensional example

To illustrate how the framework works on a non-trivial example, we use the function `part2indices`, shown in Figure 6. The function receives a boolean array `conds` and produces two results: a split point and an index array. The split point is the number of `true` values in `conds`. The index array `inds` maps each input position to its target position in a stable two-way partition, where the `true` positions are placed first and the `false` positions are placed afterwards.

Code and concrete execution. Figure 6 shows the source code on the left and a small concrete execution on the right. The postcondition on lines 3–5 states the property that the returned index array must satisfy. It uses `InvFiltPart` to express that `inds` is an inverse filtering/partitioning index array over the range $(0, n)$. The filter predicate is $\lambda\_i \to$ `true`, meaning that no input position is filtered away, and the partition predicate is $\lambda i \to$ `conds[i]`, meaning that the partition is determined by the boolean value at each input position.

```
1  def part2indices [n]
2    (conds: [n]bool)
3    : {(i64, [n]i64) | \(split, inds) ->
4        InvFiltPart inds (0, n)
5          (\_i -> true) (\i -> conds[i]) } =
6    let tflgs = map (\c -> if c then 1 else 0)
     ↪  conds
7    let fflgs = map (\b -> 1 - b) tflgs
8    let indsT = scan (+) 0 tflgs
9    let tmp   = scan (+) 0 fflgs
10   let lst   = if n > 0 then indsT[n-1] else
     ↪  0
11   let indsF = map (\t -> t + lst) tmp
12   let inds  = map3 (\c indT indF ->
13                if c then indT-1 else indF-1)
14              conds indsT indsF
15   in (lst, inds)
```

(a) Futhark source code

$$\texttt{conds} = [\texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}]$$
$$\texttt{tflgs} = [0, 1, 0, 1, 0]$$
$$\texttt{fflgs} = [1, 0, 1, 0, 1]$$
$$\texttt{indsT} = [0, 1, 1, 2, 2]$$
$$\texttt{tmp} = [1, 1, 2, 2, 3]$$
$$\texttt{split} = 2$$
$$\texttt{indsF} = [3, 3, 4, 4, 5]$$
$$\texttt{inds} = [2, 0, 3, 1, 4]$$

The true positions $1, 3$ are mapped to $0, 1$, while the false positions $0, 2, 4$ are mapped to $2, 3, 4$.

(b) Example execution

Figure 6: Source code and a small execution example for `part2indices`.

The computation follows the usual prefix-sum structure for stable partitioning. First, line 6 converts the boolean array into a 0/1 array `tflgs`, where `true` becomes 1 and `false` becomes 0. Line 7 constructs the complementary flag array `fflgs`, which marks the `false` positions instead. Lines 8 and 9 then compute prefix sums over these two flag arrays. The array `indsT` tells, for each position, how many `true` values have appeared up to that point, while `tmp` does the same for the `false` values.

Line 10 computes `lst`, which is the total number of `true` values. This is also the split

point returned by the function. Line 11 shifts the false-side indices by this split point, so that false values are placed after all true values. Finally, lines 12–14 combine the two candidate positions: if `conds[i]` is true, the result uses the true-side index `indT - 1`; otherwise, it uses the false-side index `indF - 1`. The function returns the split point and the final index array on line 15.

The concrete execution in Figure 6 shows the intended behavior. For the input

$$[\texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}],$$

there are two `true` values, so the split point is 2. The two `true` positions are assigned output positions 0 and 1, while the three `false` positions are assigned output positions 2, 3, and 4. Therefore the final index array is

$$[2, 0, 3, 1, 4].$$

4.2.1 Inference rules

We now present the inference rules required to derive an index function for `part2indices`. There are three rules for source constructs (the rules for `map`, `scan`, and conditionals) and two supporting rewrite rules (RecSum and Sub). PropProp's full rule set is much larger, but for this example, these five suffice.

The conditional rule. The first line of `part2indices` maps the lambda `(\c -> if c then 1 else 0)` over `conds`. The lambda's body is a boolean expression, which is converted by the following rule. We present it before the Map rule because the lambda body is what Map ultimately delegates to; understanding the conditional case in isolation makes the subsequent map application straightforward.

Conditional

$$\frac{\Gamma \vdash B \rightsquigarrow \lambda\,() \,.\, p_T \qquad \Gamma \vdash \neg p_T \rightsquigarrow p_F \qquad \Gamma, p_T \vdash E_T \rightsquigarrow (\Gamma', \lambda\,(D) \,.\, e_T) \qquad \Gamma, p_F \vdash E_F \rightsquigarrow (\Gamma'', \lambda\,(D) \,.\, e_F)}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ E_T\ \mathsf{else}\ E_F \to (\Gamma, \lambda\,(D) \,.\, [\,p_T\,] * 1 \cdot e_T + [\,p_F\,] * 1 \cdot e_F)}$$

The first premise derives an index function for the condition B; since the condition is boolean, the body of this index function is a predicate p_T. The second premise rewrites the negation of p_T into a normalized predicate p_F. The third and fourth premises derive index functions for the then- and else-branches in environments extended with the corresponding predicate, so that any indexing inside a branch may use the branch's assumption when bounds-checking. The conclusion combines the two branches under guards.

In the body `if c then 1 else 0` of the first lambda, the condition `c` is the lambda's argument, which the surrounding Map rule (below) will bind to $\lambda\,() \,.\, \mathit{conds}(i)$. The two branches are the scalar constants 1 and 0. The rule then yields the body

$$[\,\mathit{conds}(i)\,] * 1 + [\,\neg\, \mathit{conds}(i)\,] * 0.$$

The `map` rule. With the conditional in hand, we can derive an index function for the whole expression `map (\c -> if c then 1 else 0) conds`. The rule for `map` is:

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

The first premise looks up the index function of the array x_2 being mapped over. The second premise derives an index function for the lambda body E in an environment extended with two things: the lambda's argument x_1 bound to a scalar index function with body e_2 (the body of x_2 index function, looked up in the first premise), and the iterator i's range so that indexing inside the lambda body is bounds-checkable. The conclusion gives the result the same outer domain as x_2 and the freshly derived body e_3.

Applied to `tflgs`, the array argument x_2 is `conds` with index function $\lambda\,(i : 0..n)\ .\ \mathit{conds}(i)$. The lambda's argument `c` is therefore bound to $\lambda\,()\ .\ \mathit{conds}(i)$, and the body is derived using the conditional rule above. This yields

$$\mathit{tflgs} \ \mapsto\ \lambda\,(i : 0..n)\ .\ [\,\mathit{conds}(i)\,] * 1 + [\,\neg\,\mathit{conds}(i)\,] * 0.$$

By the same rule, the index function for `fflgs` is obtained by deriving the body `1 - b` under $b \mapsto \lambda\,()\ .\ [\,\mathit{conds}(i)\,] * 1 + [\,\neg\,\mathit{conds}(i)\,] * 0$ and simplifying via the normalization rewrites of Appendix B.3.5–B.3.6 of [9] (which we use here without further introduction), giving

$$\mathit{fflgs} \ \mapsto\ \lambda\,(i : 0..n)\ .\ [\,\mathit{conds}(i)\,] * 0 + [\,\neg\,\mathit{conds}(i)\,] * 1.$$

The `scan` rule. The third line, `indsT = scan (+) 0 tflgs`, is handled by the rule:

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

The premises mirror those of `map`: the first looks up the array being scanned, and the second derives an index function for the operator body E_1 in an environment where the operator's right argument x_2 is bound to the body of the array's index function. The left argument x_1 is left free in E_1 until the conclusion replaces it with the recurrence marker $\circlearrowleft$. The conclusion's body is piecewise: at $i = 0$ it returns the first element of the scanned array, and at $i \neq 0$ it returns the operator applied to $\circlearrowleft$ (the previous element of the scan) and the current element.

Note that the source-level neutral element E_2 does not appear in the conclusion. This is because it is, by definition, the identity for the operator, so applying the operator to the neutral and the first scanned element yields the first scanned element itself. The base case is therefore just that first element.

Applied to `scan (+) 0 tflgs`, the operator body is $x_1 + x_2$. Under $x_2 \mapsto \lambda\,()\ .\ \mathit{tflgs}(i)$, this derives to $x_1 + \mathit{tflgs}(i)$, and replacing x_1 with $\circlearrowleft$ in the conclusion gives the recurrence form

$$\mathit{indsT} \ \mapsto\ \lambda\,(i : 0..n)\ .\ [\,i = 0\,] * \mathit{tflgs}(i) + [\,i \neq 0\,] * (\circlearrowleft + \mathit{tflgs}(i)).$$

A subsequent rewrite rule, RecSum, closes such recurrences into closed-form sums:

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

The premises require that the recurrence marker $\circlearrowleft$ appears exactly once in the recurrent step and not in the base case or in the summed terms; a fresh summation variable x is introduced. Two distinct sum notations appear in the rule: the outer $\sum_j$ is meta-level, ranging over the syntactic terms t_j that make up the recurrent step, while the inner $\sum_{x=1}^{i}$ is an index-function-level sum symbol denoting the closed-form prefix sum. The substitution $t_j[i := x]$ rewires each recurrent term so that the prefix sum ranges over the new summation variable.

Applied to the recurrence above, with $e_2 = \mathit{tflgs}(i)$ and a single recurrent term $\mathit{tflgs}(i)$, this yields

$$\mathit{indsT} \ \mapsto\ \lambda\,(i : 0..n)\ .\ \mathit{tflgs}(0) + \textstyle\sum_{j=1}^{i}(\mathit{tflgs}(j)) \ =\ \lambda\,(i : 0..n)\ .\ \sum_{j=0}^{i}(\mathit{tflgs}(j)).$$

The substitution rule. The closed form above is still phrased in terms of $\mathit{tflgs}(j)$, but *tflgs* is itself an array with a known index function. A general substitution rule replaces such indexing operations with the body of the indexed array's index function:

$$\textsc{Sub}\ \frac{\Gamma(x) = \lambda\,(i : 0..e_2)\ .\ e_3}{\Gamma \vdash \mathcal{C}\langle x(e_1)\rangle \to \mathcal{C}\langle (e_3[i := e_1])\rangle}$$

The premise looks up x's index function in the environment. The conclusion replaces an occurrence of $x(e_1)$ sitting inside an arbitrary surrounding context $\mathcal{C}$ by the body e_3, with the iterator i substituted by the index expression e_1. Contexts $\mathcal{C}$ are defined formally in Appendix B.3.4 of [9]; for our purposes here it suffices to view $\mathcal{C}$ as "the expression around the indexing operation", kept fixed while only the indexing itself is rewritten.

Applied to $\mathit{tflgs}(j)$ inside the sum of `indsT`, with $\mathit{tflgs} \mapsto \lambda\,(i : 0..n)\ .\ [\mathit{conds}(i)] * 1 + [\neg\,\mathit{conds}(i)] * 0$ derived earlier, the rule replaces $\mathit{tflgs}(j)$ with its guarded body (substituting tflgs's bound iterator with the summation variable j):

$$\mathit{indsT} \ \mapsto\ \lambda\,(i : 0..n)\ .\ \textstyle\sum_{j=0}^{i}([\,\mathit{conds}(j)\,] * 1 + [\,\neg\,\mathit{conds}(j)\,] * 0).$$

The normalization rewrites of Appendix B.3.5–B.3.6 of [9] then drop the 0 guard (its contribution is zero under any predicate), yielding the compact form

$$\mathit{indsT} \ \mapsto\ \lambda\,(i : 0..n)\ .\ \textstyle\sum_{j=0}^{i}([\mathit{conds}(j)]).$$

These five rules, together with the normalization rewrites of Appendix B.3.5–B.3.6 of [9], suffice to derive the remaining lines of `part2indices`. The final inferred index function for `inds` is given in the next subsection.

4.2.2 Final inferred index function

For this example, the most interesting object is the final index function for `inds`. The intermediate arrays are useful operationally, but the final symbolic description already captures the main idea of the computation:

$$[\texttt{for } i < n] \mid \begin{cases} \texttt{conds}(i) \Rightarrow \sum_{j=0}^{i-1} \texttt{conds}(j), \\ \neg\texttt{conds}(i) \Rightarrow i + \sum_{j=i+1}^{n-1} \texttt{conds}(j). \end{cases}$$

This index function has two guarded cases. In the first case, $\texttt{conds}(i)$ is true. The element at position i is therefore placed in the left part of the output. Its target position is the number of earlier true values: $\sum_{j=0}^{i-1} \texttt{conds}(j)$. For example, the first true element is placed at position 0, the second true element at position 1, and so on.

In the second case, $\texttt{conds}(i)$ is false. The element belongs to the right part of the output, after the true elements. Its target position is $i + \sum_{j=i+1}^{n-1} \texttt{conds}(j)$. This expression can be understood as follows. The value i accounts for all elements before position i. Among those earlier elements, the false elements will remain before the current false element in the false partition. The summation then adds the number of true elements that occur after position i. These later true elements must move in front of the current false element, because all true elements are placed before all false elements. Therefore, a false element is shifted to the right by exactly the number of succeeding true elements.

So the index function describes the whole partitioning behavior directly. It tells us, for each input position i, which output position it is mapped to. In particular, it makes the two-way structure of the partition explicit: the first branch maps true elements to the left side, and the second branch maps false elements to the right side while preserving their relative order.

4.2.3 Transition to the segmented case

The example above is still entirely one-dimensional. All arrays have the simple domain

$$[\texttt{for } i < n],$$

and all iterator arithmetic is expressed directly in terms of a single index i. This is the easiest setting in which to understand index functions.

The main topic of this thesis begins when this simple picture no longer holds, that is, when one flat array represents several logical segments of different lengths. In that setting, the framework must describe not only values, but also how a flat index corresponds to a segment and an offset inside that segment. The next section introduces this segmented and flattened view.

5 Framework Overview for Segmented and Flattened Domains

This section introduces the part of the framework that is most relevant for this thesis: segmented domains, flattened domains, and the rules that recover and propagate segmented structure in flattened irregular programs. In the previous section, we only needed ordinary one-dimensional index functions. Here, we extend that picture to the case where one flat array may represent several logical segments, possibly of different lengths. This is the setting used by PropProp for flattened irregular programs [9].

5.1 Notation and grammar for the segmented case

We reuse the one-dimensional notation from Section 4.1 with one addition, summarized in Table 2: a domain may now contain a flattened component, written using the symbol $\times$. We also unify the substitution notation with the inference rules used later in the section.

Notation	Meaning
$e[x := e_y]$	substituting symbol x with expression e_y in e
$i : 0..e \times D$	flattened-domain case (one flat dimension, multiple iterators)

Table 2: New notation used in the flattened framework overview.

Extended grammar. For the segmented case, the grammar of domains must be extended so that one domain component may contain more than one iterator. Following the multi-dimensional extension in [9], we use the grammar presented in Figure 7, where the new case $i : 0..e \times D$ is used for flattened domains.

$$
\begin{aligned}
p &::= \texttt{true} \mid \texttt{false} \mid e \leq e \mid \neg p \mid p \wedge p \mid p \vee p \mid \cdots \\
t &::= n \mid s \mid s \cdot t \\
e &::= t \mid t + e \\
D &::= i : 0..e \mid i : 0..e, D \mid i : 0..e \times D \\
a &::= [p] * e \\
G &::= a \mid a + G \\
f &::= \lambda(D).\, G
\end{aligned}
$$

Figure 7: Grammar used for multidimensional index functions.

Ordinary dimensions versus flattened dimensions. We distinguish between ordinary multi-dimensional domains and flattened domains. A domain of the form $i_1 : 0..e_1,\ i_2 : 0..e_2$ denotes two ordinary dimensions. In contrast, a domain of the form

$i_1 : 0..e_1 \times i_2 : 0..e_2$ denotes a single flattened dimension represented using two iterators. This distinction matters in the rest of the thesis. Segmented arrays are not represented as ordinary rank-2 arrays, but as flat arrays whose internal segmented structure is recorded through flattened domains.

Shorthand for indexing 1D arrays in flattened coordinates. When an array `a` declared with a flat 1D type is indexed inside a flattened domain $i_1 : 0..e_2 \times i_2 : 0..e_3$, we write $\mathtt{a}(i_1, i_2)$ as shorthand for `a` at the flat position $\sum_{j=0}^{i_1-1} e_3[i_1 := j] + i_2$. We also use this shorthand inside 1D index functions when an iterator variable plays the role of i_1. The array itself remains one-dimensional; the two-iterator notation only reflects how the surrounding flattened domain interprets the flat index. The rule PropFlatten (Section 5.4.2) is what makes this rewriting valid.

Why flattened domains are needed. The central problem this thesis addresses is recovering segmented structure from programs that, on the surface, manipulate only flat one-dimensional arrays. When a program builds a flat irregular array, the connection between the flat layout and the underlying segmentation is encoded indirectly through auxiliary arrays such as shapes, offsets, or flag arrays. The index-function layer makes this structure explicit by recording it in the domain itself, so that later rules can reason about segment starts, segment lengths, and per-segment offsets symbolically. Flattened domains are how this structure is recorded. The source language also supports `flatten`, which collapses two dimensions into one, and the same domain representation handles that case as a byproduct.

5.2 Segmented domains: old and new representations

The distinction between ordinary multi-dimensional domains and flattened domains becomes especially relevant in the segmented case. In the simple one-dimensional setting, a domain such as $i : 0..n$ is enough to describe the valid indices of an array. For segmented arrays, however, one flat array may represent several logical rows of different lengths, and this additional structure must also be represented in the index function.

In the older implementation, segmented flat domains were represented indirectly through the constructor `Cat`. The `Cat` constructor describes a concatenation of consecutive intervals: it stores the segment boundary expression directly and represents the flat domain as the disjoint union of those intervals. If the segment boundaries are given by a monotone boundary expression $e(k)$, then the corresponding flat segmented domain can be understood as

$$\biguplus_{k=0}^{m-1} [e(k), e(k+1)),$$

where $\biguplus$ denotes the disjoint union of intervals. Here $e(k)$ is the start of segment k, and $e(k+1) - e(k)$ is the length of that segment. In this representation, the segmented structure is encoded through absolute flat boundaries. The segment iterator is implicit, and the offset inside a segment is recovered indirectly: given a flat index i, one first

locates the boundary $e(k)$ such that $e(k) \leq i < e(k+1)$, and then computes the offset as $i - e(k)$.

In the refactored representation used in this thesis, the same segmented structure is represented directly as one flattened dimension with two iterators:

$$k : 0..m \times i : 0..(e(k+1) - e(k)).$$

The outer iterator k identifies the segment, while the inner iterator i identifies the offset inside that segment. This is still one flat dimension represented using two iterators, not an ordinary rank-2 array. This is the same distinction introduced in the previous subsection: a domain of the form $i_1 : 0..e_1,\ i_2 : 0..e_2$ denotes two ordinary dimensions, while $i_1 : 0..e_1 \times i_2 : 0..e_2$ denotes one flattened dimension with two iterator variables.

The old and new representations describe the same flat positions. The correspondence is given by the mapping

$$(k, i) \mapsto e(k) + i, \qquad 0 \leq i < e(k+1) - e(k).$$

So the new representation does not change the meaning of the domain. It only changes how that meaning is expressed. The old representation starts from absolute flat intervals, while the new one starts from a segment number k and an offset i inside that segment.

We show a small example to make this more clear. Suppose the segment boundaries are $e = [0, 2, 5, 6]$. The boundary array has length $m + 1$: one entry per segment plus a final endpoint. This splits the flat domain into three segments of lengths 2, 3, and 1, which corresponds to a shape array $\texttt{shape} = [2, 3, 1]$ of length $m = 3$, since $\texttt{shape}(k) = e(k+1) - e(k)$. The old and new views of this same domain can be shown side by side as follows.

Old representation	**New representation**
$[0, 2) \uplus [2, 5) \uplus [5, 6)$	$k : 0..3 \times i : 0..(e(k+1) - e(k))$
Segment boundaries:	Iterator pairs by segment:
$e(0) = 0$	$k = 0 :\ i \in \{0, 1\}$
$e(1) = 2$	$k = 1 :\ i \in \{0, 1, 2\}$
$e(2) = 5$	$k = 2 :\ i \in \{0\}$
$e(3) = 6$	
The segmented structure is described indirectly through the flat interval boundaries.	The segmented structure is described directly through the segment iterator k and the local offset i.

Figure 8: Old and new representations of the same segmented flat domain.

For example, in the new representation the pair $(k, i) = (1, 2)$ corresponds to the flat position $e(1) + 2 = 2 + 2 = 4$. This is the same position that belongs to the interval $[2, 5)$ (the one indexed by $k = 1$) in the old representation. Both encodings describe the same flat element; they expose different structure.

In the rest of the thesis, the boundary expression $e(k)$ is concretely the prefix sum $\sum_{j=0}^{k-1} \texttt{shape}(j)$, so that $e(k+1) - e(k) = \texttt{shape}(k)$. With this identification, the new representation simplifies to $k : 0..m \times i : 0..\texttt{shape}(k)$, which is the form used from Section 5.3 onwards.

5.3 A simple segmented example: `mk_flag_array`

To make the segmented case concrete, we use the Futhark program `mk_flag_array` as a running example. This example is small, but it already shows the main point of the refactoring: the old and new implementations infer the same segmented structure, but express it in different ways.

The starting point is the usual flat representation of an irregular array. Instead of storing a jagged array directly, we store its contents in one flat array together with an auxiliary shape array[1]. If the shape array is $\texttt{shape} = [s_0, s_1, \ldots, s_{m-1}]$, then segment k has length s_k, and the total flat length is $\sum_{k=0}^{m-1} s_k$. So the shape array tells us how the flat array should be split into logical segments.

Many flattened irregular programs need exactly this information: once segment starts are known, later computations can derive other auxiliary arrays, such as segment identifiers, or apply segmented scans, which we introduce in Section 5.4 when they first appear in the derivation.

`mk_flag_array` takes:

- a default value `zero`,
- a shape array `shape`, where `shape`$[k]$ is the length of segment k,
- a value array `xs` of length m,

and constructs a flat array `flags`[2] whose total length is the sum of the segment lengths. For each segment, the first position stores the corresponding value from `xs`, while all remaining positions store `zero`.

Code and concrete execution. This function is shown in Figure 9. The left side contains the source code, while the right side shows a small concrete execution.

[1] We write `shape` in monospace or as a source variable when referring to the program array of segment lengths. By contrast, `shape(f)` denotes the domain stored in an index function.

[2] We use the term *flag array* loosely: it is any flat array where each segment start carries a distinguishing value supplied by the caller via `xs`, with the default value at every other position. The canonical Boolean flag array, which stores `true` at each segment start and `false` elsewhere, arises when `xs` is a constant `true` array. Whatever the values, a flag array of length n split into m segments has length n and contains exactly one non-default entry per non-empty segment.

```
def mk_flag_array 't [m]
        (zero: t)
        (shape: {[m]i64 | \x -> Range x (0,
        ↪  inf) })
        (xs: [m]t) : {(i64, []t) |
          \(_, flags) -> length flags ==
          ↪  sum_i64 shape} =
  let shp_rot = map (\i -> if i==0 then 0i64
  ↪  else shape[i-1]) (iota m)
  let shp_scn = scan (\x y -> x + y) 0i64
  ↪  shp_rot
  let aoa_len = if m > 0 then shp_scn[m-1] +
  ↪  shape[m-1] else 0
  let shp_ind =
        map2 (\shp ind ->
                if shp <= 0i64 then -1i64
                ↪  else ind
              ) shape shp_scn
  let zeros = replicate aoa_len zero
  let flags = scatter zeros shp_ind xs
  in (aoa_len, flags)
```

(a) Futhark source code

$$\begin{aligned}
\texttt{shape} &= [2,0,3] \\
\texttt{xs} &= [10,20,30] \\
\texttt{shp_rot} &= [0,2,0] \\
\texttt{shp_scn} &= [0,2,2] \\
\texttt{aoa_len} &= 5 \\
\texttt{shp_ind} &= [0,-1,2] \\
\texttt{zeros} &= [0,0,0,0,0] \\
\texttt{flags} &= [10,0,30,0,0]
\end{aligned}$$

Segment 1 is empty, so its index in `shp_ind` is -1, which `scatter` treats as out of bounds and ignores. The starts of the two non-empty segments are the flat positions 0 and 2.

(b) Example execution

Figure 9: Source code and a small execution example for `mk_flag_array`.

The concrete execution shows what the program is doing. The shape array $[2,0,3]$ means that the result should contain three segments: the first of length 2, the second empty (length 0), and the third of length 3. The array `shp_rot` stores the previous segment's length at each position, so that scanning it produces the segment start offsets. In this example those starting positions are $[0,2]$, while the empty segment has no start at all. The `map2` on `shape` and `shp_scn` writes -1 in `shp_ind` at the position of the empty segment. This is the standard for handling empty segments in a flat scatter: the -1 is out of bounds for `zeros`, so `scatter` drops the corresponding write. Note also that `aoa_len` is computed as $\texttt{shp\_scn}[m-1]+\texttt{shape}[m-1]$, which equals the sum of the segment lengths. Starting from an array filled with the default value `zero`, the program writes the values from `xs` at the segment starts of the non-empty segments, giving the final result $[10,0,30,0,0]$.

This example is suitable for the present discussion because the segmented structure is simple to see both operationally and symbolically. Operationally, the program computes segment starts and scatters values to them. Symbolically, the inferred index function describes the same idea: one case for the start of a segment, and one case for the interior of a segment.

5.3.1 Old and new inferred index functions

This example is easy to understand in both representations.

In the older representation, the segmented flat domain is described indirectly through a `Cat`-style concatenation of intervals. In mathematical notation, the inferred index function for `flags` has the form

$$\lambda\left(i_{\text{flat}} \in \biguplus_{k=0}^{m-1} [e(k), e(k+1))\right) . \begin{cases} i_{\text{flat}} = e(k) \Rightarrow xs(k), \\ i_{\text{flat}} \neq e(k) \Rightarrow zero, \end{cases}$$

where

$$e(k) = \sum_{j=0}^{k-1} \texttt{shape}(j).$$

So the old representation talks directly about absolute flat positions. An element is the first element of a segment when its flat index is equal to the boundary $e(k)$.

In the new representation, the same index function is written using one flattened dimension with two iterators:

$$\lambda(k : 0..m \times i : 0..\texttt{shape}(k)) . \begin{cases} i = 0 \Rightarrow xs(k), \\ i \neq 0 \Rightarrow zero. \end{cases}$$

This form describes the same array, but now the segmented structure is explicit. The outer iterator k says which segment we are in, and the inner iterator i says where we are inside that segment.

We can verify this by walking through the concrete example with $\texttt{shape} = [2, 0, 3]$ and $\texttt{xs} = [10, 20, 30]$. At $(k, i) = (0, 0)$ the body returns $xs(0) = 10$; at $(0, 1)$ it returns $zero = 0$. For $k = 1$ the inner range $i : 0..0$ is empty, so segment 1 contributes nothing to the domain. At $(2, 0)$ the body returns $xs(2) = 30$; at $(2, 1)$ and $(2, 2)$ it returns $zero = 0$. The values, listed in domain order, are $[10, 0, 30, 0, 0]$, matching the concrete output.

Empty segments. Empty segments are handled uniformly in the new representation. When $\texttt{shape}(k) = 0$, the inner iterator range $i : 0..\texttt{shape}(k)$ is empty, so segment k simply contributes no positions to the index function's domain. No special case is needed in the body. In the old representation, the corresponding interval $[e(k), e(k+1))$ is also empty when $e(k+1) = e(k)$, but recognizing emptiness requires the solver to detect equality between two symbolic boundary expressions, which is less direct than checking that an iterator range is empty.

Why these two forms mean the same thing. The two index functions are equivalent because they describe the same flat positions, via the correspondence $(k, i) \mapsto e(k)+i$ with $0 \leq i < \texttt{shape}(k)$. Under this correspondence, the condition $i = 0$ in the new representation corresponds to $i_{\text{flat}} = e(k)$ in the old, so both representations express the same segment-start guard, just relative to different iterator variables.

5.4 A running segmented example: `part2indicesL`

To illustrate how segmented index functions arise on a non-trivial program, we use the function `part2indicesL`, the segmented analogue of `part2indices` from Section 4.2.1. Given a shape array `shape` of segment lengths and a flat boolean array `csL` of length $\sum_i \texttt{shape}(i)$, it returns a flat array `inds` that independently partitions each segment of `csL` according to its boolean values, placing the `true` positions first within each segment.

Recall from the previous subsection that `mk_flag_array` produces a flag array marking the start of each segment in a flat irregular array. `part2indicesL` uses this construction as a building block, via the auxiliary function `segment_ids`. The segmented structure that `mk_flag_array`'s inferred index function makes explicit is the same structure that `part2indicesL`'s intermediate arrays inherit.

The `segment_ids` function. `segment_ids` takes a shape array and returns a pair `(seg_ids, flags)`, where:

- `seg_ids` is a flat array of segment numbers, e.g. $[0, 0, 1, 1, 1, 2]$ for shape $[2, 3, 1]$;
- `flags` is a Boolean flag array marking segment starts, e.g. $[\texttt{T}, \texttt{F}, \texttt{T}, \texttt{F}, \texttt{F}, \texttt{T}]$ for the same shape.

It is built from `mk_flag_array` and a segmented scan. We treat `segment_ids` as given here; its derivation is in Appendix D.1 of [9].

Outline of the partitioning. `part2indicesL` mirrors `part2indices` at the segment level:

1. it converts `csL` into a 0/1 array for the true branch and its complement for the false branch,
2. it segmented-scans both arrays to obtain per-segment offsets within each branch,
3. it computes, for each segment, the total number of `true` values, used to offset the false branch within that segment,
4. it computes, for each flat position, the start of the segment it belongs to,
5. it combines per-segment offsets and segment starts to compute the final target index in the flat output.

The result is a permutation of $0, \ldots, n-1$ that, when used as a scatter index, partitions each segment of `csL` independently. The postcondition expresses this. Reading it from the outside in: `For inds (\k -> ...)` lifts the inner property over each segment k (the `For` combinator is the source-level construct introduced in Section 3.2); within segment k, `InvFiltPart` states that the slice of `inds` from `starts[k]` to `starts[k+1]` is the inverse of a filtering partition of that slice by `csL`. The `starts` array is built from the returned `ends`: it has length $m+1$, with $\texttt{starts}[0] = 0$ and $\texttt{starts}[k] = \texttt{ends}[k-1]$ for $k \geq 1$, so that $\texttt{starts}[k+1] - \texttt{starts}[k] = \texttt{shape}[k]$. The postcondition uses this to slice `inds` segment by segment.

Auxiliary functions: segmented scan. A *segmented scan* with operator $\oplus$ has the same effect as scanning each segment of the input independently with $\oplus$. It is implemented as an ordinary `scan` with a *lifted* operator applied to the array obtained by zipping the flag array with the data array: the lifted operator inspects the flag component of its right argument, resetting when the flag is set, and combining with $\oplus$ otherwise.

`part2indicesL` uses the segmented building block `sgm_sum flags arr`, a segmented prefix sum over `arr` that resets at each segment boundary. It is built from an ordinary `scan` with a lifted operator over the zipped pair, and inherits the segmented structure of `flags` in its inferred index function. We treat it as given here; its derivation is in Appendix D.1 of [9].

Code and concrete execution. The function is shown in Figure 10. The left side contains the source code, while the right side shows a small concrete execution with three segments of lengths $[2, 3, 1]$ and an alternating boolean array.

```
def part2indicesL [m][n]
    (shape: {[m]i64 | \x -> Range x (0,inf)})
    (csL: {[n]bool | \_ -> n == sum shape})
    : {([n]i64, [m]i64, [n]i64, [m]i64)
       | \(inds, ends, _, _) ->
           let starts =
             map (\k -> if k==0 then 0 else ends[k-1])
                 (iota (m+1))
           in For inds (\k ->
                InvFiltPart inds
                  (starts[k], starts[k+1])
                  (\_i -> true) (\i -> csL[i]))} =
  let (seg_ids, flags) = segment_ids shape
  let tflgs = map (\c -> if c then 1 else 0) csL
  let fflgs = map (\b -> 1 - b) tflgs
  let indsT = sgm_sum flags tflgs
  let tmp   = sgm_sum flags fflgs
  let ends  = scan (+) 0 shape
  let lst   = map2 (\s b -> if s == 0 then -1
                            else indsT[b-1])
                   shape ends
  let indsF = map2 (\t k -> t + lst[k]) tmp seg_ids
  let offs  = map (\k -> if k > 0 then ends[k-1]
                         else 0) seg_ids
  let inds  = map4 (\c iT iF o ->
                      if c then o + iT - 1
                           else o + iF - 1)
                   csL indsT indsF offs
  in (inds, ends, seg_ids, lst)
```

(a) Futhark source code

$$
\begin{aligned}
\texttt{shape} &= [2,3,1] \\
\texttt{csL} &= [F,T,F,T,F,T] \\
\texttt{flags} &= [T,F,T,F,F,T] \\
\texttt{seg\_ids} &= [0,0,1,1,1,2] \\
\texttt{tflgs} &= [0,1,0,1,0,1] \\
\texttt{fflgs} &= [1,0,1,0,1,0] \\
\texttt{indsT} &= [0,1,0,1,1,1] \\
\texttt{tmp} &= [1,1,1,1,2,0] \\
\texttt{ends} &= [2,5,6] \\
\texttt{lst} &= [1,1,1] \\
\texttt{indsF} &= [2,2,2,2,3,1] \\
\texttt{offs} &= [0,0,2,2,2,5] \\
\texttt{inds} &= [1,0,3,2,4,5]
\end{aligned}
$$

Within each segment, the `true` positions are mapped to the leftmost output indices of that segment and the `false` positions follow.

(b) Example execution

Figure 10: Source code and a small execution example for `part2indicesL`.

The concrete run shows the intended meaning of the function. Segment 0 occupies flat positions 0–1 of the output. Its element at `csL` position 0 (which is `false`) is sent to flat position 1; its element at position 1 (which is `true`) is sent to flat position 0. Segment 1 occupies flat positions 2–4. Its only `true` element, at `csL` position 3, is sent to flat position 2; the two `false` elements at positions 2 and 4 are sent to flat positions 3 and 4 respectively, preserving their relative order. Segment 2, of length 1, is left in place. Compared with the one-dimensional `part2indices`, the same partitioning logic now operates inside each segment of the flat input, and the segment iterator that selects which segment we are operating in is not visible in the source program at all.

Inferred index function. The challenge for the framework is to recover this segmented structure from a program that, on the surface, only manipulates flat one-dimensional arrays. Concretely, the inferred index function for `inds` should expose two iterators i_1, i_2 where i_1 ranges over segments and i_2 ranges over offsets within a segment, even though no such pair appears in the source code. The next subsection states this index function explicitly, and the subsection after derives it from `part2indicesL` using the segmented inference rules.

5.4.1 Inferred index function for `inds`

Section 5.3 showed the old and new representations side by side on `mk_flag_array`. The same contrast applies to `part2indicesL`, only at a larger scale: the `Cat`-based form would describe the flat dimension of `inds` as a concatenation of intervals $[e(k), e(k+1))$ with the segment number recovered indirectly from the flat position, while the flattened form makes the segment iterator explicit in the domain. We state only the flattened form here, since it is the target of the derivation in the next subsection.

Convention for `ends`. Let `ends` denote the inclusive prefix sum of `shape`, so $\texttt{ends}(k) = \sum_{j=0}^{k} \texttt{shape}(j)$. It is the inclusive counterpart of the boundary expression $e(k)$ from Section 5.2: since $e(k) = \sum_{j=0}^{k-1} \texttt{shape}(j)$, we have $e(k) = \texttt{ends}(k-1)$ and $e(k+1) = \texttt{ends}(k)$, so the segment start written $e(i_1)$ in Section 5.2, and in the scatter rule of Section 6.2, is exactly $\texttt{ends}(i_1 - 1)$ here. We adopt the convention $\texttt{ends}(-1) = 0$, so that $\texttt{ends}(i_1 - 1)$ is well-defined at $i_1 = 0$ and gives the start of the first segment (which is 0). Under this convention, $\texttt{ends}(i_1 - 1) = \sum_{j=0}^{i_1-1} \texttt{shape}(j)$ is the start of segment i_1 in the flat output, and the shorthand $\texttt{a}(i_1, i_2)$ introduced in Section 5 equals $\texttt{a}(\texttt{ends}(i_1 - 1) + i_2)$.

The flat boolean array `csL` has length $n = \sum_{j=0}^{m-1} \texttt{shape}(j) = \texttt{ends}(m-1)$. The

inferred index function for `inds` is then

$$\texttt{inds} \mapsto \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).$$
$$\begin{cases} \texttt{csL}(i_1, i_2) \Rightarrow \texttt{ends}(i_1-1) + \sum_{j=0}^{i_2-1} [\texttt{csL}(i_1, j)], \\ \neg\, \texttt{csL}(i_1, i_2) \Rightarrow \texttt{ends}(i_1-1) + i_2 + \sum_{j=i_2+1}^{\texttt{shape}(i_1)-1} [\texttt{csL}(i_1, j)]. \end{cases}$$

The two cases mirror the 1D `part2indices` index function from Section 4.2.1, but now describe behavior within segment i_1 rather than globally. The segment start $\texttt{ends}(i_1-1)$ is the constant offset of segment i_1 in the flat output. The per-segment prefix sum $\sum_{j=0}^{i_2-1}[\texttt{csL}(i_1, j)]$ in the true case and the per-segment suffix sum $\sum_{j=i_2+1}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1, j)]$ in the false case play the roles their global counterparts did in the 1D case, restricted to segment i_1 via the explicit dependence on i_1, with $\texttt{shape}(i_1)$ playing the role of n.

Take $\texttt{shape} = [2, 3, 1]$ and the values from Figure 10, and consider $(i_1, i_2) = (1, 1)$. The flat position is $\texttt{ends}(0) + 1 = 2 + 1 = 3$. Here $\texttt{csL}(1, 1)$ is $\texttt{csL}[3] = \texttt{true}$, so the first case applies. We have $\texttt{ends}(0) = 2$ and the per-segment prefix sum $\sum_{j=0}^{0}[\texttt{csL}(1, j)] = [\texttt{csL}(1, 0)] = [\texttt{false}] = 0$. So the value is $2 + 0 = 2$, matching $\texttt{inds}[3] = 2$ in the execution. The next subsection derives this index function step by step from `part2indicesL`.

5.4.2 Inference rules

We now present the inference rules required to derive the segmented index function for `inds` shown in Section 5.4.1.

Derivation roadmap. The derivation proceeds in three phases. *Phase 1* derives `tflgs`, `fflgs`, and `ends` using only the 1D rules of Section 4.2.1; these intermediates reference only 1D arrays and stay 1D throughout. *Phase 2* lifts the segmented scans `indsT` and `tmp` into flattened domains, so that their recurrence is recognized as per-segment rather than global. This is where the segmented machinery first fires. *Phase 3* combines the now-flattened scans with the 1D arrays from Phase 1 to derive the remaining intermediates `lst`, `offs`, and `indsF`, and finally assembles `inds`. Most intermediates lift to flattened domains; the exception is `lst`, which has one entry per segment and stays 1D for a size reason explained when it is derived.

Starting point. Two auxiliary arrays from `segment_ids` arrive with flattened index functions already inferred (their derivation is given in Appendix D.1 of [9]):

$$\texttt{flags} \mapsto \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\ [\,i_2 = 0\,] * \text{true} + [\,i_2 \neq 0\,] * \text{false}$$
$$\texttt{seg\_ids} \mapsto \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\ [\,\text{true}\,] * i_1.$$

We take these as given. Recall that `shape` is a free symbol and that the environment carries $\mathsf{Range}\ s\ 0..\infty$, so $\texttt{shape}(i_1) \geq 0$ for all i_1. The array `flags` is Boolean, so its body acts as a predicate in any context where a Boolean value is expected.

These two arrays are the only ones in `part2indicesL` that begin life with a flattened domain. `csL` and `shape` are formal arguments and remain one-dimensional throughout the analysis: their preconditions ($n = \sum_j \texttt{shape}(j)$ and $\mathsf{Range}\ s\ 0..\infty$) record size and range facts in the environment, but do not promote their index functions to flattened form. Every other intermediate in the function starts with an ordinary 1D outer domain produced by the segmented-aware Map and Scan rules, either $i_1 : 0..n$ or $i_1 : 0..m$ depending on which input array its size is inherited from.

The fate of each intermediate is as follows. `tflgs`, `fflgs`, and `ends` stay 1D throughout, because their bodies reference only 1D arrays. `lst` is more subtle: its body indexes into `indsT`, which later gets a flattened domain, but PropFlatten's size-equality premise (explained later) fails on `lst`. The flat domain of `lst` has size m, while `indsT`'s flattened domain has total size $\sum_j \texttt{shape}(j) = n$. The two sizes are not equal, so the rule cannot propagate `indsT`'s flattened domain into `lst`, and `lst` keeps its 1D domain $k : 0..m$. The four intermediates that do receive a flattened domain are, in derivation order, `indsT`, `tmp`, `indsF`, and `offs`. They each reference a flattened array in their body (`flags` in `indsT` and `tmp`, `seg_ids` in `offs`, and `tmp` in `indsF`), and for each one PropFlatten's size-equality premise does hold. The segmented machinery first fires at `indsT` and `tmp`, where lifting is also a prerequisite for SegRecSum to recognize the segmented scan's recurrence on $\circlearrowleft$ as per-segment rather than global.

Phase 1: 1D intermediates.

1D derivations. Before reaching the segmented scans, `tflgs`, `fflgs`, and `ends` are all derived using the 1D Map, Scan, Conditional, and Sub rules from Section 4.2.1. The segmented-aware versions of these rules reduce to their 1D counterparts when no flattened-domain array appears in the body, so we use the 1D forms here; the complete set is given in Appendix B of [9].

The derivation of `tflgs = map (\c -> if c then 1 else 0) csL` mirrors the corresponding step in `part2indices` from Section 4.2.1. The Map rule binds the lambda's argument `c` to $\texttt{csL}(j)$, and the Conditional rule splits on this Boolean, yielding

$$\texttt{tflgs} \ \mapsto\ \lambda(j : 0..n).\ [\texttt{csL}(j)] * 1 + [\neg\, \texttt{csL}(j)] * 0.$$

`fflgs = map (\b -> 1 - b) tflgs` is derived by the Map rule, inlining `tflgs`'s body via Sub and simplifying the arithmetic:

$$\texttt{fflgs} \ \mapsto\ \lambda(j : 0..n).\ [\texttt{csL}(j)] * 0 + [\neg\, \texttt{csL}(j)] * 1.$$

Finally, `ends = scan (+) 0 shape` is derived by the Scan rule, which produces a recurrence on $\circlearrowleft$, followed by RecSum, which closes it into a prefix sum:

$$\texttt{ends} \ \mapsto\ \lambda(k : 0..m).\ \sum_{j=0}^{k} \texttt{shape}(j).$$

This matches the definition $\texttt{ends}(k) = \sum_{j=0}^{k} \texttt{shape}(j)$ adopted in Section 5.4.1.

Phase 2: lifting the segmented scans.

The segmented scans: `indsT` and `tmp`. The flattened domain first reaches an intermediate's body at `indsT = sgm_sum flags tflgs`. `sgm_sum` is implemented as an ordinary `scan` with a lifted operator that resets at segment boundaries, and is given by the summary

$$\texttt{sgm\_sum flags xs} \mapsto \lambda(i_0 : 0..|\texttt{flags}|).$$
$$[\,i_0 = 0 \vee \texttt{flags}(i_0)\,] * \texttt{xs}(i_0) + [\,i_0 \neq 0 \wedge \neg\texttt{flags}(i_0)\,] * (\circlearrowleft + \texttt{xs}(i_0))$$

where $\circlearrowleft$ is the recurrence marker. The references $\texttt{flags}(i_0)$ and $\neg\,\texttt{flags}(i_0)$ are used directly inside guards because `flags` has Boolean type; PropProp treats a Boolean-valued expression as a predicate in any context where a predicate is expected. When this summary is instantiated for `indsT`, the formal parameters `flags` and `xs` are bound to the actual arguments `flags` and `tflgs`, giving

$$\texttt{indsT} \mapsto \lambda(i_0 : 0..n).$$
$$[\,i_0 = 0 \vee \texttt{flags}(i_0)\,] * \texttt{tflgs}(i_0) + [\,i_0 \neq 0 \wedge \neg\texttt{flags}(i_0)\,] * (\circlearrowleft + \texttt{tflgs}(i_0))$$

and analogously for `tmp = sgm_sum flags fflgs`:

$$\texttt{tmp} \mapsto \lambda(i_0 : 0..n).$$
$$[\,i_0 = 0 \vee \texttt{flags}(i_0)\,] * \texttt{fflgs}(i_0) + [\,i_0 \neq 0 \wedge \neg\texttt{flags}(i_0)\,] * (\circlearrowleft + \texttt{fflgs}(i_0)).$$

Both intermediates initially have a 1D domain $i_0 : 0..n$, inherited from `sgm_sum`'s 1D output type. Their bodies, however, reference $\texttt{flags}(i_0)$, and `flags` has a flattened domain. The framework needs a rule that can transport this flattened domain into `indsT` and `tmp`, so that the recurrence on $\circlearrowleft$ can later be recognized as a per-segment scan rather than a global one.

The PropFlatten rule. PropFlatten is the rule that propagates a flattened domain from one index function to another that still views the same dimension as a single flat iterator. It applies precisely when a 1D index function references, in its body, an array with a flattened domain. The rule is:

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

The first premise looks up an index function with an existing flattened domain $i_2 : 0..e_2 \times i_3 : 0..e_3$; this index function must be referenced by name in the body of the function being rewritten, which is how x is bound. The second and third premises compute the prefix-sum row start $e_{row} = \sum_{j=0}^{i_2-1} e_3[i_2 := j]$, which is the start of segment i_2 when the inner bound depends on the outer iterator, and collapses to the multiplication

$i_2 \cdot e_3$ when it does not. The fourth premise asks the algebra layer to verify that the size e_1 of the flat one-iterator domain matches the total size $e_{row}[i_2 := e_2]$ of the flattened domain. If the query succeeds, the conclusion rewrites the flat iterator i_1 as $e_{row} + i_3$ throughout the body and adopts the flattened domain in place of the flat one.

This is the rule that bridges 1D and segmented reasoning. The regular case, where e_3 does not depend on i_2, is what the implementation handled before this thesis: there the row start collapses to $i_2 \cdot e_3$ and the size check becomes $e_1 = e_2 \cdot e_3$. The irregular case, where $e_3 = \texttt{shape}(i_2)$ depends on the outer iterator, is the generalization implemented in this thesis as the new top-level rule `propFlattenOnce` (Section 6.1).

Applied to `indsT`, PropFlatten uses `flags`'s domain $i_2 : 0..m \times i_3 : 0..\texttt{shape}(i_2)$. The row start computed in the second premise is $e_{row} = \sum_{j=0}^{i_2-1} \texttt{shape}(j)$, and the size check $|\texttt{flags}| = \sum_{j=0}^{m-1} \texttt{shape}(j)$ holds. The conclusion rewrites i_0 as $e_{row}+i_3$ throughout the body and adopts the flattened domain. The rule introduces the fresh iterators i_2, i_3; for consistency with the surrounding notation we rename them to i_1, i_2 so that i_1 ranges over segments and i_2 over offsets. After this renaming, $e_{row} = \sum_{j=0}^{i_1-1} \texttt{shape}(j)$, and the body's flat references take the form $\texttt{flags}(e_{row} + i_2)$ and $\texttt{tflgs}(e_{row} + i_2)$.

The next step is to inline the bodies of `flags` and `tflgs` via Sub. Substituting `flags`'s body interpreted at flattened position (i_1, i_2) gives $\texttt{flags}(e_{row} + i_2) = (i_2 = 0)$ as a Boolean. The guard $[i_0 = 0 \vee \texttt{flags}(i_0)]$ becomes

$$[(e_{row} + i_2) = 0 \ \vee \ (i_2 = 0)].$$

Since $e_{row} \geq 0$ and $i_2 \geq 0$, the equation $e_{row} + i_2 = 0$ implies $i_2 = 0$, so the disjunction simplifies to $[i_2 = 0]$. The complementary guard $[i_0 \neq 0 \wedge \neg\, \texttt{flags}(i_0)]$ simplifies to $[i_2 \neq 0]$ by the same reasoning.

For the value expressions, the reference $\texttt{xs}(i_0)$ (bound to `tflgs` by parameter-binding substitution from the `sgm_sum` summary, not by the Sub rule) becomes $\texttt{tflgs}(e_{row} + i_2)$. The Sub rule then inlines `tflgs`'s body from Phase 1. Using the shorthand notation $\texttt{csL}(e_{row} + i_2) = \texttt{csL}(i_1, i_2)$, the reference $\texttt{tflgs}(e_{row} + i_2)$ becomes $[\texttt{csL}(i_1, i_2)] * 1 + [\neg\, \texttt{csL}(i_1, i_2)] * 0$. Note that `csL` itself remains 1D; only its indexing argument is now expressed in flattened coordinates. We write this in shorthand as $\texttt{tflgs}(i_1, i_2)$. After all these substitutions and simplifications, `indsT`'s body is in segmented-scan form:

$$\texttt{indsT} \ \mapsto \lambda\big(i_1 : 0..m \ \times \ i_2 : 0..\texttt{shape}(i_1)\big).$$
$$[\,i_2 = 0\,] * \texttt{tflgs}(i_1, i_2) + [\,i_2 \neq 0\,] * (\circlearrowleft + \texttt{tflgs}(i_1, i_2)).$$

The same chain of rewrites applies to `tmp`, yielding the same form with `fflgs` in place of `tflgs`.

The SegRecSum rule. The pattern above, a per-segment recurrence over the inner iterator, is what SegRecSum recognizes:

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

The premises require that $\circlearrowleft$ appears exactly once in the recurrent case and not in the base case or in the summed terms; a fresh summation variable x is introduced. The conclusion replaces the base case with $e_3[i_2 := 0]$ and the recurrent case with a closed-form sum over the summation variable ranging from 1 to i_2. The sum runs only over the inner iterator i_2, so each segment is summed independently. Compare with the 1D rule RecSum from Section 4.2.1, where the sum runs over the entire flat array; here the same logic operates per segment because the recurrence lives in the flattened iterator.

Applied to `indsT`, with $e_3 = \texttt{tflgs}(i_1, i_2)$ and a single recurrent term $\texttt{tflgs}(i_1, i_2)$, the rule yields

$$\texttt{indsT} \;\mapsto\; \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\; \texttt{tflgs}(i_1, 0) + \sum_{x=1}^{i_2} \texttt{tflgs}(i_1, x).$$

The base case $\texttt{tflgs}(i_1, 0)$ has the same form as the summand, so we can absorb it into the sum by extending the lower bound to 0:

$$\texttt{indsT} \;\mapsto\; \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\; \sum_{j=0}^{i_2} \texttt{tflgs}(i_1, j).$$

Substituting `tflgs`'s body, the 0 guard contributes nothing, so we are left with

$$\texttt{indsT} \;\mapsto\; \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\; \sum_{j=0}^{i_2} [\texttt{csL}(i_1, j)].$$

The same chain fires on `tmp = sgm_sum flags fflgs`. The 1D index function for `fflgs` from Phase 1 has body $1 - [\texttt{csL}(j)]$, so the sum becomes $\sum_{j=0}^{i_2}(1 - [\texttt{csL}(i_1, j)])$. Splitting the sum and using $\sum_{j=0}^{i_2} 1 = i_2 + 1$ gives

$$\texttt{tmp} \;\mapsto\; \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\; (i_2 + 1) - \sum_{j=0}^{i_2} [\texttt{csL}(i_1, j)].$$

At this point in the derivation, `indsT` and `tmp` are the first non-auxiliary intermediates to carry a flattened domain. `offs` and `indsF` will follow shortly.

Phase 3: combining flattened intermediates.

The SubFlat rule. The next intermediate is `lst = map2 (\s b -> if s == 0 then -1 else indsT[b-1]) shape ends`. Both `shape` and `ends` are 1D, so the 1D Map rule gives `lst` a 1D outer domain $k : 0..m$. The body indexes into `indsT`, which now has a flattened domain. Substituting `ends`'s body from Phase 1 for the indexing argument, the expression $\texttt{indsT}(b - 1)$ becomes $\texttt{indsT}\big(\sum_{j=0}^{k} \texttt{shape}(j) - 1\big)$.

The rule that handles this case is SubFlat. It substitutes the body of a flattened index function into an indexing expression with an arbitrary flat index, introducing a

%-expression where the outer iterator's value must be recovered. The rule operates on expressions rather than on whole index functions.

Before stating the rule, we introduce its key notation. The conclusion uses an expression of the form $\%_D(e)$, read as "the segment number in domain D that contains flat position e". This expression is deferred: it is resolved later by a SolveIdx rule when the indexing pattern is recognizable, and otherwise stays symbolic. With this in mind, the rule is:

SubFlat

$$\frac{\Gamma(x) = \lambda\,(i_1 : 0..e_2 \times i_2 : 0..e_3)\;.\;e_x \qquad \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_3[i_1 := j]) \rightsquigarrow e_{row}}{\Gamma \vdash \mathcal{C}\langle x(e_{idx})\rangle \to \mathcal{C}\langle e_x[i_1 := \%_{i_1:0..e_2\times i_2:0..e_3}(e_{idx}),\, i_2 := e_{idx} - e_{row}[i_1 := \%_{i_1:0..e_2\times i_2:0..e_3}(e_{idx})]]\rangle}$$

The first premise looks up an index function with flattened domain $i_1 : 0..e_2 \times i_2 : 0..e_3$ and body e_x. The second and third premises compute the row start as in PropFlatten. The conclusion replaces $x(e_{idx})$, an indexing operation with an arbitrary flat index, with the body e_x in which both flattened iterators have been substituted: i_1 is replaced by the symbolic %-expression $\%_{i_1:0..e_2\times i_2:0..e_3}(e_{idx})$ denoting "the segment number containing flat position e_{idx}", and i_2 is replaced by the within-segment offset.

In other words, SubFlat expresses an arbitrary flat indexing operation in terms of "the segment it lands in" and "the offset within that segment", without committing to a specific resolution of the segment number. The SolveIdx rules (below) then resolve the %-expression to a concrete iterator value when the indexing pattern is recognizable.

Inside `lst`'s body, the expression $\texttt{indsT}\big(\sum_{j=0}^{k}\texttt{shape}(j) - 1\big)$ is an indexing into a flattened array from a 1D context. SubFlat fires on this indexing expression (not on `lst` itself, which is 1D and has no flattened domain to look up). The rule inlines `indsT`'s body and introduces the %-expression

$$\%_{i_1:0..m\times i_2:0..\texttt{shape}(i_1)}\Big(\sum_{j=0}^{k}\texttt{shape}(j) - 1\Big)$$

for the segment number containing the last position of segment k. SubFlat itself does not require a flattened surrounding lambda: it only inlines a flattened index function's body and records the still-unknown segment number as a %-expression. Resolving that %-expression, on the other hand, does depend on the surrounding domain, and it is the 1D context here that determines which SolveIdx rule applies. This expression cannot be simplified without recognizing the indexing pattern, which is the job of the one-dimensional case of SolveIdx1, introduced next.

The SolveIdx1 rule (one-dimensional case). The subscript domain D of this %-expression is flattened, but the surrounding lambda `lst` is one-dimensional. The framework's SolveIdx rules (SolveIdx1, SolveIdx2, SolveIdx3) all match a *flattened* surrounding lambda $\lambda(i_1 : 0..e_1 \times i_2 : 0..e_2)$ and unify that domain with D, so none of them fires directly on a 1D lambda. What is needed is the bounds-checking idea of

SolveIdx1, specialized to a one-dimensional surrounding lambda whose single iterator names the segment directly:

$$\frac{\begin{array}{c}\text{SolveIdx1-1D}\\ \text{unify}\,(k : 0..e_1,\ i_1 : 0..e_1) = \{i_1 \mapsto k\} \qquad \text{fresh } j \qquad \Gamma \vdash \textstyle\sum_{j=0}^{k-1}(e_2[i_1 := j]) \rightsquigarrow e_{row}\\ (\Gamma, \mathsf{Range}\ k\ 0..e_1) \vdash \text{Query}\ (e_{row} \le e_{idx} < e_{row} + e_2[i_1 := k]) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \lambda\,(k : 0..e_1)\ .\ \mathcal{C}\langle \%_{i_1:0..e_1 \times i_2:0..e_2}(e_{idx})\rangle \to \lambda\,(k : 0..e_1)\ .\ \mathcal{C}\langle k\rangle}$$

The first premise aligns `lst`'s iterator k with the outer dimension of D, requiring their bounds to agree and producing the substitution $\{i_1 \mapsto k\}$. The second and third premises compute the row start $e_{row} = \sum_{j=0}^{k-1} e_2[i_1 := j]$, the start of segment k. The fourth premise asks the solver to prove that the flat index lies inside segment k, namely $e_{row} \le e_{idx} < e_{row} + e_2[i_1 := k]$; when it holds, the `%`-expression resolves to k. The segment-length information e_2 comes from the `%`-expression's own subscript domain rather than from the surrounding lambda, which is what lets the rule run in a 1D context.

Applied to `lst`, the subscript domain is $i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)$, whose outer bound m matches `lst`'s domain $k : 0..m$, so i_1 aligns with k. The row start is $e_{row} = \sum_{j=0}^{k-1} \texttt{shape}(j)$ and the current segment length is $e_2[i_1 := k] = \texttt{shape}(k)$. With $e_{idx} = \sum_{j=0}^{k} \texttt{shape}(j) - 1$, the bounds query is

$$\sum_{j=0}^{k-1} \texttt{shape}(j) \ \le\ \sum_{j=0}^{k} \texttt{shape}(j) - 1 \ <\ \sum_{j=0}^{k} \texttt{shape}(j),$$

whose upper bound holds unconditionally and whose lower bound reduces to $\texttt{shape}(k) \ge 1$. That holds under the guard $\texttt{shape}(k) \neq 0$, which is exactly the non-empty branch of the source conditional in which the indexing $\texttt{indsT}[b-1]$ occurs. The query therefore succeeds and the `%`-expression resolves to k, so `indsT`'s body $\sum_{j=0}^{i_2}[\texttt{csL}(i_1, j)]$ is instantiated at $i_1 = k$ and $i_2 = \texttt{shape}(k) - 1$, yielding the total number of `true` values in segment k:

$$\texttt{lst} \ \mapsto\ \lambda(k : 0..m).\ [\texttt{shape}(k) = 0] * -1 + [\texttt{shape}(k) \neq 0] * \sum_{j=0}^{\texttt{shape}(k)-1} [\texttt{csL}(k, j)]$$

The conditional on $\texttt{shape}(k) = 0$ handles empty segments, where the indexing would be out of bounds and the source code returns -1. For non-empty segments, $\texttt{lst}(k)$ is exactly the count of trues in segment k. `lst` itself stays 1D over $k : 0..m$: it has one entry per segment, not per flat position, which is why the size check ruled out PropFlatten earlier. The shorthand $\texttt{csL}(k, j)$ used in the result reflects k playing the role of i_1, as noted in the notation paragraph of Section 5.

Although $\sum_{j=0}^{k} \texttt{shape}(j) - 1$ is syntactically the end of segment k, it is resolved here through the bounds-checking SolveIdx1, not through an endpoint-matching rule. This is deliberate: `lst` is one-dimensional, so an endpoint rule that matches a flattened surrounding lambda would not apply, and the bounds check additionally requires $\texttt{shape}(k) \neq 0$, correctly declining the empty-segment case where $\sum_{j=0}^{k} \texttt{shape}(j) - 1$ would fall in an earlier segment.

The SolveIdx1 rule (interior case). The one-dimensional case above resolves an index landing on the last position of a segment. The references to `seg_ids` in `indsF` and `offs` below differ in two ways: the surrounding lambda is flattened rather than 1D, and the index lands at an arbitrary offset i_2 *inside* a segment, not on its first or last position. The general flattened form of SolveIdx1 resolves such an interior index:

$$\textsc{SolveIdx1}\quad \frac{\begin{array}{c}\text{unify}\,(i_1 : 0..e_1 \times i_2 : 0..e_2,\ D) = \sigma \qquad \text{fresh } j \qquad \Gamma \vdash \textstyle\sum_{j=0}^{i_1-1}(e_2[i_1 := j]) \rightsquigarrow e_{row} \\ (\Gamma, \mathsf{Range}\ i_1\ 0..e_1, \mathsf{Range}\ i_2\ 0..e_2) \vdash \text{Query}\ (0 \le \sigma(e_{idx}) - e_{row} < e_2) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ \mathcal{C}\langle \%_D(e_{idx})\rangle \to \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ \mathcal{C}\langle i_1\rangle}$$

Its first three premises unify the subscript domain D with the surrounding flattened domain and compute the row start e_{row} as in PropFlatten. The fourth premise asks the solver to prove $0 \le \sigma(e_{idx}) - e_{row} < e_2$, that is, that the index lies within row i_1's bounds. When this holds, the `%`-expression resolves to the outer iterator i_1. This is the same bounds obligation as in the one-dimensional case above; the difference is only that the segment number it produces is the flattened domain's own outer iterator rather than a separate 1D iterator. The framework also provides SolveIdx2 and SolveIdx3, which resolve indices that fall syntactically on the start or end of a segment respectively; neither is exercised by `part2indicesL`.

Deriving `indsF`. `indsF = map2 (\t k -> t + lst[k]) tmp seg_ids` indexes into both `tmp` (flattened, size n) and `seg_ids` (flattened, size n) in its argument arrays, and indexes into `lst` (1D, size m) inside the lambda body. Because `tmp` is referenced in the body and `indsF`'s outer domain matches `tmp`'s in total size, PropFlatten fires here, lifting `indsF` to the flattened domain $i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)$. After this lifting, the body's reference to `seg_ids` at the flat iterator becomes $\texttt{seg\_ids}(e_{row} + i_2)$, and `seg_ids` is itself flattened. SubFlat fires on this reference, introducing the `%`-expression $\%_{i_1:0..m \times i_2:0..\texttt{shape}(i_1)}(e_{row} + i_2)$, which resolves to i_1 via SolveIdx1: the index $e_{row} + i_2$ is the interior position at offset i_2 of row i_1, and the bounds query $0 \le (e_{row} + i_2) - e_{row} = i_2 < \texttt{shape}(i_1)$ holds because i_2 is the inner iterator, with range $0..\texttt{shape}(i_1)$. Substituting `seg_ids`'s body (which is i_1) for k in the body of `indsF` replaces the lookup `lst[k]` with $\texttt{lst}(i_1)$.

Now `lst` is 1D, so resolving the reference $\texttt{lst}(i_1)$ is a job for the ordinary Sub rule, not SubFlat. Sub inlines `lst`'s body at the argument i_1. Since $i_2 < \texttt{shape}(i_1)$ implies $\texttt{shape}(i_1) \neq 0$, the non-empty branch of `lst` applies, and $\texttt{lst}(i_1) = \sum_{j=0}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1, j)]$. After these rewrites,

$$\texttt{indsF} \ \mapsto\ \lambda\big(i_1 : 0..m \ \times\ i_2 : 0..\texttt{shape}(i_1)\big).\ \texttt{tmp}(i_1, i_2) + \texttt{lst}(i_1).$$

Substituting the bodies of `tmp` and `lst` gives

$$\texttt{indsF} \ \mapsto\ \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\ (i_2{+}1) - \sum_{j=0}^{i_2}[\texttt{csL}(i_1, j)] + \sum_{j=0}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1, j)].$$

Deriving `offs`. `offs` follows the same shape of derivation as `indsF`, with `seg_ids` playing the role that `tmp` played there. PropFlatten fires once `seg_ids`'s flattened domain is propagated into the body. SubFlat introduces the same interior `%`-expression as in `indsF`, which SolveIdx1 resolves to i_1 by the identical bounds query; inlining `seg_ids`'s body i_1 into the conditional then gives

$$\texttt{offs} \;\mapsto\; \lambda\big(i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)\big).\; [i_1 > 0] * \texttt{ends}(i_1 - 1) + [i_1 = 0] * 0$$

which is the segment start of i_1 in the flat output. Under the convention $\texttt{ends}(-1) = 0$ from Section 5.4.1, the two guarded cases collapse to the single expression $\texttt{ends}(i_1 - 1)$, independently of i_2. This matches the shorthand $\texttt{offs}(i_1, i_2) = \texttt{ends}(i_1 - 1)$ used below.

Assembling `inds`. The final `map4` takes `csL`, `indsT`, `indsF`, and `offs` as arguments. The segmented-aware Map rule unifies their domains over $i_1 : 0..m \times i_2 : 0..\texttt{shape}(i_1)$ (`csL` is 1D throughout and contributes a 1D reference indexed in flattened coordinates, per the shorthand of Section 5.4.1), and derives the body of `inds` as the conditional

$$\begin{cases} \texttt{csL}(i_1, i_2) \;\Rightarrow\; \texttt{offs}(i_1, i_2) + \texttt{indsT}(i_1, i_2) - 1, \\ \neg\,\texttt{csL}(i_1, i_2) \;\Rightarrow\; \texttt{offs}(i_1, i_2) + \texttt{indsF}(i_1, i_2) - 1. \end{cases}$$

Substituting the bodies of `offs`, `indsT`, and `indsF` into this conditional gives the final form.

For the true case, we use the case assumption $\texttt{csL}(i_1, i_2)$, which makes the indicator $[\texttt{csL}(i_1, i_2)]$ evaluate to 1. This is what makes the next step algebraic rather than guard-dependent:

$$\begin{aligned} \texttt{offs}(i_1, i_2) + \texttt{indsT}(i_1, i_2) - 1 &= \texttt{ends}(i_1 - 1) + \sum_{j=0}^{i_2} [\texttt{csL}(i_1, j)] - 1 \\ &= \texttt{ends}(i_1 - 1) + \sum_{j=0}^{i_2 - 1} [\texttt{csL}(i_1, j)] + [\texttt{csL}(i_1, i_2)] - 1 \\ &= \texttt{ends}(i_1 - 1) + \sum_{j=0}^{i_2 - 1} [\texttt{csL}(i_1, j)] + 1 - 1 \\ &= \texttt{ends}(i_1 - 1) + \sum_{j=0}^{i_2 - 1} [\texttt{csL}(i_1, j)]. \end{aligned}$$

In the false case, the two indicator sums inside `indsF` telescope directly:

$$\begin{aligned}\texttt{offs}(i_1,i_2)+\texttt{indsF}(i_1,i_2)-1 &= \texttt{ends}(i_1-1)+(i_2+1)-1\\ &\quad -\sum_{j=0}^{i_2}[\texttt{csL}(i_1,j)]+\sum_{j=0}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1,j)]\\ &= \texttt{ends}(i_1-1)+i_2-\sum_{j=0}^{i_2}[\texttt{csL}(i_1,j)]+\sum_{j=0}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1,j)]\\ &= \texttt{ends}(i_1-1)+i_2+\sum_{j=i_2+1}^{\texttt{shape}(i_1)-1}[\texttt{csL}(i_1,j)].\end{aligned}$$

The first step combines the constants $(i_2+1)-1=i_2$, and the last step rewrites $\sum_{j=0}^{\texttt{shape}(i_1)-1}-\sum_{j=0}^{i_2}$ as the single sum $\sum_{j=i_2+1}^{\texttt{shape}(i_1)-1}$. Unlike the true case, the false case closes by pure algebra and does not need to invoke the case assumption $\neg\,\texttt{csL}(i_1,i_2)$, because the two indicator sums telescope directly. Both cases match the flattened index function for `inds` stated in Section 5.4.1.

6 Implementation

This chapter describes the main implementation changes made to PropProp in this thesis. The focus is on the parts that are most important for understanding how the formal rules from the previous chapters were realized in the compiler: propagating flattened domains, removing the remaining `Cat`-based segmented construction from the scatter rule, and updating substitution and index solving for flattened irregular domains.

The implementation contains additional smaller changes, cleanups, and supporting adjustments that are not discussed in detail here. Those details are available in the source code. The goal of this chapter is instead to explain the changes that best show the connection between the formal rules and the implementation.

A useful starting point is the internal representation of domains. Before this thesis, segmented structure could still be represented explicitly in the compiler through the `Cat` constructor. One of the main goals of the implementation work was to remove this representation and instead express segmented structure using the flattened-domain machinery introduced in the previous chapters. Figure 11 shows this change at the datatype level.

```
data Domain
  = Iota (SoP Symbol) -- [0, ..., n-1]
  | Cat -- Cat{k=0}^{m-1} [b_k, ..., b_{k+1})
      VName -- k
      (SoP Symbol) -- m
      (SoP Symbol) -- b
  deriving (Show, Eq)
```

(a) Previous representation

```
data Domain
  = Iota (SoP Symbol) -- [0, ..., n-1]
  deriving (Show, Eq)
```

(b) Updated representation

Figure 11: Removing `Cat` from the internal domain representation. Segmented structure is no longer represented by a separate constructor, but instead through flattened domains built from ordinary `Iota` components.

This change is the basis for the rest of the implementation work described in this chapter. Once `Cat` is removed as a representation choice, the remaining rules must either propagate flattened structure directly or construct it explicitly when segmented information is recovered. The following sections show how this was done for PropFlatten, the segmented scatter rule, and the later substitution and simplification steps that operate on flattened irregular domains.

6.1 Implementing PropFlatten

The previous chapter introduced PropFlatten as the rule that moves an index function from a flat one-iterator view to a flattened segmented view. This section explains how this rule is implemented. The main function is `propFlattenOnce`, which tries to apply PropFlatten once to one selected dimension.

The purpose of the function is simple. We have one index function g, whose dimension

has the ordinary flat form $i_1 : 0..e_1$ and another index function f, whose corresponding dimension already exposes a flattened structure $i_2 : 0..e_2 \times i_3 : 0..e_3$. If these two domains describe the same flat positions, then g can inherit the flattened domain from f. This does not change the meaning of g.

Figure 12 shows the formal rule and the corresponding implementation. They are placed together because the implementation is easiest to read as a direct realization of the rule. Here $\mathcal{C}\langle\cdot\rangle$ denotes the surrounding expression context, so the occurrence $x(e_{idx})$ may appear inside a larger expression.

The important parts of the rule are the following. First, the environment contains an index function x with a flattened domain $i_2 : 0..e_2 \times i_3 : 0..e_3$. Second, the rule computes the row-start expression e_{row}, which gives the flat position at which segment i_2 begins. Third, it checks that the one-iterator domain $i_1 : 0..e_1$ and the flattened domain have the same total size. If this check succeeds, the rule replaces the old flat iterator i_1 by $e_{row} + i_3$ and gives the result the flattened domain.

In the implementation, the same step is performed by `propFlattenOnce`. The argument k selects which top-level dimension should be rewritten. The index function g is the function that may still use a flat iterator in this dimension, while f is the function that already exposes the flattened structure. The result is optional: if the pattern does not match, or if the required equality cannot be proved, the function returns `Nothing`. This does not necessarily mean that the program is wrong. It only means that this particular rewrite rule did not apply.

It is useful to explain a few implementation constructors that appear in the code. The implementation represents symbolic arithmetic expressions as values of type `SoP Symbol`, short for sum-of-products over symbols. The helper `sym2SoP` embeds a symbolic atom into this expression language, while `int2SoP` embeds an integer constant. For example, `sym2SoP (Var i2)` is the expression corresponding to the iterator i_2, and `int2SoP 1` is the constant 1. Operators such as `.+.`, `.-.`, and `.*.` build symbolic addition, subtraction, and multiplication. The constructors `Var`, `Apply`, and `Ix` are symbols inside this expression language: `Var i` denotes a variable or iterator, `Apply (Var x) args` denotes an application of an index function or symbolic array x, and `Ix` is the symbolic segment-index operation used later to recover an outer segment from a flat index. Expressions such as `sym2SoP (Var i2) .*. e3` are not evaluated at runtime. They construct symbolic expressions that are later rewritten or sent to the solver.

Lines 3–5 first check that the selected dimension k exists in both index functions. Here, rank(g) is the number of top-level dimensions in shape(g). A flattened dimension still counts as one top-level dimension, even if it contains more than one iterator. If k is out of bounds for either g or f, then the rule cannot apply.

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

(a) Formal rule

```
1  propFlattenOnce :: Int -> IndexFn -> IndexFn -> IndexFnM (Maybe IndexFn)
2  propFlattenOnce k g f = runMaybeT $ do
3    if k >= rank g || k >= rank f
4      then fail "No match."
5      else case (shape g !! k, shape f !! k) of
6        ([Forall i1 (Iota e1)], df@[Forall i2 (Iota e2), Forall i3 (Iota e3)]) -> do
7          if i2 `S.notMember` fv e3
8            then do
9              ans <- lift (e1 $== e2 .*. e3)
10             case ans of
11               Yes -> do
12                 eRow <- lift . rewrite $ sym2SoP (Var i2) .*. e3
13                 let s = mkRep i1 (eRow .+. sym2SoP (Var i3))
14                 let (l, _old : r) = splitAt k (shape g)
15                 let res = g { shape = l <> (df : r)
16                             , body = repCases s (body g) }
17                 if res == g
18                   then fail "regular PropFlatten made no progress"
19                   else pure res
20               Unknown -> fail "Could not prove size equality"
21           else do
22             eRow <- lift $ mkERow i2 e3
23             end  <- lift . rewrite $ rep (mkRep i2 e2) eRow
24             ans  <- lift (e1 $== end)
25             case ans of
26               Yes -> do
27                 let s = mkRep i1 (eRow .+. sym2SoP (Var i3))
28                 let (l, _old : r) = splitAt k (shape g)
29                 let res = g { shape = l <> (df : r)
30                             , body = repCases s (body g) }
31                 if res == g
32                   then fail "irregular PropFlatten made no progress"
33                   else pure res
34               Unknown -> fail "Could not prove size equality"
35       _ -> fail "No match."
```

(b) Implementation

Figure 12: The PropFlatten rule and the implementation of `propFlattenOnce`. The rule describes the rewrite abstractly, while the code shows how the regular and irregular cases are handled in the compiler.

Lines 5–6 then inspect the selected dimensions of g and f. The selected dimension of g must have the flat one-iterator form $i_1 : 0..e_1$. In the implementation, this is represented as `[Forall i1 (Iota e1)]`. The selected dimension of f must have the flattened two-iterator form $i_2 : 0..e_2 \times i_3 : 0..e_3$, represented as `[Forall i2 (Iota e2), Forall i3 (Iota e3)]`. The whole flattened dimension is stored as `df`, because it is

later inserted into shape(g). In other words, `df` is the concrete implementation object corresponding to the flattened domain that g should inherit.

The regular case. Lines 7–20 handle the regular case. This branch is taken when the inner bound e_3 does not depend on the outer iterator i_2, expressed by the condition $i_2 \notin fv(e_3)$. In this situation, all rows have the same length. Therefore, the total number of flat elements in the flattened dimension is $e_2 \cdot e_3$. Line 9 asks the solver to prove that this total size is equal to the flat size e_1 of g. This is the regular version of the size check in PropFlatten.

If the equality is proved, lines 11–19 construct the rewritten index function. Line 12 computes the row start as $e_{row} = i_2 \cdot e_3$. Line 13 builds the substitution $i_1 := e_{row} + i_3$, which says that the old flat iterator i_1 should now be expressed as the start of row i_2 plus the local offset i_3. Lines 14–16 then build the result: the selected dimension of shape(g) is replaced by the flattened dimension `df`, and the substitution is applied to body(g).

This branch corresponds to a simplified case of PropFlatten where the row-start sum collapses to a multiplication. Instead of constructing $\sum_{j=0}^{i_2-1} e_3[i_2 := j]$, the implementation can use $i_2 \cdot e_3$ directly.

The irregular case. Lines 21–34 handle the irregular case. This branch is taken when the inner bound e_3 may depend on the outer iterator i_2. This is the case needed for segmented arrays, where different segments may have different lengths. The row start can no longer be computed by multiplication, because there is no single fixed row length.

Line 22 constructs the general row-start expression $e_{row} = \sum_{j=0}^{i_2-1} e_3[i_2 := j]$. This expression gives the flat position where segment i_2 begins. Line 23 computes the end of the flattened domain by substituting the outer bound e_2 for i_2 in the row-start expression. Intuitively, this gives the flat position after the last segment, so it is the total length of the flattened dimension. Line 24 asks the solver to prove that this total length is equal to the flat length e_1 of g. This is the size condition needed to show that the flat domain of g and the flattened domain of f describe the same set of positions.

If the equality is proved, lines 26–33 perform the same transformation as in the regular branch. Line 27 builds the substitution $i_1 := e_{row} + i_3$. Lines 28–30 replace the selected dimension of shape(g) by the flattened dimension from f, and apply the substitution to body(g). The result is an index function that has the same meaning as before, but whose selected dimension is now described using the segment iterator i_2 and the local offset i_3.

How the code matches the rule. The correspondence with the formal rule is direct. The premise $\Gamma(x) = \lambda(i_2 : 0..e_2 \times i_3 : 0..e_3).\, e_x$ is represented by the pattern match on the selected dimension of f in lines 5–6. The flat index function in the conclusion of the rule is represented by g, whose selected dimension must match $i_1 : 0..e_1$. The computation of e_{row} appears in two forms: line 12 uses the simplified regular expression $i_2 \cdot e_3$, while line 22 uses the general prefix-sum construction for irregular segments.

The size premise of the rule is also visible in both branches. In the regular branch, line 9 proves the simplified equality $e_1 = e_2 \cdot e_3$. In the irregular branch, lines 23–24 compute the total flattened size and prove that it equals e_1. These checks ensure that changing the domain of g does not change the number of flat positions it describes.

The conclusion of the rule has two implementation effects. First, body(g) is rewritten by replacing the old flat iterator i_1 with $e_{row} + i_3$. This happens in lines 16 and 30, where `repCases` applies the substitution to body(g). Second, the selected flat dimension of shape(g) is replaced by the flattened dimension copied from f. This happens in lines 15 and 29. Thus, the implementation performs both parts of the formal rule: it rewrites the body and updates the domain.

The checks in lines 17–19 and lines 31–33 are not part of the formal rule. They are implementation safeguards. If the computed result is identical to the original index function, the rewrite is treated as not making progress. This prevents the rewriting process from repeatedly applying a rule that leaves the index function unchanged.

Validation. The implementation of `propFlattenOnce` was also given direct unit tests for both branches of the rule. This was useful because most existing tests are end-to-end tests over complete `.fut` programs, which makes it harder to debug a single rewrite rule in isolation. The direct tests check that the regular case rewrites the flat iterator using $i_2 \cdot e_3 + i_3$, and that the irregular case rewrites it using the prefix-sum row start $\sum_{j=0}^{i_2-1} e_3[i_2 := j] + i_3$. These tests do not replace the larger program-level regression tests, but they confirm that the implementation performs the local transformation expected by the rule. The full testing setup is discussed in Section 7.1.

6.2 Producing Segmented Flattened Domains in the Scatter Rule

After implementing PropFlatten, the next step was to update the conversion rule that most visibly still produced segmented structure in the old `Cat`-based form. The most important case was the scatter rule implemented by `scatterSc2`. This rule handles a special kind of scatter where the index array describes segment boundaries in a flat destination array. In the old implementation, this rule reconstructed the segmented structure using `Cat`. In the updated implementation, it constructs the flattened domain directly. This does not mean that the `Cat` constructor was removed from the datatype. The constructor is still present, and some legacy utilities and proof rules still pattern-match on it. The refactor in this section is narrower: `scatterSc2` should no longer produce `Cat`-shaped output, but should instead produce the explicit flattened representation used by the new theory.

This section does not reintroduce the old and new representations in full, since that correspondence was explained in Section 5.2. The important point here is how the scatter conversion now constructs its result. Instead of producing a domain where one iterator ranges over a `Cat` segment, the rule now produces one flattened dimension containing two iterators:

$$k : 0..m \times i : 0..(e(k+1) - e(k)).$$

Here k identifies the segment and i is the local offset inside that segment. The corresponding absolute flat position is $e(k) + i$.

How scatter conversion is organised. The conversion of a `scatter` has two related concerns. First, the analysis must establish that the scatter is deterministic. A scatter is safe when parallel writes cannot disagree. This can be shown, for example, when the in-bounds indices are injective, or when all written values are equal. In the implementation, this check is grouped in `scatterSafe`, which tries several sufficient conditions.

Second, if the scatter is safe, the analysis tries to construct a useful index function for the result. The implementation tries several conversion cases. The first precise case, `scatterSc1`, handles scatters whose index array is bijective over the destination domain. In that case, the result can be described using the inverse of the index array. The second precise case, `scatterSc2`, handles scatters whose indices describe segment boundaries. If neither precise case applies, `scatterSc3` can still produce an uninterpreted result for a scatter that has been proved safe.

The refactor in this section concerns `scatterSc2`. The rest of the scatter algorithm is useful context, but `scatterSc2` was the main conversion case that constructed a segmented output domain. In the old implementation, that segmented output domain was represented using `Cat`, but in the updated implementation the same structure is represented as an explicit flattened dimension.

Explaining the inference rule The formal rule corresponding to `scatterSc2` is Scatter3, shown in Figure 13 and it is taken from the appendix of [9]. This rule covers the case where a scatter does more than write unrelated values into unrelated positions. It recognizes when the in-bounds scatter indices form a monotone sequence of boundary positions in the destination array.

For a `scatter` $x_{dst}\ x_{idx}\ x_{src}$, the destination array x_{dst} is the flat array being updated, x_{idx} contains the write positions, and x_{src} contains the values being written. The goal of Scatter3 is to identify the special case where the valid values of x_{idx} can be interpreted as segment starts.

The rule first requires the in-bounds indices to be injective. This is part of scatter safety: two different source positions must not write conflicting values to the same valid destination position. The next premise rewrites the index array into two branches. The useful branch has guard p and boundary expression e_{row}. This is the branch that produces valid destination positions. The other branch is represented by a fresh dummy value $x_{\perp}$, which stands for writes that are ignored because they are out of bounds.

The following three queries check that e_{row} really behaves like a sequence of segment boundaries. The first boundary must be 0, the final boundary must reach $|x_{dst}|$, and the boundary expression must be monotone between these endpoints. If these conditions hold, then segment i_1 starts at e_{row} and ends at $e_{row}[i_1 := i_1 + 1]$. Therefore its length is $e_{row}[i_1 := i_1 + 1] - e_{row}$.

SCATTER3

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

(a) Formal monotone scatter rule

```haskell
scatterSc2 :: IndexFn -> IndexFn -> IndexFn -> MaybeT IndexFnM IndexFn
scatterSc2 xs@(IndexFn [[Forall _ d_xs]] _) is@(IndexFn [[Forall k (Iota m)]] _) vs = do
  n <- lift $ rewrite $ domainEnd d_xs .+. int2SoP 1

  let in_dom_xs y = int2SoP 0 :<= y :&& y :< n
  in_bounds <- lift $ mapM (queryCase (CaseCheck in_dom_xs) is)
                           [0 .. length (guards is) - 1]

  let gs = L.sortOn fst $ zip in_bounds (guards is)

  case gs of
    [(Unknown, _), (Yes, (c, e))] -> do
      Yes <- lift $ algebraContext is $ do
        addRelShape (shape is)
        neg c =>? (e :>= rep (mkRep k (sVar k .+. int2SoP 1)) e)

      Yes <- lift $ algebraContext is $ do
        addRelShape (shape is)
        k' <- newNameFromString "k"
        k +< k'
        c =>? (e :<= rep (mkRep k (sVar k')) e)

      Yes <- lift $ Bool True =>?
        (rep (mkRep k (int2SoP 0)) e :== int2SoP 0)

      Yes <- lift $ Bool True =>?
        (rep (mkRep k m) e .-. int2SoP 1 :== n .-. int2SoP 1)

      ...
    _ -> failMsg "scatterSc2: unable to determine OOB branch"
scatterSc2 _ _ _ = fail ""
```

(b) Selected implementation checks in `scatterSc2`

Figure 13: The SCATTER3 rule and the corresponding checks in `scatterSc2`.

Before comparing the code with the rule, it is useful to spell out the small query language used in the implementation. The function `queryCase` asks a solver question about one guarded case of an index function. In this example, `CaseCheck in_dom_xs` means: check whether the expression produced by this case is inside the destination domain. The result is an `Answer(Yes / Unknown)`. The expression `algebraContext is` runs a query in the algebraic context of the index function `is`; inside this context,

`addRelShape (shape is)` adds the iterator bounds from the shape of `is`, such as $0 \leq k < m$. The operator `=>?` sends an implication query to the solver: `a =>? b` asks whether $a \Rightarrow b$ can be proved. The comparison constructors `:==`, `:<=`, `:<`, and `:>=` build symbolic predicates; they become solver queries only when passed to `=>?` or a similar query function. Finally, `rep (mkRep k (sVar k .+. int2SoP 1)) e` corresponds to $e[k := k + 1]$.

The conclusion of the rule gives the scatter result a flattened domain. The outer iterator chooses the segment, and the inner iterator chooses the local offset inside that segment. At local offset 0, when the boundary guard holds, the result takes the scattered value $x_{src}(i_1)$. At all other offsets, the result keeps the old destination value at the corresponding absolute flat position $e_{row} + i_2$.

The implementation uses different names from the formal rule, but the correspondence is direct. The destination array x_{dst} is `xs`, the scatter index array x_{idx} is `is`, and the scattered value array x_{src} is `vs`. The rule's outer segment iterator i_1 corresponds to the implementation variable k. The local offset iterator i_2 is introduced later as the fresh variable i, when the result shape is constructed.

The first line of the implementation pattern matches on the shapes needed by this conversion case. The destination `xs` must have one flat dimension, and the index array `is` must have a one-dimensional domain $k : 0..m$. This matches the rule's use of $|x_{idx}|$ as the number of potential segment boundaries.

Lines 4–7 classify the guarded cases of the index function `is`. The local function `in_dom_xs` builds the predicate $0 \leq y < n$, where n is the destination length. The call `queryCase (CaseCheck in_dom_xs) is` then asks, for each guarded case of `is`, whether the expression produced by that case is always inside the destination domain. This is the implementation counterpart of the rule's separation of x_{idx} into an in-bounds branch $[p] * e_{row}$ and an ignored out-of-bounds branch $[\neg p] * x_{\perp}$.

Lines 9–12 identify the successful in-bounds branch. In the implementation this branch is written as (c, e). The guard c corresponds to the rule's guard p, and the expression e corresponds to the boundary expression e_{row}. From this point on, the implementation tries to prove that e can really be used as the segment-start function.

Lines 13–21 check the ordering conditions that make e behave like a boundary function. Each check is discharged by the algebra solver through `=>?`. In the first query, the implementation substitutes $k + 1$ for k in e, written as `rep (mkRep k (sVar k .+. int2SoP 1)) e`. The query then asks whether, under the out-of-bounds branch condition $\neg c$, the current boundary is at least the next boundary. This captures the case where the corresponding segment is empty. In the second query, a fresh iterator k' is introduced and constrained by `k +< k'`, meaning $k < k'$. The solver is then asked to prove that, under the in-bounds guard c, the boundary at k does not exceed the boundary at k'. Together, these checks establish the monotone behavior required by the formal rule.

Lines 23–26 check the endpoint conditions. The query `Bool True =>? (... :== ...)` is an unconditional solver query, because the antecedent is `true`. The first endpoint query proves that the first boundary is 0, by checking $e[k := 0] = 0$. The second

proves that the final boundary reaches the end of the destination domain, by checking $e[k := m] - 1 = n - 1$. The implementation writes this with `:==`, which constructs an equality predicate. The monotonicity and endpoint queries justify interpreting the interval between $e(k)$ and $e(k+1)$ as segment k.

Constructing the flattened result. Once the boundary structure has been recognized, the updated implementation no longer constructs a `Cat` domain. Figure 14 shows the old construction, commented out in the code, and the new construction.

The code in Figure 14 uses the same symbolic-expression representation introduced in Section 6.1. Thus, `.+.` and `.-.` construct symbolic arithmetic expressions, and `sym2SoP` embeds symbolic applications into the sum-of-products representation. The expressions built here are not source-level Futhark expressions. They are internal index-function expressions used by the analysis.

```
-- Old construction:
-- IndexFn
--   { shape = [[Forall i (Cat k m e)]],
--     body =
--       cases
--         [ (sVar i :== e :&& c', sym2SoP £ Apply (Var hole_vs) [sVar k]),
--           (neg p,               sym2SoP £ Apply (Var hole_xs) [sVar i])
--         ]
--   }
```

(a) Old `Cat`-based construction

```
i <- newVName "i"
hole_xs <- newVName "hole_xs"
hole_vs <- newVName "hole_vs"

let e_next = rep (mkRep k (sVar k .+. int2SoP 1)) e
let len    = e_next .-. e
let abs_i  = e .+. sVar i

let p = (sVar i :== int2SoP 0) :&& c'

let f =
      IndexFn
        { shape = [[Forall k (Iota m), Forall i (Iota len)]],
          body =
            cases
              [ (p,     sym2SoP $ Apply (Var hole_vs) [sVar k]),
                (neg p, sym2SoP $ Apply (Var hole_xs) [abs_i])
              ]
        }

lift $ substParams f [(hole_vs, vs), (hole_xs, xs)]
```

(b) New flattened-domain construction

Figure 14: The representation change in `scatterSc2`. The old construction used `Cat` to describe the segment interval, while the new construction uses a flattened dimension with a segment iterator and a local offset.

The old construction used a single iterator over `Cat` which ranged over the absolute

flat positions belonging to segment k, so the segment-start case had to be detected by checking whether the absolute iterator was equal to the boundary expression $e(k)$.

The new construction makes the local structure explicit. Line 1 introduces a fresh iterator i, which is the local offset inside the segment. Lines 5–7 compute the expressions needed for the flattened representation. The expression $e(k+1)$ is obtained by substituting $k+1$ for k in the boundary expression e. The segment length is $e(k+1) - e(k)$, and the absolute flat position represented by the local offset i is $e(k) + i$.

Line 9 defines the segment-start guard. Since i is now a local offset, the start of a segment is simply the case $i = 0$. This is the same condition that was previously expressed indirectly through the absolute index and the lower boundary of the `Cat` segment. The guard c' is retained because it records the in-bounds condition for the boundary branch, unless it has already been simplified to `true`.

Lines 13–14 are the actual removal of `Cat`. The shape of the result is now

$$[[\texttt{Forall}\ k\ (\texttt{Iota}\ m), \texttt{Forall}\ i\ (\texttt{Iota}\ (e(k+1) - e(k)))]],$$

which corresponds to the flattened domain $k : 0..m \times i : 0..(e(k+1) - e(k))$. This is still one flattened dimension, not an ordinary two-dimensional array.

The body has two cases. At the beginning of a segment, line 16 takes the value from the scattered value array at position k. At all other local offsets, line 17 keeps the original destination value at the absolute flat position $e(k) + i$. Finally, line 21 substitutes the temporary placeholders with the actual index functions `vs` and `xs`.

Effect of the change. The refactored rule describes the same scatter result as before, but it exposes the segmented structure in the representation used by the rest of the flattened-domain machinery. This makes the result easier for later rules to consume.

Validation. The refactored `scatterSc2` rule was tested both directly and through program-level regressions. The direct tests check that the rule no longer produces `Cat` domains and instead produces a shape with a flattened dimension. The program-level tests check that programs depending on this rule, such as `mk_flag_array.fut`, also infer index functions without `Cat`. These tests are representation-oriented: they focus on the structure of the inferred index functions rather than on a specific pretty-printed output. The full testing setup is discussed in Section 7.2.

6.3 Substitution Across Flattened Domains

After PropFlatten has propagated a flattened domain, later rewrite steps still need to substitute applications of index functions that live over such domains. This is the purpose of `substituteOnce`. The function replaces an occurrence of an index-function application by the body of the applied index function. For ordinary one-dimensional domains, this is mostly a direct substitution. For flattened domains, however, a flat argument must first be interpreted as a pair consisting of an outer segment index and an inner local offset.

The formal rule behind this step is SubFlat, which was introduced in Section 5.4.2. The rule rewrites an application $x(e_{idx})$, where x has a flattened domain, by expressing the flat index e_{idx} in the coordinates of that domain. The outer coordinate is written using the symbolic index operation $\%_D(e_{idx})$, where D is the flattened domain. The inner coordinate is obtained by subtracting the start of the selected row from the flat index. This section explains how that idea is realised in the implementation.

6.3.1 Implementing SubFlat with `from1Dto2DM`

Figure 15 shows the formal SubFlat rule together with the implementation helper that constructs the corresponding argument substitution. The code excerpt is taken from `from1Dto2DM`, which is called from `substituteOnce` when one flat argument must be matched against a two-iterator flattened domain.

SubFlat

$$\frac{\Gamma(x) = \lambda\,(i_1 : 0..e_2 \times i_2 : 0..e_3)\;.\;e_x \qquad \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_3[i_1 := j]) \rightsquigarrow e_{row}}{\Gamma \vdash \mathcal{C}\langle x(e_{idx})\rangle \to \mathcal{C}\langle e_x[i_1 := \%_{i_1:0..e_2\times i_2:0..e_3}(e_{idx}), i_2 := e_{idx} - e_{row}[i_1 := \%_{i_1:0..e_2\times i_2:0..e_3}(e_{idx})]]\rangle}$$

(a) Formal rule

```
from1Dto2DM :: Quantified Domain -> Quantified Domain -> SoP Symbol
               -> IndexFnM [(VName, SoP Symbol)]
from1Dto2DM (Forall i1 (Iota e2)) (Forall i2 (Iota e3)) e_idx
  | i1 `S.notMember` fv e3 =
      let outer = sym2SoP (Ix e2 e3 e_idx)
       in pure [(i1, outer), (i2, e_idx .-. outer .*. e3)]

  | otherwise = do
      j <- newVName "j"
      let ub    = sym2SoP (Var i1) .-. int2SoP 1
      let e3_j  = rep (mkRep i1 (sym2SoP (Var j))) e3
      let eRow  = sumSoP j (int2SoP 0) ub e3_j
      let outer = sym2SoP (Ix e2 e3 e_idx)
      let eRowOuter = rep (mkRep i1 outer) eRow
      pure
        [ (i1, outer)
        , (i2, e_idx .-. eRowOuter)
        ]
from1Dto2DM _ _ _ = undefined
```

(b) Implementation

Figure 15: The SubFlat rule and the implementation helper that converts a flat argument into flattened coordinates.

The purpose of `from1Dto2DM` is to construct replacements for the two iterators of a flattened domain. In the code, the flattened domain has the form $i_1 : 0..e_2 \times i_2 : 0..e_3$, and the flat index being substituted is e_{idx}. The result is a list of replacements for i_1 and i_2.

Lines 4–6 handle the regular case, where the inner bound e_3 does not depend on the outer iterator i_1. The outer coordinate is represented by the symbolic expression $\%_D(e_{idx})$, implemented as `Ix e2 e3 e_idx`. Since all rows have length e_3, the row start is $\%_D(e_{idx}) \cdot e_3$. The local offset is therefore $e_{idx} - \%_D(e_{idx}) \cdot e_3$. This is the regular version of the coordinate conversion used by SubFlat.

Lines 8–18 handle the irregular case. Here e_3 may depend on i_1, so the row start cannot be computed by multiplication. Lines 10–12 construct the prefix-sum row start. The outer coordinate is still represented by $\%_D(e_{idx})$. Lines 13–18 then compute the row start for that symbolic outer coordinate and use it to define the local offset as

$$e_{idx} - e_{row}[i_1 := \%_D(e_{idx})].$$

The idea is the same as in the regular case: the local offset is the flat index minus the start of the row that contains it. The difference is only how the row start is computed.

6.3.2 Connecting `from1Dto2DM` to `substituteOnce`

The helper `from1Dto2DM` does not perform the full substitution by itself. It only constructs the replacement needed when an application to a flat source-level index must be matched against a flattened two-iterator domain. The rest of `substituteOnce` uses this replacement to inline the body of f into the surrounding index function g.

The relevant control flow is shown in Figure 16. The excerpt is split into two parts. The first part shows where `substituteOnce` uses the argument replacement and then performs the rest of the substitution. The second part shows the branch of `mkArgs` that constructs this replacement for flattened domains.

The call `args <- mkArgs` in Figure 16a constructs a substitution from the formal iterators of f to the actual arguments used at the call site. The details of this construction are shown in Figure 16b. There are two relevant cases. If the number of actual arguments matches the total number of iterators in shape(f), then the replacement is direct: each formal iterator is paired with the corresponding actual argument.

The flattened case is handled by the `rank f == length actual_args` branch. Here the number of actual arguments matches the number of top-level dimensions, not the total number of iterators. This is the situation where a source-level dimension is flat, but the index function for f exposes that same dimension as a flattened domain with two iterators. The branch calls `mkArgM` once for each top-level dimension. If the dimension contains one iterator, `mkArgM` returns the direct replacement $i := a$. If the dimension contains two iterators, `mkArgM` calls `from1Dto2DM` to split the single flat argument a into two replacements: one for the outer segment iterator and one for the local offset.

This is where the implementation follows SubFlat. The rule replaces the outer iterator by the symbolic segment-selection expression $\%_D(e_{idx})$, and the inner iterator by the offset of e_{idx} inside that segment. In the implementation, `from1Dto2DM` constructs these replacements. In the regular case, the row start is $\%_D(e_{idx}) \cdot e_3$. In the irregular case, the row start is the prefix sum up to the symbolic outer coordinate.

```
args <- mkArgs

let g_sub =
      g
        { shape = new_shape,
          body = cases $ do
            (p_f, e_f) <- guards f
            (p_g, e_g) <- guards g
            let s = mkRep vn (rep args e_f)
            pure (sop2Symbol (rep s p_g) :&& sop2Symbol (rep args p_f),
                  rep s e_g)
        }

mg1 <- applySubRules args g_sub

mg2 <- case mg1 of
  Nothing -> pure Nothing
  Just g1 -> Just <$> solveIx (shape g1) g1

mg3 <- traverse simplify mg2
mg3' <- traverse cleanupIndexFnM mg3
pure mg3'
```

(a) Main substitution flow in `substituteOnce`

```
mkArgs =
  case actual_args of
    _
      | length actual_args == length (concat (shape f)) ->
          pure $
            mconcat $
              zipWith
                (\(Forall i _) a -> mkRep i a)
                (concat (shape f))
                actual_args

      | rank f == length actual_args -> do
          reps <- zipWithM mkArgM (shape f) actual_args
          pure $ mconcat reps

      | otherwise ->
          error "Argument mismatch."

mkArgM [Forall i _] a =
  pure $ mkRep i a

mkArgM [d1, d2] a =
  mkRepFromList <$> from1Dto2DM d1 d2 a
```

(b) The relevant `mkArgs` cases

Figure 16: Selected implementation flow connecting `substituteOnce`, `mkArgs`, and `from1Dto2DM`. The first excerpt shows where the replacement is used; the second excerpt shows how the replacement is constructed in the ordinary and flattened cases.

Once `args` has been constructed, `substituteOnce` uses it to build the substituted index function g_{sub}. For each guarded case (p_f, e_f) of f and each guarded case (p_g, e_g) of g, it substitutes the actual arguments into e_f, replaces the occurrence of the application

inside g, and conjoins the corresponding guards. This is how the implementation handles the expression context $\mathcal{C}\langle\cdot\rangle$ from the rule: the occurrence being substituted may appear inside a larger body, and the guards from both index functions must still be preserved.

After this initial substitution, `substituteOnce` calls `applySubRules`. This gives rules such as PropFlatten an opportunity to adjust the domain of the result before index expressions are simplified. The ordering matters because simplifying an index expression can depend on the current domain. If the result can inherit a flattened domain first, later simplification can use the segment iterator and local offset directly.

The next step is the call to `solveIx`. The replacement created by `from1Dto2DM` may contain symbolic expressions of the form $\%_D(e_{idx})$. These expressions allow substitution to proceed without immediately solving which segment contains e_{idx}. The call to `solveIx` then tries to simplify them using the current shape of the index function. This step is described in the next subsection.

Finally, the result is simplified and cleaned up. The cleanup step runs index solving again under the assumptions of each guard and removes impossible or duplicate branches.

6.3.3 Resolving Symbolic Segment Indices with `solveIdx1`

The previous implementation subsection showed how `substituteOnce` and `from1Dto2DM` implement the SubFlat step. As discussed in Section 5.4.2, SubFlat may introduce symbolic expressions of the form $\%_D(e_{idx})$. Such an expression means that the analysis has a flat index e_{idx} into a flattened domain D, but has not yet resolved which segment contains that index.

The framework overview introduced SolveIdx1 as well in Section 5.4.2. That rule handles the interior case: if the solver can prove that the flat index lies inside the row described by the current outer iterator, then the symbolic segment expression can be replaced by that iterator. The implementation of this idea is the function `solveIdx1`.

The rule does not compute the segment number by division. Instead, it proves a bounds condition. For a flattened domain with outer iterator i_1, inner bound e_2, and row start e_{row}, the key condition is: $e_{row} \leq e_{idx} < e_{row} + e_2$. If this condition holds, then e_{idx} is inside the current row, so $\%_D(e_{idx})$ can be simplified to i_1.

Figure 17 shows the formal SolveIdx1 rule together with the corresponding implementation. The code has two branches: one for regular flattened domains, where every row has the same length, and one for irregular flattened domains, where the row length may depend on the outer iterator.

$$\textsc{SolveIdx1}\quad \frac{\text{unify}\,(i_1 : 0..e_1 \times i_2 : 0..e_2,\ D) = \sigma \qquad \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_2[i_1 := j]) \rightsquigarrow e_{row} \qquad (\Gamma, \mathsf{Range}\ i_1\ 0..e_1, \mathsf{Range}\ i_2\ 0..e_2) \vdash \text{Query}\ (0 \le \sigma(e_{idx}) - e_{row} < e_2) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes}}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ \mathcal{C}\langle \%_D(e_{idx})\rangle \to \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ \mathcal{C}\langle i_1\rangle}$$

(a) Formal rule

```haskell
solveIdx1 :: [Quantified Domain] -> Symbol -> IndexFnM Symbol
solveIdx1 dim@[Forall i1 (Iota _), Forall _ (Iota e2)]
          sym@(Ix _ m e_idx)
  | i1 `S.notMember` fv e2 = rollbackAlgEnv $ do
      addRelDim dim
      dimensions_match <- e2 `unifiesWith` m
      q <-
        if dimensions_match
          then Bool True =>?
            (sVar i1 .*. e2 :<= e_idx
             :&& e_idx :< (sVar i1 .+. int2SoP 1) .*. e2)
          else pure Unknown
      case q of
        Yes -> pure (Var i1)
        Unknown -> pure sym

  | otherwise = rollbackAlgEnv $ do
      addRelDim dim
      dimensions_match <- e2 `unifiesWith` m
      eRow <- mkERow i1 e2
      q <-
        if dimensions_match
          then Bool True =>? (eRow :<= e_idx :&& e_idx :< eRow .+. e2)
          else pure Unknown
      case q of
        Yes -> pure (Var i1)
        Unknown -> pure sym
```

(b) Implementation

Figure 17: The SolveIdx1 rule and the corresponding implementation. The implementation resolves $\%_D(e_{idx})$ by proving that the flat index lies inside the current row.

The implementation starts by matching the current domain against a two-iterator flattened dimension. The expression being simplified must be an `Ix` expression, which is the implementation form of $\%_D(e_{idx})$.

The regular branch handles the case where e_2 does not depend on i_1. Then all rows have the same length, so row i_1 starts at $i_1 \cdot e_2$ and ends before $(i_1 + 1) \cdot e_2$. The implementation asks the solver to prove that the flat index e_{idx} lies in this interval:

$$i_1 \cdot e_2 \le e_{idx} < (i_1 + 1) \cdot e_2.$$

If this query succeeds, the `Ix` expression is replaced by i_1. If the solver returns `Unknown`, the expression is left unchanged.

The irregular branch handles the segmented case. Here the inner bound e_2 may

depend on i_1, so the start of row i_1 is no longer a multiplication. Instead, the implementation uses `mkERow` to compute the prefix-sum row start. The solver then checks whether the flat index lies between this row start and the end of the current row:

$$e_{row} \leq e_{idx} < e_{row} + e_2.$$

If this holds, then e_{idx} is inside the row described by i_1, so the symbolic segment index can be replaced by i_1.

The check `dimensions_match` is used in both branches. It verifies that the inner dimension stored inside the `Ix` expression matches the inner dimension of the current flattened domain. This prevents the implementation from solving an `Ix` expression using the wrong domain information.

The important difference between the two branches is therefore only how the row start is computed. In the regular case, the row start is $i_1 \cdot e_2$. In the irregular case, it is the prefix sum of all previous segment lengths. In both cases, the implementation follows the same idea as SolveIdx1: prove that the flat index belongs to the current row, and then replace the symbolic segment index by the current outer iterator.

6.3.4 Handling Boundary Cases After Substitution

The implementation also contains a special case for solving `Ix` expressions when the surrounding index function is one-dimensional. This case is not a separate formal rule. It is an implementation detail needed because substitution can inline information from a flattened array into an index function that is not itself flattened.

An `Ix` expression is an internal helper used by the analysis to recover the outer segment from a flat index. For a flattened segmented domain, it asks: which segment contains this flat position? For example, if the segment lengths are

$$\texttt{shape} = [2, 3, 1],$$

then flat positions 0 and 1 belong to segment 0, positions $2, 3, 4$ belong to segment 1, and position 5 belongs to segment 2. Thus, for the corresponding flattened domain D, the expression $\%_D(4)$ should simplify to 1.

The usual `solveIdx1` case handles this when the current index function already has a flattened two-iterator domain, such as

$$k : 0..m \times i : 0..\texttt{shape}(k).$$

In that setting, the current domain already contains both the segment iterator k and the local offset i, so the solver can ask whether the flat index lies inside the current segment.

The extra branch is needed for a slightly different situation. Sometimes the current result is one-dimensional, but its body contains an expression that came from indexing into a flattened array. This is what happens for `lst`, as we have previously shown in the `part2indicesL` example presented in Figure 10. The array `lst` has one element per segment, so its domain is only $k : 0..m$.

However, in the non-empty segment case, `lst` is computed by reading the last value of `indsT` in segment k:

$$\texttt{indsT}(\texttt{ends}(k) - 1).$$

The array `indsT` is flattened and segmented, so substituting its index function introduces an `Ix` expression. The surrounding result is still one-dimensional, but the expression inside its body remembers that it came from a flattened segmented array. Before this branch was added, `solveIdx1` did not handle that combination: it expected the current shape itself to be flattened, so it left the `Ix` expression unresolved.

The relevant implementation is shown in Figure 18.

```
solveIdx1 dim@[Forall k (Iota n)] sym@(Ix n' e_inner e_idx)
  | Just (Apply _ [arg]) <- justSym e_inner,
    Just (Var old_k) <- justSym arg = rollbackAlgEnv $ do
      addRelDim dim

      dimensions_match <- n `unifiesWith` n'

      let e_inner_k =
            rep (mkRep old_k (sym2SoP (Var k))) e_inner

      eRow <- mkERow k e_inner_k

      let lower = eRow :<= e_idx
      let upper = e_idx :< eRow .+. e_inner_k

      q <-
        if dimensions_match
          then Bool True =>? (lower :&& upper)
          else pure Unknown

      case q of
        Yes -> pure (Var k)
        Unknown -> pure sym
```

Figure 18: One-dimensional boundary case in `solveIdx1`. The code solves an `Ix` expression from a previous flattened representation while simplifying under a one-dimensional domain.

The branch applies when the current domain has the form $k : 0..n$, but the expression being simplified is still an `Ix` expression. The `Ix` expression stores an inner bound, such as $\texttt{shape}(old\_k)$, from the flattened domain where it was introduced. Since the current domain uses k as its iterator, the implementation first replaces the old outer iterator with k. This turns the stored inner bound into the current segment length, for example $\texttt{shape}(k)$.

After this replacement, the implementation computes the start of the current segment. It then asks the solver whether the flat index e_{idx} lies inside the current segment:

$$e_{row} \leq e_{idx} < e_{row} + \texttt{shape}(k).$$

If this can be proved, then the segment containing e_{idx} is the current segment k, so the

`Ix` expression can be replaced by k. If the solver cannot prove the bounds, the expression is left unchanged.

For `lst`, the flat index is $\texttt{ends}(k) - 1$. Since $\texttt{ends}(k) - 1 = \sum_{j=0}^{k} \texttt{shape}(j) - 1$, this is the last flat position of segment k. Under the guard $\texttt{shape}(k) \neq 0$, that position is inside segment k. Therefore the symbolic expression $\%_D(\sum_{j=0}^{k} \texttt{shape}(j) - 1)$ can be simplified to k.

The purpose of the branch is to handle this kind of mixed context. The `Ix` expression still comes from a flattened segmented array, but it appears while simplifying a one-dimensional index function. By reusing the current iterator k and proving that the flat index lies inside its segment, the implementation can resolve the symbolic segment number and continue simplifying the body.

6.3.5 Summary of the Substitution Changes

Together, the updates to `substituteOnce`, `from1Dto2DM`, and `solveIdx1` make substitution work across flattened irregular domains. `substituteOnce` performs the high-level replacement of index-function applications. `from1Dto2DM` explains how a flat actual argument should be split into outer and inner coordinates for a flattened domain. `solveIdx1` then removes the symbolic $\%_D(e_{idx})$ terms when the solver can prove which row the flat index belongs to.

This is important for the later segmented examples because PropFlatten alone only changes the domain of an index function. Once flattened domains have been propagated, the analysis still needs to substitute through applications over those domains and simplify the resulting index expressions. These implementation changes provide that missing link: they allow the analysis to move from a flat expression containing symbolic indexing operations to an index function written directly in terms of the segment iterator and local offset.

6.4 Row-Wise `InvFiltPart` Properties over Flattened Domains

The final implementation change we want to present is supporting row-wise inverse filtering and partitioning properties over flattened domains. Earlier examples used the older property `FiltPartInv`, which describes an inverse filtering/partitioning index array but does not include an explicit image range. The formal development instead uses `InvFiltPart`, which includes a range Z. For a one-dimensional array, a property of the form

$$\texttt{InvFiltPart}\ x\ Z\ p_f\ (p_1, \ldots, p_n)$$

states that x is an inverse filtering/partitioning index array over the range Z, with filtering predicate p_f and partition predicates $p_1, \ldots, p_n$.

For segmented arrays, the intended property is not one global partition over the whole flat array. Instead, each segment should satisfy the property over its own flat interval. This is expressed using `For`. In the implementation, a property of the form $\texttt{For}\ x\ (\lambda k.\ P)$ means that P should be proved for each outer row of x. The range of k is obtained from the outer dimension of x's inferred index function.

This is the form used in `part2indicesL` (The running example we presented in 5.4). Figure 19 compares the earlier flat specification with the updated row-wise specification.

```
-- Previous global flat property.
FiltPartInv inds
  (\_i -> true)
  (\i -> csL[i])
```

(a) Previous specification

```
-- New row-wise property.
let seg_starts =
  map (\k ->
    if k == 0i64
    then 0i64
    else seg_ends[k-1])
  (iota (m + 1))
in For inds (\k ->
  InvFiltPart inds
    (seg_starts[k], seg_starts[k+1])
    (\_i -> true)
    (\i -> csL[i]))
```

(b) Updated specification

Figure 19: Old and new postcondition style for `part2indicesL`. The old specification used one global flat property, while the updated version states the inverse filtering/partitioning property separately for each segment.

The older specification tried to describe the segmented result with a single flat property. The updated version instead states the property row by row. The interval $(\texttt{seg\_starts}[k], \texttt{seg\_starts}[k+1])$ is the flat range belonging to segment k. Thus, the property says that the indices produced in each segment form an inverse filtering/partitioning index array for that segment.

6.4.1 Implementing the Flattened `For` Case

The formal rule used for this case is For-InvFiltPart-Flat, shown in Figure 20. The rule applies when the array being proved about has a flattened two-iterator domain $i_1 : 0..e_1 \times i_2 : 0..e_2$, but the property should hold separately for each segment. Its purpose is to reduce the row-wise property to an ordinary one-dimensional `InvFiltPart` proof for the current segment.

FOR-INVFILTPART-FLAT

$$\frac{\begin{array}{c}\Gamma(x_1) = \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ e_x \\ \text{fresh } x_2, j \qquad \Gamma \vdash \text{Query}\left(e_3 = \sum_{i_1=0}^{e_1-1}(e_2)\right) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes} \\ \Gamma, \text{Range } k\ 0..e_1 \vdash \lambda\,(i_2 : 0..e_2)\ .\ p_f[i_3 := \sum_{j=0}^{k-1}(e_2[i_1 := j]) + i_2] \rightsquigarrow f_{p_f} \\ \Gamma, \text{Range } k\ 0..e_1 \vdash \lambda\,(i_2 : 0..e_2)\ .\ p_1[i_3 := \sum_{j=0}^{k-1}(e_2[i_1 := j]) + i_2] \rightsquigarrow f_{p_1} \\ \vdots \\ \Gamma, \text{Range } k\ 0..e_1 \vdash \lambda\,(i_2 : 0..e_2)\ .\ p_n[i_3 := \sum_{j=0}^{k-1}(e_2[i_1 := j]) + i_2] \rightsquigarrow f_{p_n} \\ \Gamma, \text{Range } k\ 0..e_1, x_2 \mapsto \lambda\,(i_2 : 0..e_2)\ .\ e_x[i_1 := k] \vdash \mathsf{InvFiltPart}\ x_1\ Z\ f_{p_f}\ (f_{p_1}, \ldots, f_{p_n})\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{For}\ (k : 0..e_1)\ (\mathsf{InvFiltPart}\ x_1\ Z\ (\lambda\,(i_3 : 0..e_3)\ .\ p_f)\ (\lambda\,(i_3 : 0..e_3)\ .\ p_1, \ldots, \lambda\,(i_3 : 0..e_3)\ .\ p_n)) \\ \quad \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{For}\ (k : 0..e_1)\ (\mathsf{InvFiltPart}\ x_1\ Z\ (\lambda\,(i_3 : 0..e_3)\ .\ p_f)\ (\lambda\,(i_3 : 0..e_3)\ .\ p_1, \ldots, \lambda\,(i_3 : 0..e_3)\ .\ p_n))), \mathsf{Yes})\end{array}}$$

(a) Formal rule

```haskell
matchProof (For x (Predicate i p)) = do
  f_x <- getFn x
  case f_x of
    IndexFn [[Forall k d_outer, Forall i2 (Iota e2)]] body_x ->
      rollbackAlgEnv $ do
        addRelIterator (Forall k d_outer)

        let p_k =
              mapProperty
                (sop2Symbol . rep (mkRep i (sym2SoP (Var k))))
                p

        case p_k of
          InvFiltPart x1 z pf pps
            | x1 == x -> do
                e_row <- mkERow k e2

                x_row <- newNameFromString "#for_flat_row"
                let f_row =
                      IndexFn
                        { shape = [[Forall i2 (Iota e2)]],
                          body = body_x
                        }

                let pf_row  = localiseFlatPred i2 e_row pf
                let pps_row = map (localiseFlatPred i2 e_row) pps

                insertIndexFn x_row [f_row]
                prove $ InvFiltPart x_row z pf_row pps_row

          _ ->
            pure Unknown
```

(b) Implementation

Figure 20: The FOR-INVFILTPART-FLAT rule and the corresponding implementation case. The implementation reduces a row-wise property over a flattened domain to a one-dimensional `InvFiltPart` proof for the current segment.

The implementation first looks up the index function for x. The special case applies when this index function has a flattened two-iterator shape, represented as

$$[[\texttt{Forall}\ k\ d_{outer}, \texttt{Forall}\ i_2\ (\texttt{Iota}\ e_2)]].$$

Here k is the segment iterator, and i_2 is the local iterator inside the current segment. This corresponds to the first premise of the formal rule, where the array being proved about has a flattened domain. The call to `addRelIterator` adds the range of k to the solver environment. This is needed because the property body may contain expressions such as `seg_starts`$[k]$ and `seg_starts`$[k+1]$, and the solver must know that k is a valid segment index.

The source-level `For` binder is then replaced by the actual outer iterator from the inferred shape. If the body is an `InvFiltPart` property for the same array, the implementation constructs a row-local proof obligation. This corresponds to the fresh row-local array in the formal rule: the implementation creates `x_row`, drops the outer dimension, and keeps only the local domain $i_2 : 0..e_2$.

The remaining step is to adapt the predicates to this row-local view. For example, in `part2indicesL`, the predicate `(\i -> csL[i])` refers to positions in the original flat array. Inside one segment, the local iterator is i_2, so `localiseFlatPred` rewrites the predicate so that i_2 is interpreted as the corresponding flat position in segment k. The resulting row-local property is then passed to the ordinary one-dimensional `InvFiltPart` prover.

6.4.2 Relation to the `InvFiltPart` Prover

The flattened `For` case does not introduce a new proof procedure for inverse filtering and partitioning. Instead, it translates the flattened row-wise property into the one-dimensional form already handled by the `InvFiltPart` prover. This is also why adding `InvFiltPart` as a separate property was useful: the older `FiltPartInv` property captured the same general idea, but did not include the explicit range Z needed to state the property for one segment at a time.

Figure 21 shows the relevant parts of the proof implementation. The first excerpt shows that `prove` first checks whether the property is already known, and otherwise continues with `matchProof`. The second excerpt shows the main checks performed by the one-dimensional `InvFiltPart` prover.

The `alreadyKnown` branch is used when a matching `InvFiltPart` property has already been recorded in the environment. It looks up a stored property for the same array and tries to unify it with the property currently being proved. If the unification succeeds, the proof can finish without expanding the full set of obligations again.

```haskell
prove :: Property Symbol -> IndexFnM Answer
prove prop = alreadyKnown prop `orM` matchProof prop

alreadyKnown wts@(InvFiltPart y _ _ _) = do
  res <- askInvFiltPart (Algebra.Var y)
  case res of
    Just (InvFiltPart y' z' pf' pps')
      | y' == y -> do
          s <- unify wts =<< fromAlgebra (InvFiltPart y' z' pf' pps')
          if isJust (s :: Maybe (Substitution Symbol))
            then pure Yes
            else pure Unknown
    _ -> pure Unknown
```

(a) Checking whether `InvFiltPart` is already known

```haskell
prove_ baggage (PInvFiltPart (za, zb) pf pps') f@(IndexFn [[Forall i dom]] _) =
  algebraContext f $ do
    let img = (za, zb .-. int2SoP 1)
    step1 <- prove_ baggage (PBijectiveRCD img img) f

    step2 <- rollbackAlgEnv $ do
      addRelShape (shape f)
      allM [if_filtered_then_OOB g | g <- guards f]

    j <- newNameFromString "j"
    let filtered_guards = [(c :&& pf i, e) | (c, e) <- guards f]
    step3 <- newProver (MonGe LT i j dom (cases filtered_guards))

    let step4 = allM [isParted p q | p : pp <- tails pps, q <- pp]

    pure step1 `andM` step2 `andM` step3 `andM` step4
```

(b) Main checks in the one-dimensional `InvFiltPart` prover

Figure 21: Selected parts of the `InvFiltPart` proof implementation. The flattened `For` case delegates to this prover after constructing a row-local index function.

When the property is not already known, the prover checks the one-dimensional `InvFiltPart` property directly. The first step proves that the index function is bijective over the requested image range.

The second step checks the filtering part of the property. If an index is filtered away by the filter predicate, then the produced value must lie outside the image range. The third step checks that the kept values preserve their order. The fourth step checks the ordering imposed by the partition predicates. Together, these obligations are the ordinary one-dimensional proof that an index array implements inverse filtering and partitioning over the requested range.

The flattened `For` case uses this prover after translating the row-wise flattened problem into this one-dimensional form. In other words, For-InvFiltPart-Flat uses the flattened domain to select one segment, rewrites the predicates so they refer to local positions in that segment, and then relies on the ordinary `InvFiltPart` prover. This is the step that allows `part2indicesL` to state its intended postcondition directly as a per-segment property while the program itself still stores all segments in one flat array.

7 Evaluation

7.1 Evaluating PropFlatten

The implementation of `propFlattenOnce` was evaluated with direct unit tests that isolate the rewrite from the rest of the compiler pipeline. This is useful because PropFlatten is a local transformation: it takes two index functions, checks whether one can inherit the flattened domain of the other, and then rewrites the body of the first index function. End-to-end tests over complete `.fut` programs are still necessary, but they make it harder to see whether a failure comes from PropFlatten itself or from another inference rule used earlier or later in the analysis.

The tests therefore construct small synthetic index functions directly and call the function on them. This gives a focused check of the implementation described in Section 6.1. In each test, the expected result is not only that the function succeeds, but also that the resulting index function has the right flattened domain and that the old flat iterator has been replaced by the correct flat-position expression.

Regular case. The first test checks the regular case, where the inner bound of the flattened domain does not depend on the outer iterator. The test constructs an index function g with a flat dimension

$$i_1 : 0..(e_2 \cdot e_3),$$

and an index function f whose corresponding dimension is already flattened:

$$i_2 : 0..e_2 \times i_3 : 0..e_3.$$

Since e_3 is independent of i_2, every segment has the same length. The expected row-start expression is therefore $i_2 \cdot e_3$, and the old flat iterator should be rewritten as

$$i_1 := i_2 \cdot e_3 + i_3.$$

The test checks that `propFlattenOnce` returns a rewritten index function rather than failing, that the selected dimension of g has been replaced by the flattened dimension from f, and that the body of g uses $i_2 \cdot e_3 + i_3$ in place of i_1. This confirms that the regular branch preserves the previous rectangular behavior.

Irregular case. The second test checks the irregular case, where the inner bound may depend on the outer iterator. This is the case needed for segmented arrays. The test constructs a source flattened domain of the form

$$i_2 : 0..m \times i_3 : 0..\texttt{shape}(i_2),$$

and a target flat domain whose size is the total number of elements across all segments:

$$i_1 : 0.. \sum_{j=0}^{m-1} \texttt{shape}(j).$$

Here there is no single constant row length, so the row start must be expressed as a prefix sum over the previous segment lengths.

The expected rewrite is

$$i_1 := \sum_{j=0}^{i_2-1} \texttt{shape}(j) + i_3.$$

The test checks that `propFlattenOnce` succeeds, that the resulting shape contains the flattened dimension

$$i_2 : 0..m \times i_3 : 0..\texttt{shape}(i_2),$$

and that the body uses the prefix-sum row start plus the local offset. This directly exercises the new branch of the implementation, where `mkERow` constructs the row-start expression and the solver checks that the flat size equals the total segmented size.

What the tests show. These tests do not prove PropFlatten correct in general, as this was already done by existing tests. Instead, they check the two implementation paths that matter for this thesis: the regular case, where the row start is multiplication, and the irregular segmented case, where the row start is a symbolic prefix sum. They also make sure that the function updates both parts of the index function: the shape is changed to the flattened domain, and the body is rewritten under the new coordinates.

The main value of these tests is that they isolate the rule from the rest of the inference pipeline. A failure in one of these tests points directly to the implementation of `propFlattenOnce`, while failures in larger program-level tests may involve several interacting rules. The direct unit tests therefore complement the program-level regression tests discussed later in this chapter.

7.2 Evaluating the `scatterSc2` Migration

The second part of the implementation changed the segmented scatter rule so that `scatterSc2` no longer produces `Cat` domains. Since this was a representation change, the tests were designed to check the structure of the inferred index functions rather than a specific printed form. This is important because pretty-printed index functions may change due to unrelated simplifications, while the relevant property for this refactor is whether the inferred domains still contain `Cat`.

The tests do not claim that the `Cat` constructor has been removed from the codebase. They check the more specific migration goal: the scatter rule, and the selected programs that exercise it, should not produce inferred output domains containing `Cat`.

We used two small structural predicates on inferred index functions. The first predicate, `hasCat`, checks whether any domain in the shape still uses `Cat`. The second predicate, `hasFlattenedDim`, checks whether at least one dimension contains more than one iterator. Together, these predicates capture the intended migration for `scatterSc2`: the rule should stop producing `Cat`, and it should instead expose segmented structure as a flattened dimension.

Direct rule test. The first test applies the segmented scatter rule directly to small synthetic index functions. The input is chosen so that one guarded branch of the scatter index function is clearly in bounds and monotonic, while another branch behaves as an out-of-bounds case. This isolates the rule from the rest of the inference pipeline and makes the expected representation change easier to inspect.

The test checks two conditions on the inferred result. First, no domain in the resulting index function may use `Cat`. Second, the resulting shape must contain a flattened dimension. These two checks verify that the rule has been migrated from the old segmented encoding to the new flattened representation.

Program-level regression test. The second test checks that the same property holds when the rule is used as part of a complete program analysis. In particular, we test programs that depend on the segmented scatter rule, such as `mk_flag_array.fut`. The test runs index-function inference on the relevant value bindings and checks that none of the inferred index functions contain `Cat`.

This complements the direct rule test. The direct test checks the local behavior of `scatterSc2`, while the program-level regression checks that the same representation is preserved when the rule is reached through the normal inference pipeline. Together, the tests give confidence that this migrated rule no longer produces `Cat`, while still preserving the segmented structure needed by later rules.

7.3 Program-Level Regression Tests

In addition to the focused tests for `propFlattenOnce` and `scatterSc2`, we also ran the existing program-level index-function test suite. These tests are useful because they exercise the complete inference pipeline rather than a single rule in isolation. A program-level test may use several interacting rules: conversion rules for `map`, `scan`, and `scatter`, property propagation, substitution, index solving, and algebraic simplification.

The current test run passes 42 out of 43 program tests. This is important because the implementation changes described in this thesis affect central parts of the rewriting pipeline, especially substitution and scatter inference. The passing tests include both older one-dimensional examples and the segmented examples discussed earlier in the thesis. In particular, `part2indices.fut`, corresponding to the one-dimensional running example in Section 4.2, still passes. The simpler segmented example `mk_flag_array.fut`, discussed in Section 5.3, also passes. Most importantly for this thesis, `part2indicesL.fut`, the larger segmented running example from Section 5.4, now passes as well.

Table 3 lists the most relevant program-level tests. The table includes the main examples used throughout the thesis, the remaining failing test, and several additional programs that exercise filtering, scatter-based compaction, irregular segmented arrays, and maximal matching.

Program	Status	Time	What the test exercises
`part2indices.fut`	✓	0.21s	One-dimensional partition-index construction.
`part2indicesL.fut`	✓	1.21s	Segmented partition-index construction over flat irregular arrays.
`mk_flag_array.fut`	✓	0.26s	Scatter-based construction of segment-start flags.
`filter.fut`	✓	0.19s	Standard filter implemented through prefix sums and scatter.
`filter_irreg.fut`	✓	0.67s	Filtering over a segmented flat array using segment identifiers.
`maxMatch.fut`	✓	1.09s	Maximal-matching step over flattened edge arrays.
`maxMatch_2d.fut`	✓	1.52s	Maximal-matching step with explicit two-dimensional edge structure.
`seg_partition.fut`	×	–	Segmented partition followed by scatter; currently fails scatter safety.

Table 3: Selected program-level regression tests after the implementation changes.

The passing of `part2indicesL.fut` is especially relevant because it depends on the interaction of several of the implemented changes. It requires flattened-domain propagation, substitution through flattened index functions, simplification of symbolic segment-index expressions, and the removal of `Cat` from the segmented scatter construction. Therefore, this regression test provides evidence that the individual implementation changes work together on a realistic segmented example, not only in isolated unit tests.

The additional filter and maximal-matching tests give further coverage of the parts of PropProp affected by the migration. The standard `filter.fut` test checks the usual filter encoding, where prefix sums construct an index array and `scatter` compacts the selected values into the result. The `filter_irreg.fut` test uses the same filtering idea in a segmented setting: it first expands a shape array into segment identifiers and then filters a flat array using a per-segment predicate. This makes it relevant for checking that segmented structure can still be inferred and consumed after the `Cat` migration.

The two maximal-matching tests exercise a larger application-style use case. Both contain filtering and scatter-based updates over graph edge arrays. The `maxMatch.fut` version works over flattened edge arrays, while `maxMatch_2d.fut` keeps the edge structure explicit as two-element rows in parts of the program and uses flattening where needed. These tests are useful because they check that the changes made for segmented irregular programs did not break existing reasoning about filtering, injectivity, and scatter in larger benchmarks.

One test remains failing: `seg_partition.fut`. During the implementation work, this test was updated from the older `FiltPartInv` postcondition form to the newer bounded form using `For` and `InvFiltPart`, following the style used for `part2indicesL`. This also required adding missing implementation cases for propagating and representing this property form. After these changes, the property is stored in the environment in the expected form, namely as a per-segment inverse filtering/partitioning property:

$$\texttt{For inds}\ (\lambda k.\ \texttt{InvFiltPart inds}\ (\texttt{seg\_start}(k), \texttt{seg\_end}(k))\ \cdots).$$

The remaining failure occurs later, when the analysis tries to infer an index function for a `scatter`. The error is not that the segmented property is absent, but that the scatter-safety check cannot yet use the new bounded `InvFiltPart` property to prove the required injectivity or bijectivity property of the scatter indices. In the previous implementation, there were special cases for the older `FiltPartInv` form that allowed the prover to reuse it as evidence for injectivity or bijectivity. The corresponding support for the newer `For–InvFiltPart` form has not yet been completed.

Thus, the remaining failing test points to a limitation in property-to-property reasoning for the updated annotation form, rather than to the flattened-domain representation itself. Extending the prover so that bounded `InvFiltPart` properties imply the scatter-safety facts needed by `scatterSafe` is left as future work.

8 Related Work

As discussed in the previous chapters, this thesis builds on PropProp [9], a compiler-based analysis for verifying selected properties of arrays in pure data-parallel programs. PropProp works by translating array computations into index functions and then proving properties such as range, injectivity, bijectivity, filtering, and partitioning. The related work in this section is discussed from that perspective. Some systems verify array programs with more general proof techniques, while others verify compiler transformations or use index-array properties for dependence analysis. PropProp is less general than a full theorem prover, but it is more automatic for the class of scan- and scatter-based programs considered in this thesis.

8.1 General-Purpose Verification of Array Programs

General-purpose verification systems such as Dafny [11], F* [15], and Liquid Haskell [17] can express a wide range of program properties, but they typically ask the programmer to supply definitions, invariants, or lemmas before a proof goes through. The difficulty is most visible in data-parallel programs that first compute an index array and then use it in a scatter or a gather. The properties that matter are facts about the index array itself: whether each entry is in bounds, whether the array is injective or bijective, and whether it describes a filter or a partition. Establishing them by hand is precisely the burden these tools impose.

A concrete illustration is the two-way stable partition that this thesis derives automatically as `part2indices` in Section 4.2. The same algorithm is studied in the PropProp work [9], where it serves as a case study in Dafny. Dafny verifies the local properties of the computed index array on its own, for example that both the passing and the failing positions form strictly increasing sequences and that every passing position precedes every failing position. Verifying the partition postcondition, however, requires the programmer to state and apply auxiliary inductive lemmas, including one that relates the prefix sums of the two complementary flag arrays and one that establishes monotonicity of a prefix sum over non-negative values. Coming up with these lemmas is not mechanical, and it is the part of the proof that a non-expert is least likely to manage.

That study also shows how brittle such proofs are under small changes to the code. Computing the failing-side positions with a reverse prefix sum instead of a forward one produces a proof obligation in which the quantified indices appear inside the bounds of the summed slices, and even after introducing a recursive definition of summation this query could not be discharged in Dafny. The scatter itself is harder again: Dafny cannot establish the partition result without a manual assumption that the scatter indices form a permutation of the output positions, which defeats the automation the tool is meant to provide.

Refinement-type systems share this profile. Liquid Haskell discharges refinement obligations through an SMT solver, building on liquid types [13], and refinement reflection [18] additionally lets source functions appear in refinements. Even so, proofs for

the class of programs considered here still rely on user-written lemmas. As reported in the same evaluation [9], showing that a small `map`-then-`zipWith` pipeline produces positive integers needed a hand-written lemma to recover the positional correlation that the bulk-parallel style discards, while the data-parallel partition and the three-way and segmented batch variants were harder still or out of reach. F* [15] combines dependent and refinement types with both SMT automation and interactive proving, and it automates much of the term-level reasoning, but the array-content properties at issue here remain manual proof work.

A separate line of systems restricts the type language to keep checking decidable. Dependent ML [20] confines dependent values to a constraint language amenable to static array bounds checking, ATS [19] admits explicit proof terms at the cost of interleaving proof and program, and Qube [16] targets array indexing and shape matching. These systems are effective within their scope, but that scope is linear indexing, whereas the programs studied in this thesis build their indices through prefix sums and scatters and therefore rely on non-linear indexing.

PropProp [9] sits at the opposite end of this trade-off. It does not attempt arbitrary program properties and instead fixes a small set of array properties that recur in data-parallel code, which is what lets the compiler carry out the proofs without user lemmas. The flattened irregular programs that this thesis targets are precisely the case where the general-purpose tools struggle most, since the segment structure is hidden in auxiliary arrays and the relevant properties must hold per segment. The contributions of the previous chapters extend PropProp's automation to that case, so that the segmented analogue of the partition above is verified with no more user input than the regular one.

8.2 Verification of Parallel Programs and Compiler Transformations

Several related systems reason about array and tensor programs, but they usually focus on a different problem from the one considered in this thesis. Some verify that compiler rewrites preserve program meaning. Others verify low-level GPU properties such as memory safety and data-race freedom. PropProp instead focuses on a smaller set of source-level array properties, and tries to prove them automatically from inferred index functions. This is the main point of comparison in the following paragraphs.

8.2.1 TensorRight

TensorRight [1] provides automated reasoning for tensor compiler rewrites. It verifies rank- and shape-polymorphic tensor graph rewrites, and its core language is modeled after tensor compiler intermediate representations such as XLA's High Level Operators. Its evaluation includes rewrite rules from XLA's algebraic simplifier [3]. This makes TensorRight useful for checking that tensor rewrite rules are valid for tensors of arbitrary rank and size.

The difference is that TensorRight proves equivalence between two tensor expressions, while PropProp proves properties of arrays computed by a source program. This matters when the important question is not whether two expressions are equal, but whether a

computed index array is safe to use later. For example, PropProp can prove that an index array is in range, injective, or describes a filter or partition.

TensorRight and PropProp also treat reductions and flattening differently. TensorRight includes HLO-style operators such as `reduce`, but general reductions over unbounded axes are represented abstractly using uninterpreted reduction elements, with special support for some cases. PropProp gives a more direct interpretation to `scan`, which is relevant for Futhark because scans are often used in segmented computations. TensorRight also does not target layout-sensitive operators such as reshape and bitcast, while PropProp supports flattened dimensions through rules such as Flatten and PropFlatten.

8.2.2 ATL

ATL [12] verifies transformations of tensor programs through user-guided scheduling rewrites. Programs are written in an index-based notation with array generations, sums, guards, and explicit indexing expressions. Optimizations are expressed as rewrite theorems in Coq, and the programmer applies these rewrites step by step to derive an optimized program.

This is related to PropProp because both systems use symbolic reasoning about indices. However, ATL starts from an index-centric language, while PropProp starts from a data-parallel source language and infers the index representation as part of the analysis. This inference is important for operations such as `scatter`, where the analysis first needs to know facts about the index array before it can construct a useful index function.

ATL also has reshape operators such as `flatten` and `split`. These operators express regular changes in loop and storage structure during scheduling. PropProp uses flattened domains for a different purpose: to describe the structure of arrays that appear in source programs. In this thesis, the important case is not only flattening a regular tensor, but recovering segmented structure from a flat array whose shape is described by auxiliary arrays.

8.2.3 Collective operations from low-level code

Essertel et al. [4] translate low-level imperative programs into a pure functional language with collective forms such as sequence construction and summation. Variables, heap allocations, and loop iterations are modeled as symbolic sequences. This allows the analysis to recover high-level structure from low-level code, and it can be used for verification, equivalence checking, and translation of legacy C code to higher-level DSLs.

The similarity to PropProp is that both approaches replace step-by-step execution with a more mathematical representation. The difference is the starting point. Essertel et al. recover collective operations from imperative loops and heap updates. PropProp starts from pure data-parallel programs and uses index functions to prove selected array properties. Their collective language includes sequences and sums, but it does not target the irregular index arrays constructed by scans and scatters.

8.2.4 Descend

Descend [10] is a safe GPU systems programming language. It is closer to CUDA than to Futhark: the programmer still controls memory layout, thread scheduling, and low-level GPU execution. Descend adds safety by tracking ownership, lifetimes, execution resources, and memory views in the type system. In this way, it can reject programs with unsafe memory accesses, data races, or incorrect synchronization.

This is different from PropProp's goal. Descend checks whether low-level GPU memory accesses are safe. PropProp checks semantic properties of arrays before lowering to low-level GPU code. For example, Descend can check that different GPU threads access disjoint parts of an array through structured views. PropProp instead tries to prove that the index array used by an operation has the required property.

The distinction is especially clear for irregular programs. Descend works best when the programmer can express the memory access pattern using structured views. For graph-like programs, where an index may be read from memory at runtime, Descend may require `unsafe` code. PropProp addresses a different setting: pure data-parallel programs where irregular indices are computed by the program itself.

8.2.5 VerCors verification of CUDA prefix sum and stream compaction

Safari and Huisman [14] verify CUDA implementations of parallel prefix sum and stream compaction using the VerCors [2] verifier. Their work proves both data-race freedom and partial functional correctness. The verification is done at the CUDA level, using permission-based separation logic, annotations, ghost variables, and auxiliary pure functions.

This work is closely related because stream compaction is one of the motivating examples for PropProp. A typical compaction algorithm computes a prefix sum over a flag array and then uses the prefix sums as output positions:

```
if flag[i] then output[prefix[i]] = input[i].
```

To verify this at the CUDA level, Safari and Huisman prove extra properties about the prefix sums, the flag array, and a pure `filter` function. For example, they show that the prefix-sum values used for flagged elements are valid output indices, and that the final output corresponds to filtering the input by the flags.

This comparison illustrates the main tradeoff. VerCors can verify low-level CUDA implementations with strong guarantees, but it requires detailed proof structure from the user. PropProp is less general, but it targets properties such as filtering and partitioning directly as source-level postconditions.

8.3 Dependence Analysis and Index-Array Properties

Index-array properties are also important in dependence analysis and safety checking. For irregular programs, an analysis may need to know whether an index array is in range, injective, bijective, monotone, or otherwise structured. Such facts can justify parallel

execution, remove runtime checks, or show that operations such as scatters and gathers are safe. This is one reason why PropProp's property set is useful.

A related problem is bounds checking for array programs. Henriksen and Oancea [7] present a hybrid bounds-checking analysis for Futhark. Their analysis extracts the bounds checks from the original program and turns them into a separate predicate, or inspector, that is evaluated before the main computation. If the predicate succeeds, then the original computation can run without the corresponding checks. For example, for an indirect access such as `x[is[i]]`, the extracted predicate checks that all values used from `is` are within the bounds of `x`. The predicate may itself be a parallel computation, for example a `map` followed by a `reduce` with conjunction.

This approach is similar in spirit to inspector-executor techniques: the inspector checks whether the indexes are safe, and the executor performs the actual computation. The difference from PropProp is that this check is still dynamic. The compiler may simplify the predicate, and in many cases it may become cheap or even disappear, but if the compiler cannot prove it statically, the program still has to run the predicate at runtime. PropProp instead tries to prove selected index-array properties statically from the data-parallel source program. In that sense, PropProp can be seen as a complementary static approach: when it succeeds, the property is established without running a separate inspector.

Henriksen [5] studies a lower-level version of the same safety problem: how to implement bounds checking efficiently in GPU code generated from high-level array languages. Since CUDA and OpenCL do not provide convenient support for safely aborting a running kernel, the compiler-generated code records failures in GPU memory and reports them back to the host in a controlled way. The implementation is designed so that the non-failing case remains efficient, and the evaluation reports low overhead for checked Futhark programs.

This work addresses runtime safety rather than static property verification. It ensures that out-of-bounds accesses are reported safely during execution, while PropProp aims to prove before execution that certain index arrays have the required properties. The two approaches therefore sit at different points in the design space. Runtime bounds checking is more broadly applicable, because it can catch errors even when the compiler cannot prove safety. Static verification is more limited, but when it succeeds it avoids runtime checks and gives stronger information to later compiler reasoning. The segmented and flattened case studied in this thesis is an example where such static reasoning is difficult: the relevant structure may be hidden in flat arrays, and must be recovered before properties such as range, injectivity, filtering, or partitioning can be proved.

9 Conclusion and Future Work

The motivation set out in Section 1 is that programs operating on flat irregular arrays hide their segmented structure in auxiliary arrays such as `shape`, `flags`, and `offsets`. A verifier that wants to reason about per-segment behavior must first recover this structure. The PropProp framework [9] already includes a flattened-domain representation that exposes the segment iterator and the local offset directly. However, the existing implementation only supported this representation for regular flattened arrays. Irregular segmented arrays were still represented through the `Cat` constructor, which records segmented structure indirectly through absolute interval boundaries.

The contrast between these two representations, and the reason the flattened one is preferable for irregular segmented arrays, is laid out in Section 5.2. This thesis closes the gap between framework and implementation by extending PropFlatten to the irregular case and migrating the remaining rules that produced `Cat`.

Contributions. The thesis makes four implementation-level contributions to PropProp.

First, Section 6.1 generalizes the implementation of the PropFlatten rule to the irregular case, where the inner bound of a flattened dimension may depend on the outer iterator. In the regular case, the row start can be written as $i_2 \cdot e_3$, and the size check is $e_1 = e_2 \cdot e_3$. In the irregular case, the row start must instead be computed as the prefix sum $\sum_{j=0}^{i_2-1} e_3[i_2 := j]$, and the flat size must be checked against the total length of all segments. This is implemented in `propFlattenOnce`. The result is that flattened structure can be propagated not only for regular arrays, but also for segmented arrays whose rows have different lengths.

Second, Section 6.2 migrates the segmented scatter rule so that it constructs flattened-domain results directly instead of producing `Cat`-based domains. The rule already recognized a boundary expression describing segment starts. The updated implementation uses this expression to construct a flattened dimension with a segment iterator and a local offset. In this representation, the beginning of a segment is simply the case $i = 0$, rather than an equality between an absolute flat iterator and a symbolic boundary. Since this scatter rule is the main place where segmented structure is recovered from flat source programs, this change is central to removing `Cat` from the system.

Third, Section 6.3 updates substitution and index solving so that later rewrites can consume the flattened irregular domains produced by PropFlatten and the segmented scatter rule. The function `substituteOnce` performs the high-level substitution of index-function applications. The helper `from1Dto2DM` turns a flat argument into an outer segment coordinate and an inner local offset, using a prefix-sum row start in the irregular case. The function `solveIdx1` then simplifies symbolic segment-index expressions by proving that a flat index lies inside the current segment. These changes are necessary because changing the domain alone is not enough: the analysis must also be able to substitute through that domain and simplify the expressions that result.

Fourth, Section 6.4.1 adds support for row-wise `InvFiltPart` properties over flat-

tened domains. The implementation already had the older `FiltPartInv` property, but it did not carry the explicit range Z used by the formal `InvFiltPart` property. The new support makes it possible to state that each segment satisfies an inverse filtering/-partitioning property over its own flat interval. The flattened `For` case reduces this row-wise property to an ordinary one-dimensional `InvFiltPart` proof by constructing a row-local index function and rewriting the predicates into that row-local view. This is the property-layer change needed for `part2indicesL` to state its intended postcondition directly as a per-segment property.

Effects of the migration. With irregular segmented arrays now represented in the same flattened form as regular ones, several pieces of the analysis become more direct. Per-segment recurrences are recognized by SegRecSum in the same way that the one-dimensional RecSum rule recognizes a global scan, without requiring the solver to detect equalities on absolute boundaries. Indexing operations expressed in segmented coordinates can be substituted via SubFlat in terms of segment number and local offset, with SolveIdx1 resolving the symbolic segment number by proving that the flat index lies within the current row, in both its flattened form and the one-dimensional case that arises when the surrounding index function is not itself flattened. The framework's SolveIdx2 and SolveIdx3 resolve indices that fall syntactically on the start or end of a segment. Empty segments are handled uniformly through empty iterator ranges. In the old representation, recognizing an empty segment required the solver to detect that two symbolic boundary expressions were equal, while in the flattened representation an empty segment is just an empty iterator range and no guard is needed. This removes a class of solver-side work rather than just a representational quirk.

The derivation of `part2indicesL` in Section 5.4 illustrates these mechanisms end to end. It uses flattened-domain propagation, segmented scatter reconstruction, substitution across flattened domains, symbolic index solving, and the row-wise `InvFiltPart` property. Section 7 evaluates these effects through direct rule tests and program-level regressions.

Limitations. The rules and the implementation handle a flattened dimension consisting of exactly two iterators (one segment iterator and one local offset), and an array containing at most one such flattened dimension. Deeper nesting within a single flattened dimension is not supported: the rules would need to match more than two iterators in the same flattened dimension, although the choice of how to rewrite would remain unambiguous. Arrays with multiple flattened dimensions are also not supported, and are harder to address because more than one dimension could in principle inherit the flattened structure, so the rule would need a heuristic to choose between them. The latter is noted in the PropProp [9] appendix as an open issue.

A separate limitation appears in the current program-level regression suite. All tests except `seg_partition.fut` pass after the migration. The remaining failure is not caused by a missing flattened-domain representation: the newer per-segment `For`–`InvFiltPart` property is inferred and stored, but the scatter-safety check does not yet use this property

form to derive the injectivity or bijectivity facts required for the scatter. Thus, the failure points to a missing property-propagation case rather than to the `Cat`-to-flattened-domain migration itself.

9.1 Future work

The most direct follow-up is to re-examine the rules in the property and algebra layers in light of `Cat`'s removal. Several of these rules were written to recover segmented structure from absolute boundaries, and may admit shorter formulations now that the segment iterator and local offset are exposed directly. The migration of `scatterSc2` in Section 6.2 already illustrates this on a small scale: a guard that previously compared an absolute iterator to a symbolic boundary became the case $i = 0$. Whether similar simplifications are available elsewhere in the analysis is left open.

A second concrete follow-up is to complete the property-layer support needed by `seg_partition.fut`. The program now uses the newer bounded `For`–`InvFiltPart` property form, and the property is recorded in the environment, but `scatterSafe` still needs injectivity or bijectivity evidence for the scatter index array. The older implementation contained special cases that reused the older `FiltPartInv` form for such facts. Adding the corresponding reasoning for bounded `InvFiltPart` properties would likely make this benchmark pass again.

A third direction is to lift the dimensionality restrictions noted above. The implementation of `propFlattenOnce` already takes a top-level dimension index as an argument, which positions it for the multiple-flattened-dimensions case, but the logic for choosing which dimension should inherit the flattened structure has not been worked out. Supporting deeper nesting within a single flattened dimension is a separate extension: the grammar in Section 5 already permits the required domain structure, but the rule and implementation patterns would need updating to match more than two iterators.

References

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

[2] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. ISBN: 978-3-319-66845-1.

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

[4] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. "Precise reasoning with structured time, structured heaps, and collective operations". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360583. URL: https://doi.org/10.1145/3360583.

[5] Troels Henriksen. "Bounds Checking on GPU". In: *International Journal of Parallel Programming* (Mar. 2021). DOI: 10.1007/s10766-021-00703-4.

[6] Troels Henriksen and Martin Elsman. "Towards size-dependent types for array programming". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2021, pp. 1–14. DOI: 10.1145/3460944.3464310.

[7] Troels Henriksen and Cosmin E. Oancea. "Bounds Checking: An Instance of Hybrid Analysis". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 88:88–88:94. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627388. URL: http://doi.acm.org/10.1145/2627373.2627388.

[8] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "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 (PLDI)*. 2017, pp. 556–571. DOI: 10.1145/3140587.3062354.

[9] Nikolaj Hey Hinnerskov, Robert Schenck, and Cosmin Eugen Oancea. *Verifying Properties of Index Arrays in a Purely-Functional Data-Parallel Language*. 2025. arXiv: 2506.23058 [cs.PL]. URL: https://arxiv.org/abs/2506.23058.

[10] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. "Descend: A Safe GPU Systems Programming Language". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656411. URL: https://doi.org/10.1145/3656411.

[11] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[12] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. “Verified tensor-program optimization via high-level scheduling rewrites”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[13] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid Types”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[14] Mohsen Safari and Marieke Huisman. “Formal verification of parallel prefix sum and stream compaction algorithms in CUDA”. In: *Theoretical Computer Science* 912 (2022), pp. 81–98. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2022.02.027. URL: https://www.sciencedirect.com/science/article/pii/S0304397522001232.

[15] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. “Dependent Types and Multi-Monadic Effects in F*”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655.

[16] Kai Trojahner and Clemens Grelck. “Dependently Typed Array Programs Don’t Go Wrong”. In: *The Journal of Logic and Algebraic Programming* 78.7 (2009), pp. 643–664. DOI: 10.1016/j.jlap.2009.03.002.

[17] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161.

[18] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. “Refinement Reflection: Complete Verification with SMT”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 53:1–53:31. DOI: 10.1145/3158141.

[19] Hongwei Xi. “Applied Type System: An Approach to Practical Programming with Theorem-Proving”. In: *arXiv preprint arXiv:1703.08683* (2017). arXiv: 1703.08683 [cs.PL].

[20] Hongwei Xi. “Dependent ML: An Approach to Practical Programming with Dependent Types”. In: *Journal of Functional Programming* 17.2 (2007), pp. 215–286. DOI: 10.1017/S0956796806006216.