Appendix

This document contains supplementary material for the paper *Verifying Array Properties in Pure Data-Parallel Programs*.

Contents

A Source language

x, y, z			Variables
F			Function variables
$n \in \mathbb{Z}$			Constants
β	::=	i64 | f64 | bool | ...	Base types
τ	::=	β | $[]\tau$	Array types
$\prec$	::=	$< \mid \leq \mid > \mid \geq$	Linear orders
Op	::=	$\prec \mid + \mid - \mid * \mid = \mid \neq \mid \wedge \mid \vee$	Binary operators
B	::=		Base expressions
	|	x	Variable
	|	n	Constant
	|	$\lvert x \rvert$	Length
	|	2^B	Power of 2
	|	$B \;\; Op \;\; B$	Binary operation
	|	$x_{\text{array}}[\overline{B_{\text{index}}}]$	Array index
E^π	::=	B | Sum $x[E^\pi : E^\pi]$	Property expressions
π	::=	Range $x\ E^\pi..E^\pi$ | Mono $x \prec$ | Equiv $x\ E^\pi$ | Inj $x\ E^\pi..E^\pi$	Properties
	|	Bij $x\ E^\pi..E^\pi\ E^\pi..E^\pi$ | InvFiltPart $x\ E^\pi..E^\pi\ (\lambda x.\, E^\pi)\ (\overline{\lambda x.\, E^\pi})$	
	|	FiltPart $x\ x\ (\lambda x.\, E^\pi)\ (\overline{\lambda x.\, E^\pi})$ | Filt $x\ x\ (\lambda x.\, E^\pi)$ | Part $x\ x\ (\lambda x.\, E^\pi)$	
	|	For $(i : E^\pi..E^\pi)\ \pi$	
$E^\varnothing$	::=	B	Base expression
	|	$B..B$	Sequence
	|	map $(\lambda \overline{x}^{(n)}.\, E)\ \overline{x_{\text{array}}}^{(n)}$	Map
	|	scan $(\lambda \overline{x_1}^{(n)}\ \overline{x_2}^{(n)}.\, E)\ \ \overline{B}^{(n)}\ \overline{x_{\text{array}}}^{(n)}$	Scan
	|	scatter $x_{\text{dst}}\ x_{\text{idx}}\ x_{\text{val}}$	Scatter
	|	$F\ \overline{x}$	Function application
	|	if B then E else E	Conditional
	|	**loop** $(\overline{x}^{(n)}) = (\overline{x_{init}}^{(n)})$ **while** x_n **do** $F\ \overline{x}^{(n)}$	While loop
	|	**loop** $(\overline{x}^{(n)}) = (\overline{x_{init}}^{(n)})$ **for** $x_{n+1} < B$ **do** $F\ \overline{x}^{(n+1)}$	For loop
E	::=	**let** $\overline{x} = E^\varnothing$ **in** E | $\overline{x}$	Expressions
Fun	::=	**def** $F\ \overline{(x : \tau \mid \overline{\pi})} : \ (\tau \mid \lambda \overline{x}.\, \overline{\pi}) = E$	Function definition
$Prog$	::=	ϵ | $Fun\ \ Prog$	Programs

We write $\overline{X}^{(n)}$ to denote a sequence of objects $X_1, \ldots, X_n$ and we may just write $\overline{X}$ if the length is not important or is clear from the context. Sequences are separated by white space or by comma, as dictated by the context. We use F exclusively for the names of function definitions. We use Greek letters to denote types and properties and capital letters to denote terms. Variadic uses of map and

scan result in multiple results (i.e., structure of arrays rather than array of structures):

$$\textbf{let } ys_1, ys_2 = \text{map } (\lambda x_1\ x_2.\ \textbf{let } y_1 = x_1 + x_2 \textbf{ in let } y_2 = x_1 - x_2 \textbf{ in } y_1, y_2)\ xs_1\ xs_2 \textbf{ in } ys_1, ys_2.$$

B Formalization

B.1 Prerequisites

B.1.1 *Index function grammar.*

$$\begin{array}{rrcl}
\text{Variable} & x, y, z, i, j, k & & \\
\text{Constant} & n, m \in \mathbb{Z} & & \\
\text{Symbol} & s & ::= & [p] * e \mid x \mid |x| \mid \sum_{x=e}^{e}(e) \mid x(e) \mid x^{-1}(e) \mid p \mid 2^e \mid \circlearrowleft \\
\text{Predicate} & p & ::= & x \mid x(e) \mid \text{true} \mid \text{false} \mid \neg p \mid e \le e \mid p \wedge p \mid p \vee p \mid \circlearrowleft \\
\text{Term} & t & ::= & n \mid s \mid s \cdot t \\
\text{Expression} & e & ::= & t \mid t + e \\
\text{Domain} & D & ::= & i : 0..e \\
\text{Index fun.} & f & ::= & \lambda\ (D)\ .\ e
\end{array}$$

Index functions have the form $\lambda\ (D)\ .\ e$ where D is the domain and e is a guarded expression. Scalars are single element arrays: $\lambda\ ()\ .\ g$ abbreviates $\lambda\ (i : 0..1)\ .\ g$. Comparisons are syntactic sugar: $e_1 \in 0..e_2$ and $0 \le e_1 < e_2$ are both $(0 \le e_1) \wedge (e_1 + 1 \le e_2)$. Inference rules match to expressions and index functions via unification with bound variables [60].

For clarity, we initially only treat 1D arrays, so index function domains only have one iterator variable and indexing only takes one argument.

We may write sums of terms using both cases and $\sum$ notation (note that this is distinct from the $\sum_{x=e}^{e}(e)$ symbol) depending on which is clearer in the given context:

$$t_1 + \ldots + t_n = \sum_{j=1}^{n} t_j = \begin{cases} t_1 \\ \vdots \\ t_n \end{cases}$$

B.1.2 *Query answer grammar.*

$$\text{Answer } A ::= \text{Yes} \mid \text{Unknown}$$

Queries either return Yes, which means the query holds, or Unknown, which means the query could not be verified by our system.

B.1.3 Properties. We distinguish between source-level properties π and index function properties P. The grammar for P copies the grammar for π, but replaces property expressions E^π for index function expressions e and predicate lambdas $(\lambda x.\ E^\pi)$ for index functions f with predicate p bodies. Properties Filt and Part are removed and instead expressed using FiltPart.

$$\begin{array}{rcl}
P & ::= & \text{True} \mid \text{Range } x\ e..e \mid \text{Mono } x \prec \mid \text{Equiv } x\ e \mid \text{Inj } x\ e..e \mid \text{Bij } x\ e..e\ e..e \\
& \mid & \text{InvFiltPart } x\ e..e\ (\lambda\ (i : 0..e)\ .\ p)\ \overline{(\lambda\ (i : 0..e)\ .\ p)} \\
& \mid & \text{FiltPart } x\ x\ (\lambda\ (i : 0..e)\ .\ p)\ \overline{(\lambda\ (i : 0..e)\ .\ p)} \\
& \mid & \text{For } (i : e..e)\ P
\end{array}$$

All constructors except For are base properties. Each property in P targets a single variable x, which structurally corresponds to the first argument of the base property. We use the notation $P(x)$ to project the target variable x generically.

True is a trivial property that always holds, but carries no information. Our formalization assumes that all function arguments are annotated with properties; arguments without property annotations in the source code are given the True property during parsing.

B.1.4 Environments. Environments (Γ) map variables to index functions and properties via subenvironments Γ_{Ixfn} for index functions, Γ_{Range} for ranges, and so on for each property in P. Unbound variables map to $\varnothing$. We write $\Gamma, x \mapsto f$ to extend Γ_{Ixfn} mapping x to f, and Γ, Range x $0..e$ to extend Γ_{Range} mapping x to $0..e$, etc.

Notation	Definition
$\Gamma, x \mapsto f$	Γ with $\Gamma_{\text{Ixfn}}\{x \mapsto f\}$
Γ, Range x $e_1..e_2$	Γ with $\Gamma_{\text{Range}}\{x \mapsto e_1..e_2\}$
Γ, Mono $x \prec$	Γ with $\Gamma_{\text{Mono}}\{x \mapsto \prec\}$
...	
Γ, For $(i : e_1..e_2)$ $P(x)$	Γ with $\Gamma_{\text{For}}\{x \mapsto (i : e_1..e_2, P)\}$

The subenvironment Γ_{Def} tracks function definitions. We write:

$$\Gamma \text{ with } \Gamma_{\text{Def}}(F) = ((x_1, \overline{P_1}) \ \ldots \ (x_n, \overline{P_n}), (\overline{y}^{(q)}, \overline{P_{post}}), \overline{f}^{(q)})$$

to map a function F to its formal arguments and preconditions $(x_i, \overline{P_i})$, its return variables and postconditions $(\overline{y}, \overline{P_{post}})$, and—optionally—its index functions $\overline{f}$. If no index function is inferred for a particular return y_i, we put $(n, \varnothing)$ in place of an index function where n is the rank of the return. In inference rule premises, we may write _ to match bindings for F with or without index functions: $\Gamma_{\text{Def}}(F) = ((x_1, \overline{P_1}) \ \ldots \ (x_n, \overline{P_n}), (\overline{y}^{(q)}, \overline{P_{post}}), \_)$.

Finally, the set-valued subenvironment Γ_{Pred} tracks arbitrary predicates that cannot be directly expressed as properties (e.g., $0 < |x|$). We again use standard comma-extension, writing $\Gamma, e_1 = e_2, 0 \le i < j < n$ to add predicates to Γ_{Pred}. These are later lowered to ranges and equivalences in the algebra layer (Appendix B.5).

B.2 The Algorithm

The overall algorithm converts programs to index functions while verifying pre- and postconditions in a back-and-forth interaction between the index function, property and algebra layers.

We formalize the algorithm using inference rules. First we give an overview of the relations used:

Notation	Definition	
$\Gamma \vdash e \rightsquigarrow e'$	$\Gamma \vdash e \to e'_1, \quad \Gamma \vdash e'_1 \to e'_2, \quad \ldots, \quad \Gamma \vdash e'_n \to e'$	(No rule applies to e')
$\Gamma \vdash f \rightsquigarrow f'$	$\Gamma \vdash f \to f'_1, \quad \Gamma \vdash f'_1 \to f'_2, \quad \ldots, \quad \Gamma \vdash f'_n \to f'$	(No rule applies to f')
$\Gamma \vdash B \rightsquigarrow f'$	$\Gamma \vdash B \to f, \quad \Gamma \vdash f \rightsquigarrow f'$	
$\Gamma \vdash E^{\varnothing} \rightsquigarrow (\Gamma', f'_1, \ldots, f'_n)$	$\Gamma \vdash E^{\varnothing} \to (\Gamma', f_1, \ldots, f_n), \quad \Gamma' \vdash f_1 \rightsquigarrow f'_1, \quad \ldots, \quad \Gamma' \vdash f_n \rightsquigarrow f'_n$	
$\Gamma \vdash E \rightsquigarrow (\Gamma', f'_1, \ldots, f'_n)$	$\Gamma \vdash E \to (\Gamma', f_1, \ldots, f_n), \quad \Gamma' \vdash f_1 \rightsquigarrow f'_1, \quad \ldots, \quad \Gamma' \vdash f_n \rightsquigarrow f'_n$	
$\Gamma \vdash \text{Query}_x\,(e) \rightsquigarrow_{\text{Q}} A$	$\Gamma \vdash e \rightsquigarrow e', \quad \Gamma \vdash \text{Query}_x\,(e') \to_{\text{Q}} A$	

The judgment $\Gamma \vdash e \rightsquigarrow e'$ rewrites expressions to a fixed point (that is, no rewrite rules apply to e'). Similarly for $\Gamma \vdash f \rightsquigarrow f'$. Relations from the source language to index functions f correspond to one source-to-index-function step, disambiguated by type (e.g., $B \to f$), followed by one or more rewrite steps $f \to f'$ until a fixed point is reached. $\text{Query}_x\,(e) \rightsquigarrow_{\text{Q}} A$ uses the $e \rightsquigarrow e'$ relation to rewrite the query expression before attempting to solve it.

B.2.1 Verifying programs ($\Gamma \vdash Prog \to$ Verified). The analysis starts with a program, which consists of function definitions, and an empty environment Γ. The judgment $\Gamma \vdash Prog \to$ Verified says that all annotations are verified and that indexing is within bounds in the program *Prog* under environment Γ.

We give the rule for a function definition with one argument, one precondition, one return and one postcondition below. We also assume that the formal argument is a one-dimensional array. The rule is straightforward to generalize, but it is filled with ancillary detail.

$$\boxed{\Gamma \vdash \mathit{Prog} \longrightarrow \mathsf{Verified}}$$

FunDef

$$\frac{\begin{array}{c}\Gamma \vdash \pi_{pre} \rightarrow_{\mathsf{Parse}} P_{pre} \qquad \Gamma \vdash \pi_{post} \rightarrow_{\mathsf{Parse}} P_{post} \\ \text{fresh } i \qquad \Gamma, P_{pre} \vdash \lambda\,(i : 0..|x|)\,.\,x(i) \rightsquigarrow f_1 \\ \Gamma, P_{pre}, x \mapsto f_1, \vdash E_{body} \rightsquigarrow (\Gamma', f_2) \\ \mathrm{fv}(f_2) \subseteq \mathrm{dom}(\Gamma_{\mathsf{Def}}) \cup \{x\} \\ \Gamma', y \mapsto f_2 \vdash P_{post} \rightarrow_{\mathsf{Prop}} (\Gamma'', \mathsf{Yes}) \\ \Gamma \text{ with } \Gamma_{\mathsf{Def}}(F) = ((x, P_{pre}), (y, P_{post}), f_2) \vdash \mathit{Prog} \longrightarrow \mathsf{Verified}\end{array}}{\Gamma \vdash \textbf{def } F\ (x : \tau_1 \mid \pi_{pre}) : \tau_2 \mid \lambda y.\, \pi_{post} = E_{body}\ \ \mathit{Prog} \longrightarrow \mathsf{Verified}}$$

The two first premises parse the source-level properties π into index function properties P. We don't formalize the $\rightarrow_{\mathsf{Parse}}$ relation.

The third premise creates a fresh variable i and uinterpreted index function for the formal argument. The $\rightsquigarrow$ relation rewrites this function under Γ extended with the property P_{pre}.

The fourth premise binds the formal argument to its precondition and index function and then derives an index function for the function body (Appendix B.2.2). Only preceding function definitions are in scope for the duration of this analysis.

The fifth premise checks that the resulting index function is expressed only in terms of F's formal arguments.[1] The sixth premise uses the property layer ($\rightarrow_{\mathsf{Prop}}$) to verify the postcondition (Appendix B.4). Finally, the seventh premise extends Γ with information about F before treating the rest of the program, where F is in scope.

If the postcondition cannot be proved, we terminate the analysis in failure.

FunDefFail

$$\frac{\begin{array}{c}\Gamma \vdash \pi_{pre} \rightarrow_{\mathsf{Parse}} P_{pre} \qquad \Gamma \vdash \pi_{post} \rightarrow_{\mathsf{Parse}} P_{post} \\ \text{fresh } i \qquad \Gamma, P_{pre} \vdash \lambda\,(i : 0..|x|)\,.\,x(i) \rightsquigarrow f_1 \\ \Gamma, P_{pre}, x \mapsto f_1 \ \vdash\ E_{body} \rightsquigarrow (\Gamma', f_2) \\ \Gamma', y \mapsto f_2 \vdash P_{post} \rightarrow_{\mathsf{Prop}} (\Gamma'', \mathsf{Unknown})\end{array}}{\begin{array}{r}\Gamma \vdash \textbf{def } F\ (x : \tau_1 \mid \pi_{pre}) : \tau_2 \mid \lambda y.\, \pi_{post} = \\ E_{body}\ \ \mathit{Prog} \longrightarrow \textbf{Error}\end{array}}$$

On empty programs, we're done:

EndOfProgram

$$\frac{}{\Gamma \vdash \epsilon \longrightarrow \mathsf{Verified}}$$

B.2.2 Converting let expressions ($\Gamma \vdash E \longrightarrow (\Gamma', \overline{f})$). A function's body is defined by non-nested let bindings of the form $\textbf{let } \overline{x} = E^{\oslash} \textbf{ in } E$. We convert $E^{\oslash}$ to index functions and also infer properties from $E^{\oslash}$, binding the results to the names $\overline{x}$, before converting the body E.

[1] A separate rule for function definition records F's index function as $\oslash$ in Γ_{Def} when this is not the case.

The judgment $\Gamma \vdash E \rightarrow (\Gamma', \overline{f})$ says that, under environment Γ, E has index functions $\overline{f}$ and produces environment Γ'.

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

Let

$$\frac{\Gamma \vdash E^{\oslash} \rightsquigarrow (\Gamma'_1, f_1, \ldots, f_n) \qquad \Gamma'_1 \vdash (x_1, \ldots, x_n, E^{\oslash}) \rightarrow_{\mathsf{Infer}} \Gamma'_2 \qquad \Gamma'_2, x_1 \mapsto f_1, \ldots, x_n \mapsto f_n \vdash E \rightsquigarrow (\Gamma'_3, f'_1, \ldots, f'_n)}{\Gamma \vdash \textbf{let } x_1, \ldots, x_n = E^{\oslash} \textbf{ in } E \rightarrow (\Gamma'_3, f'_1, \ldots, f'_n)}$$

Both conversion (the first premise) and property inference (the second premise) may add properties to the environment. However, the grammar for $E^{\oslash}$ ensures that no new index function bindings are introduced. For example, if $E^{\oslash}$ applies a function, the environment is extended with that function's preconditions in the first premise and its postconditions in the second premise (bound to the relevant names).

The bindings are kept in the returned environment because they are eventually used to check the function's postcondition in the function definition rule. Function definitions, map and scan may contain let expressions and their corresponding rules are responsible for discarding bindings as they go out of scope.

For the remainder of the document, we will assume that let expressions only bind one name for clarity. Extending the rules to handle multiple bound/returned values is uncomplicated.

B.2.3 Converting expressions. The rule for function application checks that the actual arguments satisfy the function's preconditions via the property layer, adding them to Γ one by one. If an index function was inferred for the applied function, it is returned after substituting the actual arguments for the formal arguments. Otherwise, an uninterpreted function is created.

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

App

$$\frac{\Gamma_{\mathsf{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), f) \qquad \Gamma \vdash P_1[x'_1 := x_1] \rightarrow_{\mathsf{Prop}} (\Gamma'_1, \mathsf{Yes}) \qquad \ldots \qquad \Gamma'_{n-1} \vdash P_n[x'_1 := x_1, \ldots, x'_n := x_n] \rightarrow_{\mathsf{Prop}} (\Gamma'_n, \mathsf{Yes}) \qquad \Gamma'_n \vdash f[x'_1 := x_1, \ldots, x'_n := x_n] \rightsquigarrow f'}{\Gamma \vdash F\ x_1\ \ldots\ x_n \rightarrow (\Gamma'_n, f')}$$

The first premise looks up information about F. In each of the remaining premises, F's formal arguments are substituted for the names of the actual arguments. For example, if P_1 is $\mathsf{Mono}\ x'_1 \leq$ then $P_1[x'_1 := x_1]$ is $\mathsf{Mono}\ x_1 \leq$. Each precondition has only the names of the preceding arguments and itself in scope. The $\rightsquigarrow$ relation on index functions substitutes the index functions of the actual arguments, which are bound in Γ, into F's index function (see Appendix B.3.4). The rule for function definition ensures that the free variables of f is a subset of the formal arguments, so that function application does not leak locally bound variables.[2]

We have briefly introduced how converting the source language to index functions drives the overall algorithm. We now turn our attention to property verification and inference: the $\rightarrow_{\mathsf{Infer}}$ and $\rightarrow_{\mathsf{Prop}}$ relations, which are used in the rules for function definition, application and let expressions.

[2]Since variable names are unique and we are in a purely functional context, it is sound to have out-of-scope variables in the environment. However, not creating an index function for function definitions that have out-of-scope variables improves compilation time because we won't attempt to apply rewrites to it.

B.2.4 Property inference. The inference judgment $\Gamma \vdash (x, E^{\oslash}) \rightarrow_{\mathsf{Infer}} \Gamma'$ extends an environment Γ to Γ' with properties inferred from the binding **let** $x = E^{\oslash}$. The most basic form of property inference adds postconditions to the environment on function application (AppProp). The App rule ensures that the function's preconditions are satisfied and the function definition rule ensures that the postcondition holds given the preconditions, so we can use the postcondition here without further checks.

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

AppProp
$$\frac{\Gamma_{\mathsf{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), f)}{\Gamma \vdash (y, F\ x_1\ \ldots\ x_n) \rightarrow_{\mathsf{Infer}} \Gamma, P_{post}[x'_1 := x_1, \ldots, x'_n := x_n, y' := y]}$$

InferNothing
$$\frac{\text{No other inference rule applies.}}{\Gamma \vdash (y, E^{\oslash}) \rightarrow_{\mathsf{Infer}} \Gamma}$$

If nothing can be inferred from the binding, another rule returns $\Gamma' = \Gamma$.

B.2.5 Property verification. The judgment $\Gamma \vdash P \rightarrow_{\mathsf{Prop}} (\Gamma', A)$ checks whether P holds under Γ. It produces an updated environment Γ' extended with P. The $\rightarrow_{\mathsf{Prop}}$ relation is defined by a set of "high-level" and "low-level" inference rules. The high-level rules verify properties using other properties, while the low-level rules reify proof obligations as solver queries.

We give one of the low-level rules for $\mathsf{Inj}\ x\ Y$ here:

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

QueryInjNeq
$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \text{fresh } j \qquad \Gamma, \mathsf{Range}\ i\ 0..j, \mathsf{Range}\ j\ 0..e_1 \vdash \mathsf{Query}_x\,([x(i) \in Y \wedge x(j) \in Y] * (x(i) \neq x(j))) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Inj}\ x\ Y \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Inj}\ x\ Y), \mathsf{Yes})}$$

The first premise looks up the index function of x to get the size of its domain e_1. The second premise creates a fresh variable j, and the third premise adds $0 \leq i < j < e_1$ to Γ using Range properties while querying the solver. The solver query reifies the proof obligation

$$\forall i, j \in 0..|x|\,.\,x(i) \in Y \wedge x(j) \in Y \wedge i \neq j \Rightarrow x(i) \neq x(j)$$

While the condition $i < j$ is used to build a sufficient query at two different indices, the codomain conditions $x(i) \in Y \wedge x(j) \in Y$ are part of the query itself so they are represented as a guard on the inequality expression. The return extends Γ with the proven property if the query succeeds in Yes.

Query rewriting specializes the injectivity query to x's index function by substituting and hoisting indexing symbols. This generates all possible cases for the query and may eliminate vacuously true cases. For example, if

$$\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_{dom})\,.\,[p_1] * e_1 + \ldots + [p_n] * e_n$$

then the query premise in QueryInjNeq is rewritten to:

$$\Gamma, \text{Range } i\ 0..j, \text{Range } j\ 0..e_{dom} \vdash \text{Query}_x \left(\begin{cases} [\,p_1 \wedge p_1[i := j] \wedge e_1 \in Y \wedge e_1[i := j] \in Y\,] * (e_1 \neq e_1[i := j]) \\ [\,p_1 \wedge p_2[i := j] \wedge e_1 \in Y \wedge e_2[i := j] \in Y\,] * (e_1 \neq e_2[i := j]) \\ \vdots \\ [\,p_n \wedge p_n[i := j] \wedge e_n \in Y \wedge e_n[i := j] \in Y\,] * (e_n \neq e_n[i := j]) \end{cases} \right)$$

using Sub and Hoist (Appendix B.3.4).

B.3 Index function layer

B.3.1 Base expression conversion rules ($\Gamma \vdash B \rightarrow f$). Base expressions B correspond to index function expressions e and cannot change the environment. Program variable references simply return the index function they are bound to. Array indexing requires that the index expression is scalar (of the form $\lambda\ () \ .\ e$) and checks that the index is within bounds by querying the solver. Idx requires that the query returns Yes, in which case conversion is successful. IdxFail requires that the query returns Unknown, in which case we terminate the algorithm in failure (**Error**).

Base expression conversion rules $\boxed{\Gamma \vdash B \rightarrow f}$

$$\textsc{Var}\ \frac{\Gamma_{\text{lxfn}}(x) = f}{\Gamma \vdash x \rightarrow (\Gamma, f)} \qquad \textsc{Idx}\ \frac{\Gamma \vdash B \rightsquigarrow \lambda\ () \ .\ e \qquad \Gamma \vdash \text{Query}\ (0 \leq e < |x|) \rightsquigarrow_{\text{Q}} \text{Yes}}{\Gamma \vdash x[B] \rightarrow (\Gamma, \lambda\ () \ .\ x(e))}$$

$$\textsc{IdxFail}\ \frac{\Gamma \vdash B \rightsquigarrow \lambda\ () \ .\ e \qquad \Gamma \vdash \text{Query}\ (0 \leq e < |x|) \rightsquigarrow \text{Unknown}}{\Gamma \vdash x[B] \rightarrow \textbf{Error}} \qquad \textsc{Constant}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n \rightarrow \lambda\ () \ .\ [\,\text{true}\,] * n}$$

$$\textsc{Length}\ \frac{}{\Gamma \vdash |x| \rightarrow \lambda\ () \ .\ |x|} \qquad \textsc{Mul}\ \frac{\Gamma \vdash B_1 \rightarrow \lambda\ () \ .\ e_1 \qquad \Gamma \vdash B_2 \rightarrow \lambda\ () \ .\ e_2}{\Gamma \vdash B_1 * B_2 \rightarrow \lambda\ () \ .\ e_1 \cdot e_2}$$

Mul converts multiplication, requiring that each operand steps to a scalar index function. The result is a scalar index function that multiplies the bodies of the operand's index functions. Rules for the other binary operators in Op are analogous.

B.3.2 Complete $E^{\oslash}$ conversion rules ($\Gamma \vdash E^{\oslash} \rightarrow (\Gamma', \overline{f})$). We give the complete $E^{\oslash}$ conversion rules here. For scatter, the rules are tried in the order they appear (also enumerated so).

AppUninterpreted applies a function definition that does not have an index function bound in Γ_{Def} by creating an uninterpreted function over a fresh name. Creating this on function application ensures sure that applications with distinct arguments yield distinct index functions.

Map and Scan convert the corresponding SOACs by binding the lambda's argument to the array argument's index function with the outer dimension dropped, extending the environment with the now-captured index variable i's range, and then deriving the lambda body. Adding the range enables bounds checking for indexing within the lambda body. The resulting environment Γ' is discarded as local bindings go out of scope.

All of the scatter rules require that the index and value arrays have the same length. Scatter1 converts scatters where the in-bounds indices are unique. Specifically, x_{idx}'s values within x_{dest}'s bounds are unique (i.e., when $x_{idx}|_{x_{idx}^{-1}(0..|x_{dest}|)}$ is invertible). The query checks that an existing bijective property fulfills these requirements.

Scatter2 is like Scatter1 except it asks the property layer to prove the bijectivity property.

Scatter3 covers the case where in-bounds written indices (x_{idx}) are strictly monotonically increasing,[3] producing a flat jagged array representation where rows start at the written indices. A variation on Scatter3 could match the case when there are no out-of-bounds indices in the reification of $x_{idx}|_{x_{idx}^{-1}(0..|x_{dst}|)}$ (i.e., it matches the case where there is only one guard after the rewrite).

Scatter4 creates an uninterpreted index function for a safe scatter (see Appendix B.3.3).

Conditional's first premise converts the condition into a predicate p_T. The second premise obtains a normalized form of the negation of p_T. The third premise converts p_T to properties, temporarily adds them to the environment and then converts the body of the then-branch. (Extending the environment can be necessary, e.g., to verify bounds checks in Idx.) Similarly for the else-branch using p_F. The returned index function combines the conditional with the guards in each of the branches: $[\,p_T\,] * 1 \cdot e_T$ will distribute $[\,p_T\,]$ over each of the terms in e_T, putting $[\,p_T\,]$ in conjunction with each term's existing guard. For example, $[\,p_T\,] * 1 \cdot ([\,p_1\,] * t_1 + [\,p_2\,] * t_2)$ distributes to $[\,p_T\,] * 1 \cdot [\,p_1\,] * t_1 + [\,p_T\,] * 1 \cdot [\,p_2\,] * t_2$, which yields $[\,p_T \wedge p_1\,] * t_1 + [\,p_T \wedge p_2\,] * t_2$.

BaseExpression converts base expressions. Seq converts a sequence to an index function; if B_1 is 0, then this corresponds to a "iota".

Complete $E^{\oslash}$ conversion rules (1/3) $\boxed{\Gamma \vdash E^{\oslash} \longrightarrow (\Gamma', \overline{f})}$

App
$$\frac{\Gamma_{\text{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), f) \qquad \Gamma \vdash P_1[x'_1 := x_1] \rightarrow_{\text{Prop}} (\Gamma'_1, \text{Yes}) \qquad \ldots \qquad \Gamma'_{n-1} \vdash P_n[x'_1 := x_1, \ldots, x'_n := x_n] \rightarrow_{\text{Prop}} (\Gamma'_n, \text{Yes}) \qquad \Gamma'_n \vdash f[x'_1 := x_1, \ldots, x'_n := x_n] \rightsquigarrow f'}{\Gamma \vdash F\ x_1\ \ldots\ x_n \longrightarrow (\Gamma'_n, f')}$$

AppUninterpreted
$$\frac{\Gamma_{\text{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), (1, \varnothing)) \quad \Gamma \vdash P_1[x'_1 := x_1] \rightarrow_{\text{Prop}} (\Gamma'_1, \text{Yes}) \qquad \ldots \qquad \Gamma'_{n-1} \vdash P_n[x'_1 := x_1, \ldots, x'_n := x_n] \rightarrow_{\text{Prop}} (\Gamma'_n, \text{Yes}) \quad \text{fresh } y, i \qquad \Gamma'_n \vdash \lambda\,(i : 0..|y|)\,.\,y(i) \rightsquigarrow f'}{\Gamma \vdash F\ x_1\ \ldots\ x_n \longrightarrow (\Gamma'_n, f')}$$

AppFail
$$\frac{\Gamma_{\text{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), f) \quad \Gamma \vdash P_1[x'_1 := x_1] \ldots P_n[x'_1 := x_1, \ldots, x'_n := x_n] \rightarrow_{\text{Prop}} (\Gamma', \text{Unknown})}{\Gamma \vdash F\ x_1\ \ldots\ x_n \longrightarrow \textbf{Error}}$$

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

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

Scatter1
$$\frac{\Gamma_{\text{Bij}}(x_{idx}) = (Y, Z) \quad \Gamma \vdash \text{Query}\ (Z \subseteq 0..|x_{dst}| \subseteq Y) \rightsquigarrow_{\text{Q}} \text{Yes} \qquad \Gamma \vdash \text{Query}\ (|x_{idx}| = |x_{val}|) \rightsquigarrow_{\text{Q}} \text{Yes} \qquad \text{fresh } i}{\Gamma \vdash \text{scatter}\ x_{dst}\ x_{idx}\ x_{val} \longrightarrow (\Gamma, \lambda\,(i : 0..|x_{dst}|)\,.\,[\,i \in Z\,] * x_{val}(x_{idx}^{-1}(i)) + [\,i \notin Z\,] * x_{dst}(i))}$$

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

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

Scatter3
$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Inj}\ x_{idx}\ 0..|x_{dst}| \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes}) \qquad \text{fresh } x_\perp, k, l, i_1, i_2 \\ \Gamma' \vdash \lambda\ (i_1 : 0..|x_{idx}|)\ .\ \begin{cases}[x_{idx}(i_1) \in 0..|x_{dst}|] * x_{idx}(i_1) \\ [x_{idx}(i_1) \notin 0..|x_{dst}|] * x_\perp\end{cases} \rightsquigarrow \lambda\ (i_1 : 0..e_{|x_{idx}|})\ .\ \begin{cases}[p] * e_{row} \\ [\neg p] * x_\perp\end{cases} \\ \Gamma' \vdash \mathsf{Query}\ (e_{row}[i_1 := 0] = 0) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes} \qquad \Gamma' \vdash \mathsf{Query}\ (e_{row}[i_1 := |x_{idx}|] = |x_{dst}|) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes} \\ \Gamma', 0 \le j < k < |x_{idx}| \vdash \mathsf{Query}\ (0 \le e_{row}[i_1 := j] \le e_{row}[i_1 := k] \le |x_{dst}|) \rightsquigarrow_{\mathsf{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{src} \rightarrow \left(\Gamma', \lambda\ (i_1 : 0..|x_{idx}| \times i_2 : 0..(e_{row}[i_1 := i_1 + 1] - e_{row}))\ .\ \begin{cases}[i_2 = 0 \wedge p] * x_{src}(i_1) \\ [i_2 \neq 0 \vee \neg p] * x_{dst}(e_{row} + i_2)\end{cases}\right)}$$

Scatter4
$$\frac{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow_{\mathsf{Safe}} \mathsf{Yes} \qquad \text{fresh } x_4}{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow (\Gamma, \lambda\ (i : 0..|x_{dst}|)\ .\ [\mathsf{true}] * x_4(i))}$$

Scatter5
$$\frac{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow_{\mathsf{Safe}} \mathsf{Unknown}}{\Gamma \vdash \mathsf{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow \mathbf{Error}}$$

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

BaseExpression
$$\frac{\Gamma \vdash B \rightarrow f}{\Gamma \vdash B \rightarrow (\Gamma, f)}$$

Seq
$$\frac{\Gamma \vdash B_1 \rightarrow \lambda\ ()\ .\ e_1 \qquad \Gamma \vdash B_2 \rightarrow \lambda\ ()\ .\ e_2 \qquad \text{fresh } i}{\Gamma \vdash B_1..B_2 \rightarrow (\Gamma, \lambda\ (i : e_1..e_2)\ .\ i + e_1)}$$

(More inference rules are given for this judgment below.)

The rules discussed so far have been restricted to one return for clarity. For the loop rules we will do the opposite; they are best understood in their most general form with n returns. All of the preceding rules are straightforward to generalize to multiple returns.

WhileLoop converts a while loop with n returns to n uninterpreted index functions. The pre- and postconditions must be syntactically equivalent up to variable names: the preconditions are expressed in terms of the formal arguments while the postconditions are expressed in terms of the function's returns. This ensures that the properties are loop invariants and that the postconditions can only be defined in terms of the return variables y (i.e., that $\bigcup_{j=1}^{n} \mathrm{fv}(P'_j) \subseteq \{y'_1, \ldots, y'_n\}$). In the first premise, _ will match both the case where F has an index function and where it does not. (It's irrelevant for the rule.)

ForLoop is like WhileLoop but further requires that the loop iterator variable x_{n+1} has a range property corresponding to the loop iteration space and that x_{n+1} is not referenced in the postconditions.[4] This is because F does not return x_{n+1} and yet it will vary across iterations.[5] The upper bound of x_{n+1} is derived and its range is added to Γ while deriving the loop body.

[4]The WhileLoop rule is written in a more compact style because it is included in the paper.

[5]This could be relaxed using a more complicated rule that requires any preconditions on x_{n+1} are also satisfied by $x_{n+1} + 1$.

Complete $E^{\oslash}$ conversion rules (2/3) $\boxed{\Gamma \vdash E^{\oslash} \to (\Gamma', \overline{f})}$

$$\textsc{WhileLoop}\ \frac{\Gamma_{\text{Def}}(F) = (\overline{(x', P_{pre})}^{(n)}, (\overline{y}^{(n)}, \overline{P_{post}}^{(n)}), \_) \qquad \Gamma_{\text{lxfn}}(x_{init_n}) = \lambda\ () \,.\, [\,p\,] * 1 \qquad \text{fresh } \overline{i}^{(n)}, \overline{z}^{(n)}}{\Gamma, p \vdash F\ \overline{x_{init}}^{(n)} \to (\Gamma', \overline{f}^{(n)}) \qquad \forall j \in 1, \ldots, n \,.\, P_{pre_j} \text{ unifies with } P_{post_j}[y_1 := x'_1, \ldots, y_n := x'_n]}$$
$$\Gamma \vdash \textbf{loop}\ \overline{x}^{(n)} = \overline{x_{init}}^{(n)}\ \textbf{while}\ x_n\ \textbf{do}\ F\ \overline{x}^{(n)} \ \to\ (\Gamma, \lambda\ (i_1 : 0..|z_1|) \,.\, z_1(i), \ldots, \lambda\ (i_n : 0..|z_n|) \,.\, z_n(i))$$

ForLoop

$$\frac{\begin{array}{c}\Gamma_{\text{Def}}(F) = (\overline{(x', P)}^{(n)}\ (x_{n+1}, \text{Range } x_{n+1}\ 0..e_1), (\overline{y}^{(n)}, \overline{P'}^{(n)}), \_) \qquad x_{n+1} \notin \bigcup_{j=1}^{n} \text{fv}(P'_j) \\ P_1, \ldots, P_n \text{ unifies with } P'_1[y_1 := x'_1, \ldots, y_n := x'_n], \ldots, P'_n[y_1 := x'_1, \ldots, y_n := x'_n] \\ \Gamma \vdash B \to \lambda\ () \,.\, e'_1 \qquad e_1 \text{ unifies with } e'_1 \\ \Gamma, x_{n+1} \mapsto \lambda\ () \,.\, x_{n+1}, \text{Range } x_{n+1}\ 0..e_1 \vdash F\ \overline{x_{init}}^{(n)}\ x_{n+1} \to (\Gamma', f) \qquad \text{fresh } \overline{i}^{(n)}, \overline{z}^{(n)}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{loop}\ (\overline{x}^{(n)}) = (\overline{x_{init}}^{(n)})\ \textbf{for}\ x_{n+1} < B\ \textbf{do}\ F\ \overline{x}^{(n+1)} \\ \quad \to (\Gamma, \lambda\ (i_1 : 0..|z_1|) \,.\, z_1(i_1), \ldots, \lambda\ (i_n : 0..|z_n|) \,.\, z_n(i_n))\end{array}}$$

WhileLoopFail

$$\frac{\text{No other rule applies.}}{\Gamma \vdash \textbf{loop}\ \overline{x}^{(n)} = \overline{x_{init}}^{(n)}\ \textbf{while}\ x_n\ \textbf{do}\ F\ \overline{x}^{(n)} \to \textbf{Error}}$$

ForLoopFail

$$\frac{\text{No other rule applies.}}{\Gamma \vdash \textbf{loop}\ (\overline{x}^{(n)}) = (\overline{x_{init}}^{(n)})\ \textbf{for}\ x_{n+1} < B\ \textbf{do}\ F\ \overline{x}^{(n+1)} \to \textbf{Error}}$$

B.3.3 Scatter safety rules ($\Gamma \vdash f \to A$). The rules Scatter3, Scatter4, and Scatter5 determine whether an application of scatter satisfies its deterministic semantics. SafeScatterFail yields Unknown if determinism cannot be established. The remaining rules assert determinism via sufficient conditions: SafeScatterInj requires a duplicate-free index array by proving an injectivity property on the index array. SafeScatterConst1 requires a constant index function, ensuring the value array replicates a single constant. SafeScatterConst2 requires that the value array has a singleton range.

$\boxed{\Gamma \vdash f \to_{\text{Safe}} A}$

SafeScatterInj

$$\frac{\Gamma \vdash \text{Query } (\text{Inj } x_{idx}\ 0..|x_{dst}|) \rightsquigarrow_{\text{Q}} \text{Yes} \qquad \Gamma \vdash \text{Query } (|x_{idx}| = |x_{val}|) \rightsquigarrow_{\text{Q}} \text{Yes}}{\Gamma \vdash \text{scatter } x_{dst}\ x_{idx}\ x_{val} \to_{\text{Safe}} \text{Yes}}$$

SafeScatterConst1

$$\frac{\Gamma(x_{val}) = \lambda\ (i : 0..e_1) \,.\, e_2 \qquad i \notin \text{fv}(e_2) \qquad \Gamma \vdash \text{Query } (|x_{idx}| = |x_{val}|) \rightsquigarrow_{\text{Q}} \text{Yes}}{\Gamma \vdash \text{scatter } x_{dst}\ x_{idx}\ x_{val} \to_{\text{Safe}} \text{Yes}}$$

$$\textsc{SafeScatterConst2}\quad \frac{\Gamma_{\text{Range}}(x_{val}) = e_1..e_2 \qquad \Gamma \vdash \text{Query}\ (e_1 = e_2 - 1) \rightsquigarrow_{\text{Q}} \text{Yes} \qquad \Gamma \vdash \text{Query}\ (|x_{idx}| = |x_{val}|) \rightsquigarrow_{\text{Q}} \text{Yes}}{\Gamma \vdash \text{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow_{\text{Safe}} \text{Yes}}$$

$$\textsc{SafeScatterFail}\quad \frac{\text{No other} \rightarrow_{\text{Safe}} \text{rule applies.}}{\Gamma \vdash \text{scatter}\ x_{dst}\ x_{idx}\ x_{val} \rightarrow_{\text{Safe}} \text{Unknown}}$$

B.3.4 Term rewriting and contexts. We rewrite index functions and expressions to simplify eventual queries (e.g., bounds checking in Idx). We formalize expression rewrite rules $\Gamma \vdash e \rightarrow e'$ using expression contexts C.

$$\begin{aligned} C \quad ::= \quad & \square \mid C + e \mid C \cdot t \mid [C] * e \mid [p] * C \mid |C| \mid \textstyle\sum_{x=C}^{e}(e) \mid \sum_{x=e}^{C}(e) \mid \sum_{x=e}^{e}(C) \\ \mid \quad & x(C) \mid x^{-1}(C) \mid 2^C \mid \neg C \mid C \wedge p \mid p \wedge C \mid C \vee p \mid p \vee C \mid C \leq e \mid e \leq C \end{aligned}$$

A context contains a single hole $\square$, where $C\langle e \rangle$ denotes substituting the subexpression e into that hole. For example, given $C = [p_1] * e_1 + [p_2] * x_1(\square)$, applying e_2 yields $C\langle e_2 \rangle = [p_1] * e_1 + [p_2] * x_1(e_2)$.

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

We write unify $(e_1,\ e_2) = \sigma$ when e_1 and e_2 are syntactically equivalent up to renaming of bound variables, where σ substitutes values for free variables (acting as identity on variables outside its domain).

Rewriting expressions. We substitute index functions in to expressions by reduction over indexing symbols (Sub), yielding nested guarded expressions. Hoist moves guards that are nested inside a symbol to the root of the surrounding expression (the context C), provided that no guard depends on a variable bound in C. The MECE property of guards ensures the transformation's validity: exactly one term in the sum $\sum_j [p_j]$ equals 1 while all others equal 0.

$$\boxed{\Gamma \vdash e \rightarrow e'}$$

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

For example, given $\Gamma(x) = \lambda\,(i : 0..e_1)\ .\ [\text{false}] * n + [\text{true}] * i$ and $[\text{true}] * x(i) + [\text{true}] * 1$, Sub steps to:

$$[\text{true}] * ([\text{false}] * n + [\text{true}] * i) + [\text{true}] * 1$$

The nested guarded expression is then hoisted:

$$[\text{false}] * 1 \cdot ([\text{true}] * n + [\text{true}] * 1) + [\text{true}] * 1 \cdot ([\text{true}] * i + [\text{true}] * 1)$$

Multiplication of expressions distributes hoisted guards over the new nested guards:

$$[\text{false}] * ([\text{true}] * n) + [\text{false}] * ([\text{true}] * 1) + [\text{true}] * ([\text{true}] * i) + [\text{true}] * ([\text{true}] * 1)$$

This naturally preserves the mutually exclusive and collectively exhaustive (MECE) property. We define multiplication of guards with other guards to yield logical conjunction. This generates all

possible combinations of the involved guards, including contradictory guards. Guards are simplified using boolean arithmetic and false-guarded terms are eliminated (see Appendix B.3.5):

$$[\,\text{false} \wedge \text{true}\,] * n + [\,\text{false} \wedge \text{true}\,] * 1 + [\,\text{true} \wedge \text{true}\,] * i + [\,\text{true} \wedge \text{true}\,] * 1 \rightsquigarrow [\,\text{true}\,] * i + [\,\text{true}\,] * 1$$

Rewrites enable automatic tracking of positional dependencies backwards to the formal arguments of a function definition in this manner.

Rewriting index functions and queries. Term rewriting extends to index functions and queries, enabling substitution and hoisting for these objects. An index function f's body is rewritten under Γ extended with ranges implied by f's domain.

$$\boxed{\Gamma \vdash f \to f'} \qquad \boxed{\Gamma \vdash \text{Query}_x\,(e) \to \text{Query}_x\,(e')}$$

$$\textsc{RewriteBody}\ \frac{\Gamma, \text{Range}\ i\ 0..e_1, 0 < e_1 \vdash e_2 \rightsquigarrow e_2'}{\Gamma \vdash \lambda\,(i : 0..e_1)\,.\,e_2 \to \lambda\,(i : 0..e_1)\,.\,e_2'} \qquad \textsc{RewriteQuery}\ \frac{\Gamma \vdash e \to e'}{\Gamma \vdash \text{Query}_x\,(e) \to \text{Query}_x\,(e')}$$

B.3.5 Complete rewrite rules for index functions ($\Gamma \vdash f \to f'$). RewriteDomain rewrites the domain of an index function. In particular, this will substitute dimension sizes via Hoist. RecSum rewrites recurrences introduced by scan into closed-form sums. The rule requires that $\circlearrowleft$ appears exactly once in the recurrence step and not in the base case ($[\,i = 0\,]$) or summed terms. (Note $\sum$ is distinct from the sum symbol $\sum$.) RecReplicate is like RecSum except it matches the case where the recurrence just replicates the base case.

$$\boxed{\Gamma \vdash f \to f'}$$

$$\textsc{RewriteDomain}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1'}{\Gamma \vdash \lambda\,(i : 0..e_1)\,.\,e_2 \to \lambda\,(i : 0..e_1')\,.\,e_2} \qquad \textsc{RewriteBody}\ \frac{\Gamma, \text{Range}\ i\ 0..e_1, 0 < e_1 \vdash e_2 \rightsquigarrow e_2'}{\Gamma \vdash \lambda\,(i : 0..e_1)\,.\,e_2 \to \lambda\,(i : 0..e_1)\,.\,e_2'}$$

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

$$\textsc{RecReplicate}\ \frac{\circlearrowleft \text{ does not occur in } e_2}{\Gamma \vdash \lambda\,(i : 0..e_1)\,.\,[\,i = 0\,] * e_2 + [\,i \neq 0\,] * \circlearrowleft \to \lambda\,(i : 0..e_1)\,.\,e_2[i := 0]}$$

B.3.6 Complete rewrite rules for index function expressions ($\Gamma \vdash e \to e'$). JoinGuards joins guards in disjunction when all of their terms are equivalent under either of the guards. QueryPred queries predicates to prove them true. RewriteTerm rewrites terms under assumption of their guards. FalsifyGuard and ElimGuard falsify and eliminate contradictory guards, respectively. NegUnhoistableGuard rewrites a sum symbol over negated predicate that cannot be hoisted to a form that uses the non-negated predicate. This is a heuristic to aid reasoning about overlapping sums of boolean arrays in the solver. IntegerToBool rewrites guarded integers to a sum of integers times guards. This normalizes source-level conversion of booleans to integers. SplitSum splits a sum symbol into multiple sums symbol. SumConst eliminates a sum symbol when the summands are constant with respect to the iterator variable. ElimLen replaces a $|x|$ symbol for the size of x's

outer dimension when x is bound to an index function in Γ. Recall that scalar index functions of the form $\lambda\,()\,.\,e$ abbreviate $\lambda\,(i : 0..1)\,.\,e$, so if x is bound to a scalar function, e_{dom} is simply 1.

We also rewite predicates using standard Boolean algebra (not shown) such as reducing $p \wedge \neg p$ to false.

$\boxed{\Gamma \vdash e \longrightarrow e'}$

$$\textsc{Hoist}\ \frac{\Gamma \vdash [\ \bigvee_j p_j\ ] \rightsquigarrow [\,\mathrm{true}\,] \qquad \mathrm{bv}(C) \cap \left(\bigcup_j \mathrm{fv}(p_j)\right) = \emptyset}{\Gamma \vdash C\langle \textstyle\sum_j [\ p_j\ ] * e_j\rangle \longrightarrow \sum_j [\ p_j\ ] * 1 \cdot (C\langle e_j\rangle)}$$

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

$$\textsc{JoinGuards}\ \frac{\Gamma, p_1 \vdash e_2 \rightsquigarrow e_2' \qquad e_1 \text{ unifies with } e_2'}{\Gamma \vdash C\langle [\ p_1\ ] * e_1 + [\ p_2\ ] * e_2\rangle \longrightarrow C\langle [\ p_1 \vee p_2\ ] * t_1\rangle}$$

$$\textsc{QueryPred}\ \frac{\Gamma \vdash \mathrm{Query}\ (p) \rightsquigarrow_{\mathrm{Q}} \mathrm{Yes}}{\Gamma \vdash C\langle p\rangle \longrightarrow C\langle \mathrm{true}\rangle}$$

$$\textsc{RewriteTerm}\ \frac{\Gamma, p \vdash e \rightsquigarrow e'}{\Gamma \vdash C\langle [\ p\ ] * e\rangle \longrightarrow C\langle [\ p\ ] * e'\rangle}$$

$$\textsc{FalsifyGuard}\ \frac{\begin{array}{c}\text{Conjunctive normal form of } p \text{ is } p_1' \wedge \cdots \wedge p_n'.\\ \exists h \in \{1,\ldots,n\}\ .\ \left(\Gamma, \bigwedge_{i \in \{1,\ldots,n\}\setminus\{h\}} p_i' \vdash \mathrm{Query}\ (\neg p_h) \rightsquigarrow_{\mathrm{Q}} \mathrm{Yes}\right)\end{array}}{\Gamma \vdash C\langle [\ p\ ]\rangle \longrightarrow C\langle [\,\mathrm{false}\,]\rangle}$$

$$\textsc{ElimGuard}\ \frac{}{\Gamma \vdash C\langle [\,\mathrm{false}\,] * e\rangle \longrightarrow C\langle 0\rangle}$$

$$\textsc{NegUnhoistableGuard}\ \frac{x \in \mathrm{fv}(p)}{\Gamma \vdash C\langle \textstyle\sum_{x=e_1}^{e_2}([\,\mathrm{true}\,] * \neg p)\rangle \longrightarrow C\langle \sum_{x=e_1}^{e_2}([\,\mathrm{true}\,] * 1) + (-1) \cdot \sum_{x=e_1}^{e_2}([\,\mathrm{true}\,] * p)\rangle}$$

$$\textsc{IntegerToBool}\ \frac{\Gamma \vdash [\ \bigvee_j p_j\ ] \rightsquigarrow [\,\mathrm{true}\,]}{\Gamma \vdash C\langle \textstyle\sum_j [\ p_j\ ] * n_j\rangle \longrightarrow C\langle \sum_j [\,\mathrm{true}\,] * (n_j \cdot p_j)\rangle}$$

$$\textsc{SplitSum}\ \frac{}{\Gamma \vdash C\langle \textstyle\sum_{x=e_1}^{e_2}(\sum_j t_j)\rangle \longrightarrow C\langle \sum_j \sum_{x=e_1}^{e_2}(t_j)\rangle}$$

$$\textsc{SumConst}\ \frac{x \notin \mathrm{fv}(e_3)}{\Gamma \vdash C\langle \textstyle\sum_{x=e_1}^{e_2}(e_3)\rangle \longrightarrow C\langle (e_2 - e_1 + 1) \cdot e_3\rangle}$$

$$\textsc{ElimLen}\ \frac{\Gamma_{\mathrm{lxfn}}(x) = \lambda\,(i : 0..e_{dom})\,.\,e_x}{\Gamma \vdash C\langle |x|\rangle \longrightarrow C\langle e_{dom}\rangle}$$

B.3.7 Multi-dimensional index functions.

Variable	x, y, z, i, j, k	
Constant	$n, m \in \mathbb{Z}$	
Symbol	s	$::= [\ p\ ] * e \mid x \mid \lvert x\rvert \mid \sum_{x=e}^{e}(e) \mid x(e,\ldots,e) \mid x^{-1}(e,\ldots,e) \mid p \mid 2^e \mid \%_D(e_{idx}) \mid \circlearrowleft$
Predicate	p	$::= x \mid x(e,\ldots,e) \mid \mathrm{true} \mid \mathrm{false} \mid \neg p \mid e \le e \mid p \wedge p \mid p \vee p \mid \circlearrowleft$
Term	t	$::= n \mid s \mid s \cdot t$
Expression	e	$::= t \mid t + e$
Domain	D	$::= i : 0..e \mid i : 0..e, D \mid i : 0..e \times D$
Index fun.	f	$::= \lambda\,(D)\,.\,e$

Expression contexts are updated with the changed symbols. Order of evaluation for multi-dimensional indexing is left-to-right, using the same logic as before: $x(C, e, \ldots, e)$ is rewritten to a

fixed point before $x(e, C, \ldots, e)$ and so on.

$$\begin{array}{rcl} C & ::= & \square \mid C + e \mid C \cdot t \mid [C] * e \mid |C| \mid [p] * C \mid \sum_{x=C}^{e}(e) \mid \sum_{x=e}^{C}(e) \mid \sum_{x=e}^{e}(C) \\ & \mid & x(C, e, \ldots, e) \mid x(e, C, \ldots, e) \mid x(e, e, \ldots, C) \\ & \mid & x^{-1}(C, e, \ldots, e) \mid x^{-1}(e, C, \ldots, e) \mid x^{-1}(e, e, \ldots, C) \\ & \mid & 2^{C} \mid \%_D(C) \mid \neg C \mid C \leq e \mid e \leq C \mid C \wedge p \mid p \wedge C \mid C \vee p \mid p \vee C \end{array}$$

The corresponding rule changes are straightforward. For example, Map becomes:

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

B.3.8 Complete rewrite rules for flattened domains. Flatten derives index functions for flattened arrays, preserving their multi-dimensional structure in the domain. Unflatten derives index functions for unflattened arrays by exploiting multi-dimensional structure preserved via Flatten. We know that the dimensions in question mut be regular (i.e, the inner dimension cannot depend on the outer) because the source language does not support irregular arrays hence calling unflatten on a flat representation of an irregular array would cause a type error.

Complete $E^{\oslash}$ conversion rules (3/3) $\boxed{\Gamma \vdash E^{\oslash} \longrightarrow (\Gamma', \overline{f})}$

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

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

PropFlatten propagates flattened domains to index functions that index into them, rewriting flat indices to use the flattened domain's variables, when the flat and non-flat domains range over the same flat indices. ElimLenFlat is like ElimLen but returns the total length of the flat dimension (the size of the corresponding array in the program). SubFlat substitutes index function bodies across flattened dimensions, introducing $\%_D(e_{idx})$ to express outer dimension indices in terms of flat indices.

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

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

$\boxed{\Gamma \vdash e \longrightarrow e'}$

$$\textsc{ElimLenFlat}\quad \frac{\Gamma(x) = \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;e_x \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_2[i_1 := j]) \rightsquigarrow e_3}{\Gamma \vdash C\langle |x| \rangle \longrightarrow C\langle e_3 \rangle}$$

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

Idealy we want to convert $\%_{i_1:0..e_1 \times i_2:0..e_2}(e_{idx})$ to a value for i_1. In general, this would lead to non-linear arithmetic (i.e., division) which our solver does not support.

Common index expressions do, however, admit static solutions to $\%_{i_1:0..e_1 \times i_2:0..e_2}(e_{idx})$ while avoiding non-linear arithmetic. When substituting an indexing operation into a flattened multi-dimensional array, we use three rules that address specialized cases by solving the following equation for i_1:

$$e_{idx} = \sum_{l=0}^{\%_{i_1:0..e_1 \times i_2:0..e_2}(e_{idx})-1}(e_2[i_1 := l]) + i_2 \qquad (0 \le i_2 < e_2) \tag{1}$$

SolveIdx1 finds these solutions by checking bounds via the solver. (This is undecidable.) While SolveIdx2 and SolveIdx2 check whether e_{idx} is syntactically equivalent with the start or the end of a segment (e_1 and e_2, respectively). (This is decidable.)

Without static solutions, P^2 exploits $0 \le \%_{i:0..e_1 \times j:0..e_2}(e_3) < e_2$ when e_2 is independent of i_1, which holds for regular arrays.

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

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

$$\textsc{SolveIdx2}\quad \frac{\text{unify}\,(i_1 : 0..e_1 \times i_2 : 0..e_2,\; D) = \sigma \qquad \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_2[i_1 := j]) \rightsquigarrow e_{row} \qquad \text{unify}\,(e_{row},\; e_{idx}) = \{i_1 \mapsto e_{i_1'}\}}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;C\langle \%_D(e_{idx}) \rangle \longrightarrow \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;C\langle \sigma(e_{i_1'}) \rangle}$$

$$\textsc{SolveIdx3}\quad \frac{\text{unify}\,(i_1 : 0..e_1 \times i_2 : 0..e_2,\; D) = \sigma \qquad \text{fresh } j \qquad \Gamma \vdash \sum_{j=0}^{i_1-1}(e_2[i_1 := j]) \rightsquigarrow e_{row} \qquad \text{unify}\,(e_{row}[i_1 := i_1 + 1] - 1,\; e_{idx}) = \{i_1 \mapsto e_{i_1'}\}}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;C\langle \%_D(e_{idx}) \rangle \longrightarrow \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;C\langle \sigma(e_{i_1'}) \rangle}$$

For example, in the below program, Flatten is used to derive x's index function and PropFlatten is used to derive y's index function.[6]

let a = map $(\lambda i.\,\text{map}\ (\lambda j.\,i + j)\ (0..m))\ (0..n)$	$a = \lambda\,(i : 0..n, j : 0..m)\;.\;i + j$
let x = flatten a	$x = \lambda\,(i : 0..n \times j : 0..m)\;.\;i + j$
let y = map $(\lambda k.\,x[k] + 1)\ (0..n * m)$	$y = \lambda\,(i : 0..n \times j : 0..m)\;.\;i + j + 1$

The above shows normalized index functions for each program variable; via Map, y's index function

[6]For brevity, we allow nesting expressions inside map without a let-binding here.

is initially $y = \lambda\,(k : 0..(n \cdot m))\ .\ x(k) + 1$. PropFlatten then rewrites the domain to x's domain: $y = \lambda\,(i : 0..n \times j : 0..m)\ .\ x(i \cdot m + j) + 1$. Rule SubFlat replaces $x(i \cdot m + j)$ with x's body, expressing i as $\%_{i:0..n\times j:0..m}(i \cdot m + j)$:

$$y = \lambda\,(i : 0..n \times j : 0..m)\ .\ \%_{i:0..n\times j:0..m}(i \cdot m + j) + \left(i \cdot m + j - \sum_{l=0}^{\%_{i:0..n\times j:0..m}(i\cdot m+j)-1}(m)\right) + 1$$

Here the equation simplifies $\%_{i_1:0..e_1\times i_2:0..e_2}(e_{idx})$ to $i \cdot m + j = \%_{i:0..n\times j:0..m}(i \cdot m + j) \cdot m + j$ which means $\%_{i:0..n\times j:0..m}(i \cdot m + j)$ is equal to i. More generally, SolveIdx1 finds this solution by checking that the index expression lies within the ith row's bounds—normalizing this unwieldy form to:

$$y = \lambda\,(i : 0..n \times j : 0..m)\ .\ i + j + 1.$$

Even without static solutions,[7] P^2 exploits $0 \le \%_{i:0..e_1\times j:0..e_2}(e_3) < e_2$ when e_2 doesn't depend on i_1, which is the case for flattened regular arrays. (Below we let the inner dimension depend on the outer, hence the sum in Eq. (1).)

The rules assume 1- or 2D arrays. Generalization to n dimensions is straightforward for Flatten and SubFlat, but requires heuristics for other rules due to ambiguities.[8]

Augmenting the remaining rewrite rules to handle flattened domains is straightforward. For example, below, SegRecSum and SegRecReplicate make use of information in the flattened domain to recognize scan applied over the rows of flattened arrays (segmented prefix sums).

$$\boxed{\Gamma \vdash f \to f'}$$

SegRecReplicate

$$\frac{\circlearrowleft \text{ does not occur in } e_3}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ [\,i_2 = 0\,] * e_3 + [\,i_2 \neq 0\,] * \circlearrowleft \to \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ e_3[i_2 := 0]}$$

SegRecSum

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

B.3.9 Simplified rewrite rules for flattened regular arrays. We give simplified rules for handling flattened index functions that describe only regular arrays. The rules shown above generalize these to cover both regular and irregular arrays by replacing $i_1 \cdot e_2$ for $\Gamma \vdash \sum_{l=0}^{i_1-1}(e_2\{i_1 \mapsto l\}) \rightsquigarrow e_{row}$ and $e_1 = e_2 \cdot e_3$ for $e_1 = e_{row}\{i_1 \mapsto e_2 - 1\}$. The rules in above essentially reduce to these rules when $i_1 \notin \mathrm{fv}(e_2)$. (Only the rules above are required.)

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

PropFlatten-Simplified

$$\frac{\Gamma(x) = \lambda\,(i_2 : 0..e_2 \times i_3 : 0..e_3)\ .\ e_x \qquad i_2 \notin \mathrm{fv}(e_3) \qquad \Gamma \vdash \mathrm{Query}\ (e_1 = e_2 \cdot e_3) \rightsquigarrow_{\mathrm{Q}} \mathrm{Yes}}{\Gamma \vdash \lambda\,(i_1 : 0..e_1)\ .\ C\langle x(e_{idx})\rangle \to \lambda\,(i_2 : 0..e_2 \times i_3 : 0..e_3)\ .\ C\langle x(e_{idx})\rangle[i_1 := i_2 \cdot e_3 + i_3]}$$

SolveIdx1-Simplified

$$\frac{(\Gamma, \mathrm{Range}\ i_1\ 0..e_1, \mathrm{Range}\ i_2\ 0..e_2) \vdash \mathrm{Query}\ (i_1 \cdot e_2 \le e_{idx} < i_1 \cdot e_2 + e_2) \rightsquigarrow_{\mathrm{Q}} \mathrm{Yes} \qquad i_1 \notin \mathrm{fv}(e_2)}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ C\langle \%_{e_1,e_2}(e_{idx})\rangle \to \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\ .\ C\langle i_1\rangle}$$

[7] Additional rules for solving the equation appear in the appendix.

[8] For instance, D in $\%_D(\cdot)$ might unify with multiple flat dimensions, and PropFlatten must choose which dimension inherits the flattened structure.

SolveIdx2-Simplified

$$\frac{\text{unify}\,(i_1 \cdot e_2,\ e_{idx}) = \{i_1 \mapsto e_{i_1}\} \qquad i_1 \notin \text{fv}(e_2)}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\,.\,C\langle \%_{e_1,e_2}(e_{idx})\rangle \longrightarrow \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\,.\,C\langle e_{i_1}\rangle}$$

SolveIdx3-Simplified

$$\frac{\text{unify}\,(i_1 \cdot e_2 + e_2 - 1,\ e_{idx}) = \{i_1 \mapsto e_{i_1}\} \qquad i_1 \notin \text{fv}(e_2)}{\Gamma \vdash \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\,.\,C\langle \%_{e_1,e_2}(e_{idx})\rangle \longrightarrow \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\,.\,C\langle e_{i_1}\rangle}$$

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

SubFlat-Simplified

$$\frac{\Gamma(x) = \lambda\,(i_2 : 0..e_2 \times i_3 : 0..e_3)\,.\,e_x \qquad i_2 \notin \text{fv}(e_3)}{\Gamma \vdash C\langle x(e_{idx})\rangle \longrightarrow C\langle e_x[i_2 := \%_{e_1,e_2}(e_{idx}),\ i_3 := e_{idx} - \%_{e_1,e_2}(e_{idx}) \cdot e_3]\rangle}$$

B.4 Property layer

B.4.1 Property verification ($\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma', A)$). Properties are verified either by expanding the proof obligations into (in)equalities discharged by the algebra layer (low-level reasoning), or by reasoning in terms of other properties in the environment (high-level reasoning).

We formalize both levels of reasoning in the $\rightarrow_{\text{Prop}}$ relation. The judgment $\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma', A)$ asserts whether P verifies under environment Γ and that its verification extends Γ to Γ'. The only environment update needed for our system to function is to add P to Γ when the answer A is Yes. An obvious optimization would be to also add properties proven during verification of P. For example, to prove bijectivity, we often first prove injectivity.

High-level rules. The high-level rules are tried before the low-level rules because they are often more powerful (e.g., can prove properties on uninterpreted functions) and are often computationally less expensive. The low-level rules are tried when no high-level rule applies. The names of low-level rules are prefixed by Query to discern them from high-level rules. Note that high-level rules may use low-level rules to verify premises that use the $\rightarrow_{\text{Prop}}$ relation and vice versa.

We support lists of properties in postconditions (the source-level $\overline{\pi}$ gets parsed to $\overline{P}$), in which case each property has to hold and is verfied left-to-right while assuming preceding properties (ListOfProperties).

KnownRange says that a range property is verified if it already exists in Γ. Similar rules can be defined for all the other properties.

FiltPartPreservesRange says that filter and partition operations preserve range properties. Specifically, it verifies verifies a range property on an index function x_1 by first verifying that range property on an index function x_2 when x_1 is the result of filtering/partitioning x_2. This is useful because x_1 may be uninterpreted and hence a query for the range would be unprovable and because the range property may already be in Γ for x_2 and hence verified via KnownRange.

Similarly, FiltPreservesMono says that filter preserves monotonicity properties, FiltPartPreservesInj says that filter and partition preserve injectivity, and FiltPreservesBij says that filter preserves bijectivity.

InjSubset verifies an injectivity property for an index function restricted to the preimage of codomain Y_1 using a known injectivity property that restrictes that function to the preimage of codomain Y_2 given that $Y_1 \subseteq Y_2$.

BijSubset is like InjSubset but for bijectivity which further requires that the images Z_1 and Z_2 are equal. Since $Z_2 \subseteq Y_2$ from the already-established bijective property, further requiring $Z_1 = Z_2$ and $Y_1 \subseteq Y_2$ then gives us $Y_1 - Z_1 = \emptyset$.

InjFromBij uses that a bijective function is also injective to verify injectivity. Specifically, it verifies an injectivity property for an index function x_1 restricted to the preimage of Y_1 using that x_1 is bijective when restricted to Y_2 (a property already in Γ) and $Y_1 \subseteq Y_2$.

FilterInj verifies that x_1 is injective given that it is the result of filtering an array x_2 by a predicate p. The rule premises ensure that x_2, restricted to the indices for which p is true, is an injective function. This is again useful because x_1 may be uninterpreted, but the restriction of x_2, constructed in the premises, is not and therefore queries over that function may succeed. We use $\perp$ to mean some sentinel value not in Y (it's straightforward to pick one given Y); cases generated with this value then hold trivially (e.g., in QueryInjEq).

FilterRange is like FilterInj, but for range properties.

InjConcat verfies that x_1 is injective given that it is a concatenation of two index functions f_1 and f_2. The rule premises ensure that f_1 and f_2 are both injective and that their images are disjoint.

MonoConcat verfies that x_1 is monotonic given that it is a concatenation of two index functions f_1 and f_2, which are themselves monotonic and for which f_1 maps to values that precede f_2's values.

Complete property verification rules (1/4) (high-level) $\boxed{\Gamma \vdash P \rightarrow_{\mathsf{Prop}} (\Gamma', A)}$

$$\textsc{ListOfProperties}\quad \frac{\Gamma \vdash P_1 \rightarrow_{\mathsf{Prop}} (\Gamma'_1, \mathsf{Yes}) \qquad \ldots \qquad \Gamma'_{n-1} \vdash P_n \rightarrow_{\mathsf{Prop}} (\Gamma'_n, \mathsf{Yes})}{\Gamma \vdash P_1 \ldots P_n \rightarrow_{\mathsf{Prop}} (\Gamma'_n, \mathsf{Yes})}$$

$$\textsc{KnownRange}\quad \frac{\Gamma_{\mathsf{Range}}(x) = (e_1, e_2)}{\Gamma \vdash \mathsf{Range}\ x\ e_1..e_2 \rightarrow_{\mathsf{Prop}} (\Gamma, \mathsf{Yes})}$$

$$\textsc{FiltPartPreservesRange}\quad \frac{\Gamma_{\mathsf{FiltPart}}(x_1) = (x_2, f_1, (f_2, \ldots, f_n)) \qquad \Gamma \vdash \mathsf{Range}\ x_2\ e_1..e_2 \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes})}{\Gamma \vdash \mathsf{Range}\ x_1\ e_1..e_2 \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Range}\ x_1\ e_1..e_2), \mathsf{Yes})}$$

$$\textsc{FiltPreservesMono}\quad \frac{\Gamma_{\mathsf{FiltPart}}(x_1) = (x_2, f_1, (\lambda\ (D)\ .\ \mathsf{true}, \lambda\ (D)\ .\ \mathsf{false})) \qquad \Gamma \vdash \mathsf{Mono}\ x_2\ \prec \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes})}{\Gamma \vdash \mathsf{Mono}\ x_1\ \prec \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Mono}\ x_1\ \prec), \mathsf{Yes})}$$

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

$$\textsc{PartPreservesBij}\quad \frac{\Gamma_{\mathsf{FiltPart}}(x_1) = (x_2, \lambda\ (D)\ .\ \mathsf{true}, (f_1, \ldots, f_n)) \qquad \Gamma \vdash \mathsf{Bij}\ x_2\ Y\ Z \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes})}{\Gamma \vdash \mathsf{Bij}\ x_1\ Y\ Z \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Bij}\ x_1\ Y\ Z), \mathsf{Yes})}$$

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

$$\textsc{BijSubset}\quad \frac{\Gamma_{\mathsf{Bij}}(x) = (Y_2, Z_2) \qquad \Gamma \vdash \mathsf{Query}\ (Z_1 = Z_2 \wedge Y_1 \subseteq Y_2) \rightarrow_{\mathsf{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Bij}\ x\ Y_1\ Z_1 \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Bij}\ x\ Y_1\ Z_1), \mathsf{Yes})}$$

$$\textsc{InjFromBij}\quad \frac{\Gamma_{\mathsf{Bij}}(x_1) = (Y_2, Z) \qquad \Gamma \vdash \mathsf{Query}\ (Y_1 \subseteq Y_2) \rightarrow_{\mathsf{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Inj}\ x_1\ Y_1 \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Inj}\ x_1\ Y_1), \mathsf{Yes})}$$

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

$$\textsc{FilterRange}\quad \frac{\begin{array}{c}\Gamma_{\mathsf{Filt}}(x_1) = (x_2, \lambda\,(i_1 : 0..e_1)\ .\ p, (\lambda\,(D)\ .\ \mathsf{true}, \lambda\,(D)\ .\ \mathsf{false})) \qquad \Gamma(x_2) = \lambda\,(i_2 : 0..e_1)\ .\ e_2 \\ \mathsf{fresh}\ x_3 \qquad \Gamma \vdash \lambda\,(i_2 : 0..e_1)\ .\ [\,p[i_1 := i_2]\,] * x_2(i_2) + [\,\neg p[i_1 := i_2]\,] * \bot \rightsquigarrow f \\ \Gamma, x_3 \mapsto f \vdash \mathsf{Range}\ x_3\ Y \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes})\end{array}}{\Gamma \vdash \mathsf{Range}\ x_1\ Y \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Range}\ x_1\ Y), \mathsf{Yes})}$$

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

$$\textsc{MonoConcat}\quad \frac{\begin{array}{c}\Gamma(x_1) = f_1 +\!+ f_2 \qquad \mathsf{fresh}\ x_2, x_3 \qquad \Gamma, x_2 \mapsto f_1 \vdash \mathsf{Mono}\ x_2 \prec \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes}) \\ \Gamma, x_3 \mapsto f_2 \vdash \mathsf{Mono}\ x_3 \prec \rightarrow_{\mathsf{Prop}} (\Gamma'', \mathsf{Yes}) \qquad \Gamma, x_2 \mapsto f_1, x_3 \mapsto f_2 \vdash \mathsf{Query}\ (x_2(|x_2| - 1) \prec x_3(0))\end{array}}{\Gamma \vdash \mathsf{Mono}\ x_1 \prec \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Mono}\ x_1 \prec), \mathsf{Yes})}$$

$$\textsc{NoProp}\quad \frac{}{\Gamma \vdash \mathsf{True} \rightarrow_{\mathsf{Prop}} (\Gamma, \mathsf{Yes})}$$

(More inference rules are given for this judgment below.)

Low-level rules. QueryRange reifies the proof obligation for Range $x\ e_1..e_1$ by querying the solver to verify that the values of x_1 are in $\{e_1, \ldots, e_2 - 1\}$.

QueryMono reifies the proof obligation for Mono $x \prec$ by querying whether $x(i) \prec x(j)$ for $i < j$ in the domain of x.

QueryEquiv reifies the proof obligation for Equiv $x\ e$, simply checkign that x has a scalar index function and that its value equals e.

QueryInjEq and QueryInjNeq together reify the proof obligation for Inj $x\ Y$. The premises build a sufficient query at two (different) indices. The restriction of x to the preimage of Y is encoded using a a guard on the queries expression $(x(i) \in Y \wedge x(j) \in Y)$. The solver may either disprove this guard or assume it and prove the corresponding predicate expression to prove the query.

QueryBij reifies the proof obligation for Bij $x\ Y\ Z$. Like in QueryInjEq/Neq, the bijectivity is restricted to the values in x's domain that map to the set Y. The first premise checks that x is injective when restricted to the preimage of Y. The second premise checks that the image Z is a subset of Y. The remaining premises check that the number of indices that x map to Y equals the

size of the image Z. Together with the injectivity requirement, this ensures that restricting x to the preimage Y results in a bijective function with image Z.

Specifically, the third premise looks up the index function of x, which is assumed to have been rewritten to a fixed point already. The fourth premise checks that the solver can show that each term in the guarded expression for x's body is in Y or not in Y. While this query is clearly a tautology, our solver may return Unknown for either disjunct in which case the entire query returns Unknown. (We spefically disallow rewriting the query by using the single step $\rightarrow_{\mathrm{Q}}$ relation rather than the rewrite-then-query $\rightsquigarrow_{\mathrm{Q}}$ relation, which may rewrite the query to true using boolean algebra.) Once we know that the solver can show whether each term is in Y or not, we can use the rewrite system to discriminate in-bounds terms from out-of-bounds terms. In the fifth premise, construct an index function that identifies the terms that are in Y from those that are not: for all in-bounds terms $e_j \in Y$ is rewritten to true. The rewrite rule JoinGuards will join all those terms under one guard because their value (true) does not depend on the guard. The remaining guards correspond to the out-of-bounds indcies. We use X to denote some subset of the term labels corresponding to the in-bounds terms. The remaining premises construct a count using the corresponding predicates and check this count against the size of Z.

QueryInvFiltPart reifies the proof obligation for

$$\mathsf{InvFiltPart}\ x\ Z\ (\lambda\ (i : 0..e)\ .\ p_f)\ (\lambda\ (i : 0..e)\ .\ p_1, \ldots, \lambda\ (i : 0..e)\ .\ p_n).$$

The first premise checks that x is an array of permutation indices: x restricted to the preimage of Z is a bijection $Z \rightarrow Z$. The second premise checks that the size of Z equals the number of elements that remain after filtering x by p_f. The third premise looks up the index function of x to get its domain. The fourth premise checks that the partition predicates $p_1, \ldots, p_n$ are mutually exclusive. To keep the rule readable, we write $\forall q, r \in \{1, \ldots, n\}\ .\ q < r \Rightarrow \ldots$ to generate $\frac{n(n-1)}{2}$ queries for all combinations of q and r where $q < r$. This is not part of the query itself. Similarly for the remaining premises. The fifth premise ensures that permuting an array by gathering its values at indices x yields a stable filter and/or partition (i.e., order is preserved after filtering and within each partition). This is done by checking sortedness of the index array x within each partition for $q = r$. The fifth premise also ensures ordering across partitions: x permutes indices so that partition 1 comes before partition 2, and so on, when $i < j$ and $x(i)$ is in a partition that precedes $x(j)$ (i.e., queries where $q < r$). The sixth premise analogously ensures ordering across partitions when $i < j$ and $x(i)$ is in a partition that succeeds $x(j)$ (i.e., queries where $q > r$).

QueryFiltPart reifies the proof obligation for

$$\mathsf{FiltPart}\ y\ x\ (\lambda\ (i : 0..e)\ .\ p_f)\ (\lambda\ (i : 0..e)\ .\ p_1, \ldots, \lambda\ (i : 0..e)\ .\ p_n).$$

The first premise checks that y is the result of filtering x by p_f using a scatter. Filtering using scatter yields an index function of the shown form via Scatter1 or Scatter2 (a gather over an index array z^{-1}). The second premise checks that the index array z is an inverse filtering partition.

Complete property verification rules (2/4) (low-level) $\boxed{\Gamma \vdash P \rightarrow_{\mathsf{Prop}} (\Gamma', A)}$

QueryRange

$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\ (i : 0..e_{dom})\ .\ e_x \qquad \Gamma, \mathsf{Range}\ i\ 0..e_{dom} \vdash \mathrm{Query}_x\ (e_1 \leq e_x < e_2) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Range}\ x\ e_1..e_2 \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Range}\ x\ e_1..e_2), \mathsf{Yes})}$$

QueryMono

$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \text{fresh } j \qquad \Gamma, \mathsf{Range}\ j\ 0..e_1, \mathsf{Range}\ i\ 0..j \vdash \mathsf{Query}_x\ (x(i) \prec x(j)) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Mono}\ x \prec \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Mono}\ x \prec), \mathsf{Yes})}$$

QueryEquiv

$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\,()\,.\,e_x \qquad \Gamma \vdash \mathsf{Query}_x\ (e_x = e_1) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Equiv}\ x\ e_1 \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Equiv}\ x\ e_1), \mathsf{Yes})}$$

QueryInjEq

$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \text{fresh } j \qquad \Gamma, \mathsf{Range}\ i\ 0..e_1, \mathsf{Range}\ j\ 0..e_1 \vdash \mathsf{Query}\ ([\,x(i) \in Y \wedge x(j) \in Y \wedge x(i) = x(j)\,] * (i = j)) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Inj}\ x\ Y \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Inj}\ x\ Y), \mathsf{Yes})}$$

QueryInjNeq

$$\frac{\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_1)\,.\,e_2 \qquad \text{fresh } j \qquad \Gamma, \mathsf{Range}\ i\ 0..j, \mathsf{Range}\ j\ 0..e_1 \vdash \mathsf{Query}_x\ ([\,x(i) \in Y \wedge x(j) \in Y\,] * (x(i) \neq x(j))) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathsf{Inj}\ x\ Y \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Inj}\ x\ Y), \mathsf{Yes})}$$

QueryBij

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Inj}\ x\ Y \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes}) \\ \Gamma \vdash \mathsf{Query}\ (e_{Z_a}..e_{Z_b} \subseteq Y) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \qquad \Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_{dom})\,.\,[\,p_1\,] * e_1 + \cdots + [\,p_n\,] * e_n \\ \Gamma, \mathsf{Range}\ i\ 0..e_{dom} \vdash \mathsf{Query} \begin{pmatrix} [\,p_1\,] * (e_1 \in Y \vee e_1 \notin Y) \\ \ldots \\ [\,p_n\,] * (e_n \in Y \vee e_n \notin Y) \end{pmatrix} \rightarrow_{\mathrm{Q}} \mathsf{Yes} \\ \Gamma \vdash \lambda\,(i : 0..e_{dom})\,. \begin{cases} [\,p_1\,] * (e_1 \in Y) \\ \ldots \\ [\,p_n\,] * (e_n \in Y) \end{cases} \rightsquigarrow \lambda\,(i : 0..e_{dom})\,. \begin{cases} [\,\bigvee_{j \in X} p_j\,] * \mathsf{true} \\ \ldots \\ \sum_{j \in \{1,\ldots,n\} \setminus X} [\,p_j\,] * e'_j \end{cases} \\ \text{where } X \text{ is some subset of } \{1, \ldots, n\} \\ \text{fresh } k \qquad \Gamma \vdash \sum_{k=0}^{e_{dom}} (\sum_{j \in X} p_j[i := k]) \rightsquigarrow e_{x_{size}} \\ \Gamma \vdash e_{Z_b} - e_{Z_a} \rightsquigarrow e_{Z_{size}} \qquad \Gamma \vdash \mathsf{Query}\ (e_{Z_{size}} = e_{x_{size}}) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \mathsf{Bij}\ x\ Y\ e_{Z_a}..e_{Z_b} \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Bij}\ x\ Y\ e_{Z_a}..e_{Z_b}), \mathsf{Yes})}$$

QueryInvFiltPart

$$\frac{\begin{array}{c}\Gamma_{\mathsf{lxfn}}(x) = \lambda\,(i : 0..e_{dom})\,.\,e_x \qquad \Gamma \vdash \mathsf{Bij}\ x\ e_{Z_a}..e_{Z_b}\ e_{Z_a}..e_{Z_b} \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes}) \\ \text{fresh } k \qquad \Gamma \vdash \mathsf{Query}\left(e_{Z_b} - e_{Z_a} = \sum_{k=0}^{|x|} (p_f[i := k])\right) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \\ \forall q, r \in \{1, \ldots, n\}\,.\,q < r \Rightarrow (\Gamma, \mathsf{Range}\ i\ 0..e_{dom} \vdash \mathsf{Query}\ (\neg p_q \vee \neg p_r) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}) \qquad \text{fresh } j \\ \forall q, r \in \{1, \ldots, n\}\,.\,q \leq r \Rightarrow (\\ \Gamma, \mathsf{Range}\ i\ 0..j, \mathsf{Range}\ j\ 0..e_{dom} \vdash \mathsf{Query}\ ([\,p_q \wedge p_r[i := j] \wedge p_f \wedge p_f[i := j]\,] * (x(i) < x(j))) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}) \\ \forall q, r \in \{1, \ldots, n\}\,.\,q > r \Rightarrow (\\ \Gamma, \mathsf{Range}\ i\ 0..j, \mathsf{Range}\ j\ 0..e_{dom} \vdash \mathsf{Query}\ ([\,p_q \wedge p_r[i := j] \wedge p_f \wedge p_f[i := j]\,] * (x(i) > x(j))) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes})\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{InvFiltPart}\ x\ e_{Z_a}..e_{Z_b}\ (\lambda\,(i : 0..e_{dom})\,.\,p_f)\ (\lambda\,(i : 0..e_{dom})\,.\,p_1, \ldots, \lambda\,(i : 0..e_{dom})\,.\,p_n) \\ \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{InvFiltPart}\ x\ e_{Z_a}..e_{Z_b}\ (\lambda\,(i : 0..e_{dom})\,.\,p_f)\ (\lambda\,(i : 0..e_{dom})\,.\,p_1, \ldots, \lambda\,(i : 0..e_{dom})\,.\,p_n)), \mathsf{Yes})\end{array}}$$

QueryFiltPart

$$\frac{\Gamma_{\text{lxfn}}(y) = \lambda\,(i : 0..\textstyle\sum_{j=0}^{e_1}(p_f(i)))\,.\,[\,\text{true}\,] * x(z^{-1}(i)) \qquad \Gamma \vdash \text{InvFiltPart } z\ 0..\textstyle\sum_{j=0}^{e_1}(p_f(i))\ (\lambda\,(i : 0..e_1)\,.\,p_f)\ (\lambda\,(i : 0..e_1)\,.\,p_1, \ldots, \lambda\,(i : 0..e_1)\,.\,p_n) \rightsquigarrow (\Gamma', \text{Yes})}{\begin{array}{l}\Gamma \vdash \text{FiltPart } y\ x\ (\lambda\,(i : 0..e_1)\,.\,p_f)\ (\lambda\,(i : 0..e_1)\,.\,p_1, \ldots, \lambda\,(i : 0..e_1)\,.\,p_n) \\ \quad \rightarrow_{\text{Prop}} ((\Gamma, \text{FiltPart } y\ x\ (\lambda\,(i : 0..e_1)\,.\,p_f)\ (\lambda\,(i : 0..e_1)\,.\,p_1, \ldots, \lambda\,(i : 0..e_1)\,.\,p_n)), \text{Yes})\end{array}}$$

(More inference rules are given for this judgment below.)

Property verification over array rows. The For property makes it possible to express other properties over the inner dimensions of an array. The For rule creates a new index function by dropping the outer dimension, then verifies P over this function where i_1 is a free variable. One rule handles all properties over multi-dimensional arrays. The rule simply recurses until the property P is not For, in which case the rules previously presented are tried.

A rule similar to For could match flattened dimensions. (We don't use this in the paper.)

FiltPart and InvFiltPart over flattened arrays must be handled specially because the predicate arguments, which are represented as index functions, must also have their outer dimension "dropped". We give rules for 2D flattened arrays, which is used in the evaluation.

For-InvFiltPart-Flat and For-FiltPart-Flat are like QueryInvFiltPart and QueryFiltPart except over flat 2D arrays. Specifically, For-InvFiltPart-Flat checks that the index function of x_1 is a flat 2D array, that the predicate index functions range over this flat domain, and rewrites each predicate index function into new index functions where the "outer" dimension is droppped. The results are then used to prove InvFiltPart over a new one-dimensional index function where the "outer" dimension k is a free variable with a range property on it. Specifically, For-FiltPart-Flat checks that y is of the expected form discussed for QueryFiltPart, that the predicate index functions range over y's flat domain, infers the size of the filtered flat array and then uses this to construct the image Z when proving InvFiltPart.

Complete property verification rules (3/4) $\boxed{\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma', A)}$

$$\text{For}\ \frac{\Gamma(x_1) = \lambda\,(i_1 : 0..e_1, i_2 : 0..e_2, \ldots, i_n : 0..e_n)\,.\,e_x \qquad \text{fresh } x_2 \qquad \Gamma, \text{Range } k\ 0..e_1, x_2 \mapsto \lambda\,(i_2 : 0..e_2, \ldots, i_n : 0..e_n)\,.\,e_x[i_1 := k] \vdash P(x_2) \rightarrow_{\text{Prop}} (\Gamma', \text{Yes})}{\Gamma \vdash \text{For } (k : 0..e_1)\ P(x_1) \rightarrow_{\text{Prop}} ((\Gamma, \text{For } (k : 0..e_1)\ P(x_1)), \text{Yes})}$$

For-InvFiltPart-Flat

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

For-FiltPart-Flat

$$\frac{\Gamma(y) = \lambda\,(i_1 : 0..e_1 \times i_2 : 0..e_2)\;.\;x(z^{-1}(e_{idx})) \qquad \text{fresh } j_1, j_2, j_3 \\ e_{idx} \text{ unifies with } \textstyle\sum_{j_1=0}^{i_1-1}(e_2[i_1 := j_1]) + i_2 \qquad \Gamma \vdash \text{Query}\left(e_3 = \sum_{i_1=0}^{e_1-1}(e_2)\right) \rightsquigarrow_{\text{Q}} \text{Yes} \\ \Gamma, \text{Range } k\ 0..e_1 \vdash \textstyle\sum_{j_1=0}^{k-1}(\sum_{j_2=0}^{e_2[i_1:=j_1]-1}(p_f[i_3 := \sum_{j_3=0}^{j_1-1}(e_2[i_1 := j_3]) + j_2])) \rightsquigarrow e_{len}}{\Gamma \vdash \text{For } (k : 0..e_1)\ (\text{InvFiltPart } z\ (e_{len}..e_{len}[k := k+1])\ (\lambda\,(i_3 : 0..e_3)\;.\;p_f)\ (\lambda\,(i_3 : 0..e_3)\;.\;p_1, \ldots, \lambda\,(i_3 : 0..e_3)\;.\;p_n))}$$

$$\Gamma \vdash \text{For } (k : 0..e_1)\ (\text{FiltPart } y\ x\ (\lambda\,(i_3 : 0..e_3)\;.\;p_f)\ (\lambda\,(i_3 : 0..e_3)\;.\;p_1, \ldots, \lambda\,(i_3 : 0..e_3)\;.\;p_n)) \\ \rightarrow_{\text{Prop}} ((\Gamma, \text{For } (k : 0..e_1)\ (\text{FiltPart } y\ x\ (\lambda\,(i_3 : 0..e_3)\;.\;p_f)\ (\lambda\,(i_3 : 0..e_3)\;.\;p_1, \ldots, \lambda\,(i_3 : 0..e_3)\;.\;p_n))), \text{Yes})$$

(More inference rules are given for this judgment below.)

Failure to verify. Finally, properties fail to verify if none of the displayed rules match.

Complete property verification rules (4/4) $\boxed{\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma', A)}$

PropFail

$$\frac{\text{No other rule applies.}}{\Gamma \vdash P \rightarrow_{\text{Prop}} (\Gamma, \text{Unknown})}$$

(End of relation.)

B.4.2 Property inference ($\Gamma \vdash (y, E^{\oslash}) \rightarrow_{\text{Infer}} \Gamma'$). The property layer infers properties from the source code using the $\rightarrow_{\text{Infer}}$ relation. The judgment $\Gamma \vdash (y, E^{\oslash}) \rightarrow_{\text{Infer}} \Gamma'$ says that y bound to $E^{\oslash}$ updates Γ to Γ' with inferred properties. AppProp instantiates a function's postcondition for an application of that function.

WhileLoopProp and ForLoopProp are like AppProp, but for loops. All loop restricitons on pre- and postconditions have already been checked in their conversion rules WhileLoop and ForLoop. But because the loop invariant (pre/postconditions) were checked under the assumption of the loop conditions (x_n is true for while loop and B is positive)—and those conditions may never be true (and also go out of scope in the for-loop case)—we check it once more on the initial values without extending Γ.

If no other rule applies, InferNothing infers nothing.

Complete property inference rules $\boxed{\Gamma \vdash P \rightarrow_{\text{Infer}} \Gamma'}$

AppProp

$$\frac{\Gamma_{\text{Def}}(F) = ((x'_1, P_1)\ \ldots\ (x'_n, P_n), (y', P_{post}), f)}{\Gamma \vdash (y, F\ x_1\ \ldots\ x_n) \rightarrow_{\text{Infer}} \Gamma, P_{post}[x'_1 := x_1, \ldots, x'_n := x_n, y' := y]}$$

WhileLoopProp

$$\frac{\Gamma_{\text{Def}}(F) = (\overline{(x', P)}^{(n)}, (\overline{y'}^{(n)}, \overline{P'}^{(n)}), \_) \\ \Gamma \vdash P'_1[y'_1 := x^0_1] \rightarrow_{\text{Prop}} (\Gamma'_1, \text{Yes}) \qquad \ldots \qquad \Gamma \vdash P'_n[y'_1 := x^0_1, \ldots, y'_n := x^0_n] \rightarrow_{\text{Prop}} (\Gamma'_n, \text{Yes})}{\Gamma \vdash (\overline{y}^{(n)}, \vdash \textbf{loop}\ \overline{x}^{(n)} = \overline{x^0}^{(n)}\ \textbf{while}\ x_n\ \textbf{do}\ F\ \overline{x}^{(n)}) \rightarrow_{\text{Infer}} \Gamma', P'_1[y'_1 := y_1], \ldots, P'_n[y'_1 := y_1, \ldots, y'_n := y_n]}$$

ForLoopProp

$$\frac{\Gamma_{\text{Def}}(F) = (\overline{(x', P)}^{(n)}\ (x_{n+1}, \text{Range } x_{n+1}\ 0..e_1), (\overline{y'}^{(n)}, \overline{P'}^{(n)}), \_) \\ \Gamma \vdash P'_1[y'_1 := x^0_1] \rightarrow_{\text{Prop}} (\Gamma'_1, \text{Yes}) \qquad \ldots \qquad \Gamma \vdash P'_n[y'_1 := x^0_1, \ldots, y'_n := x^0_n] \rightarrow_{\text{Prop}} (\Gamma'_n, \text{Yes})}{\Gamma \vdash (\overline{y}^{(n)}, \textbf{loop}\ \overline{x}^{(n)} = \overline{x^0}^{(n)}\ \textbf{for}\ x_{n+1} < B\ \textbf{do}\ F\ \overline{x}^{(n+1)}) \rightarrow_{\text{Infer}} \Gamma', P'_1[y'_1 := y_1], \ldots, P'_n[y'_1 := y_1, \ldots, y'_n := y_n]}$$

InferPart

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

FiltPartPresInj'

$$\frac{\Gamma_{\mathsf{FiltPart}}(x_1) = (x_2, f_1, (f_2, \ldots, f_n)) \qquad \Gamma_{\mathsf{Inj}}(x_2) = \mathsf{Inj}\ x_2\ Y}{\Gamma \vdash (x_1, \_) \rightarrow_{\mathsf{Infer}} \Gamma, \mathsf{Inj}\ x_1\ Y}$$

InferNothing

$$\frac{\text{No other inference rule applies.}}{\Gamma \vdash (y, E^{\emptyset}) \rightarrow_{\mathsf{Infer}} \Gamma}$$

Property inference can readily be extended. For example, two bijectivity properties can be unified over a conditional expression whose branches are variable names bound before the conditional (i.e., they do not depend on x_c):

BijOverConditional

$$\frac{\Gamma_{\mathsf{Bij}}(x_1) = (Y_1, Z_1) \qquad \Gamma_{\mathsf{Bij}}(x_2) = (Y_2, Z_2) \qquad \Gamma \vdash \mathsf{Query}\ (Z_1 = Z_2 \wedge Y_1 \subseteq Y_2) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}}{\Gamma \vdash \mathbf{let}\ x_{res} = \mathsf{if}\ x_c\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2\ \mathbf{in}\ E \rightarrow_{\mathsf{Infer}} \Gamma, \mathsf{Bij}\ x_{res}\ Y_1\ Z_1}$$

(And similarly picking Y_2 to be smaller.)

SOACs like reduce-by-index also admit inferring properties even when we cannot create an index function for it. For example, reduce-by-index with min/max operator allows us to infer the range of the result as the union of ranges of the neutral element and argument arrays. Reduce-by-index with addition and a constant array similarly lets us infer ranges.

B.4.3 Queries over multi-dimensional index functions. For each query over a 1-dimensional index function, an n-dimensional index function x_1 requires $2^n - 1$ queries as shown in the paper. The corresponding low-level rules for Range, Equiv, Mono and Inj properties are straightforward to write down but very verbose. (See QueryMono-nD below.) Supporting low-level multi-dimensional rules for Bij, InvFiltPart and FiltPart is beyond the scope of this work, but most high-level rules already support multiple dimensions.

QueryMono-nD

$$\frac{\begin{array}{c}\Gamma_{\mathsf{lxfn}}(x_1) = \lambda\,(i_1 : 0..e_1, i_2 : 0..e_2, \ldots, i_n : 0..e_n)\,.\,e_{body} \qquad \text{fresh } j_1, j_2, \ldots, j_n \\ (\Gamma, \mathsf{Range}\ j_1\ 0..e_1, \mathsf{Range}\ i_1\ 0..j_1, \mathsf{Range}\ j_2\ 0..e_2, \mathsf{Range}\ i_2\ 0..e_2, \ldots, \mathsf{Range}\ j_n\ 0..e_n, \mathsf{Range}\ i_n\ 0..e_n) \vdash \mathsf{Query}\ (x_1(i_1, \ldots, i_n) \prec x_1(j_1, \ldots, j_n)) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \\ (\Gamma, \mathsf{Range}\ j_1\ 0..e_1, \mathsf{Equiv}\ j_1\ i_1, \mathsf{Range}\ j_2\ 0..e_2, \mathsf{Range}\ i_2\ 0..j_2, \ldots, \mathsf{Range}\ j_n\ 0..e_n, \mathsf{Range}\ i_n\ 0..e_n) \vdash \mathsf{Query}\ (x_1(i_1, \ldots, i_n) \prec x_1(j_1, \ldots, j_n)) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes} \\ \ldots \\ (\Gamma, \mathsf{Range}\ j_1\ 0..e_1, \mathsf{Equiv}\ j_1\ i_1, \mathsf{Range}\ j_2\ 0..e_2, \mathsf{Equiv}\ j_2\ i_2, \ldots, \mathsf{Range}\ j_n\ 0..e_n, \mathsf{Range}\ i_n\ 0..j_n) \vdash \mathsf{Query}\ (x_1(i_1, \ldots, i_n) \prec x_1(j_1, \ldots, j_n)) \rightsquigarrow_{\mathrm{Q}} \mathsf{Yes}\end{array}}{\Gamma \vdash \mathsf{Mono}\ \prec x_1 \rightarrow_{\mathsf{Prop}} \Gamma, \mathsf{Mono}\ \prec x_1}$$

B.4.4 Rewriting queries continued: Equality solving. The algebra layer does not a have a syntactic notion of equality and does not have the capability to reason at the level of properties. Therefore, equality queries are augmented with transitive equalities and equalities inferred from properties before being relegated to the algebra layer. For example, the below rule uses x's injectivity to equate two indexing expressions.

$$\boxed{\Gamma \vdash \mathsf{Query}_x\ (e) \rightarrow \mathsf{Query}_x\ (e')}$$

RewriteQueryInjEq

$$\frac{\Gamma_{\mathsf{Inj}}(x) = Y \qquad \Gamma \vdash \mathsf{Query}\ (e_1 \in Y \wedge e_2 \in Y)}{\Gamma \vdash \mathsf{Query}_{x'}\ (x(e_1) = x(e_2)) \rightarrow \mathsf{Query}_{x'}\ (x(e_1) = x(e_2) \wedge e_1 = e_2)}$$

B.5 Algebra layer

B.5.1 Algebra language syntax.

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

B.5.2 Leading variables. The leading variable is defined as follows

$$
\begin{aligned}
\mathrm{lv}(x) &= \{x\} \\
\mathrm{lv}(\vee_{\perp} X) &= X \\
\mathrm{lv}(a[e]) &= \mathrm{lv}(a) \\
\mathrm{lv}(\textstyle\sum a[e_1 : e_2]) &= \mathrm{lv}(a)
\end{aligned}
$$

B.5.3 Simplification strategy. The full set of rules used by the algebraic solver to normalize lowered expressions are shown in Appendix B.5.4.

Judgments $\Delta \vdash s \longrightarrow e$ rewrite symbol s, which may be nested inside an expression e. Rules Equiv1 and Equiv2 replace a symbol with its binding in Δ_{Equiv}. (In Equiv2, just one symbol has to be 1 to make the disjunction 1.) EmptySum replaces sums of provably empty slices with 0. ExtendSum extends a sum slice with an end index bound in Δ_{Equiv}. (An analogous rule exists for the end of a slice.) The slice's lower bound must be at most 1 greater than its upper bound, which guarantees that the extended slice contains said element. SingletonSum collapses a singleton slice to an index while doing away with the sum. PeelEquiv1 and 2 extract off an index bound in Δ_{Equiv} from the beginning of a sum slice. (Analogous rules exist for the end of a slice.) PeelOnRange, extracts an end-of-slice-sum index that has a more specialized range than the one of the whole array. (Only used in the solver algorithm).

Judgments $\Delta \vdash e \longrightarrow e$ rewrite any two (not necessarily adjacent) terms in an expression. JoinSums1 rewrites two sum slices that are in continuation of each other into one sum, requiring a similar side condition to ExtendSum. JoinSums2 extends a slice sum with an end index. JoinSums4 simplifies two overlapping sum slices over arrays of disjunctions, whose sequences of variable names do not overlap, but are in the same $\Delta_{\perp}$ class. ElimSumOverlap eliminates the common part of two overlapping sum slices that are subtracted. (An analogous rule handles the case where $e_1 \le e_3 \le e_4 \le e_2$.) ExtractSumOverlap extends this simplification of slice-sum subtraction to arrays of disjunctions. (An analogous rule handles the case where $e_1 \le e_3 \le e_2 \le e_4$.)

Simplify() applies these rules in the following order until a fixed point is reached:

(1) Equiv1, Equiv2, EmptySum.
(2) ExtendSum to a fixed point.
(3) JoinSums1–3, ElimSumOverlap, ExtractSumOverlap to a fixed point.
(4) EmptySum
(5) SingletonSum, PeelEquiv1–2, EmptyDisjunction, to a fixed point.
(6) PeelOnRange.

PeelOnRange is applied to increase the accuracy of the solver. When rewrites are used to normalize expressions (outside the solver algorithm), PeelOnRange is not applied.

B.5.4 *Complete algebraic rewrite rules.*

$\boxed{\Delta \vdash s \longrightarrow e}$

$$\textsc{Equiv1}\ \frac{\Delta_{\mathsf{Equiv}}(s) = e}{\Delta \vdash s \longrightarrow e} \qquad \textsc{Equiv2}\ \frac{\exists i \in 1, \ldots, n\ .\ \Delta_{\mathsf{Equiv}}(x_i[e]) = 1}{\Delta \vdash \vee_\perp\{x_1, \ldots, x_n\}[e] \longrightarrow 1} \qquad \textsc{EmptySum}\ \frac{\mathrm{Solve}(\Delta, e_1 > e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow 0}$$

$$\textsc{ExtendSum}\ \frac{\Delta \vdash a[e_1 - 1] \longrightarrow e_3 \qquad \mathrm{Solve}(\Delta, e_1 \leq e_2 + 1)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow \sum a[e_1 - 1 : e_2] - e_3} \qquad \textsc{SingletonSum}\ \frac{\mathrm{Solve}(\Delta, e_1 = e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow a[e_1]}$$

$$\textsc{PeelEquiv1}\ \frac{\Delta_{\mathsf{Equiv}}(a[e_1]) = e_3 \qquad \mathrm{Solve}(\Delta, e_1 \leq e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow \sum a[e_1 + 1 : e_2] + e_3} \qquad \textsc{PeelEquiv2}\ \frac{\Delta \vdash a[e_1] \longrightarrow 1 \qquad \mathrm{Solve}(\Delta, e_1 \leq e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow \sum a[e_1 + 1 : e_2] + 1}$$

$$\textsc{EmptyDisjunction}\ \frac{}{\Delta \vdash \vee_\perp\{\} \longrightarrow 0} \qquad \textsc{PeelOnRange}\ \frac{a \notin \mathrm{dom}(\Delta_{\mathsf{Range}}) \qquad a[e_2] \in \mathrm{dom}(\Delta_{\mathsf{Range}}) \qquad \mathrm{Solve}(\Delta, e_1 \leq e_2)}{\Delta \vdash \sum a[e_1 : e_2] \longrightarrow \sum a[e_1 : e_2 - 1] + a[e_2]}$$

$\boxed{\Delta \vdash e \longrightarrow e}$

$$\textsc{JoinSums1}\ \frac{\mathrm{Solve}(\Delta, e_2 + 1 = e_3 \wedge (e_1 \leq e_2 + 1 \vee e_3 \leq e_4 + 1))}{\Delta \vdash t \cdot \sum a[e_1 : e_2] + t \cdot \sum a[e_3 : e_4] \longrightarrow t \cdot \sum a[e_1 : e_4]}$$

$$\textsc{JoinSums2}\ \frac{\mathrm{Solve}(\Delta, e_2 + 1 = e_3 \wedge e_1 \leq e_2 + 1)}{\Delta \vdash t \cdot \sum a[e_1 : e_2] + t \cdot a[e_3] \longrightarrow t \cdot \sum a[e_1 : e_2 + 1]}$$

$$\textsc{JoinSums3}\ \frac{\mathrm{Solve}(\Delta, e_1 - 1 = e_3 \wedge e_1 \leq e_2 + 1)}{\Delta \vdash t \cdot \sum a[e_1 : e_2] + t \cdot a[e_3] \longrightarrow t \cdot \sum a[e_1 - 1 : e_2]}$$

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

$$\textsc{ElimSumOverlap}\ \frac{n_1 = -n_2 \qquad \mathrm{Solve}(\Delta, e_1 \leq e_3 \leq e_4 \leq e_2)}{\Delta \vdash n_1 \cdot t \cdot \sum a[e_1 : e_2] + n_2 \cdot t \cdot \sum a[e_3 : e_4] \longrightarrow n_1 \cdot t \cdot \sum a[e_1 : e_3 - 1] + n_2 \cdot t \cdot \sum a[e_4 + 1 : e_2]}$$

$$\textsc{ExtractSumOverlap}\ \frac{n_1 = -n_2 \qquad \bigcup_{x \in X} \Delta_\perp(x) = \bigcup_{y \in Y} \Delta_\perp(y) \qquad X \cap Y \neq \emptyset}{\begin{array}{l}\Delta \vdash n_1 \cdot t \cdot \sum(\vee_\perp X)[e_1 : e_2] + n_2 \cdot t \cdot \sum(\vee_\perp Y)[e_3 : e_4] \\ \quad \longrightarrow n_1 \cdot t \cdot \sum(\vee_\perp(X - Y))[e_1 : e_2] + n_2 \cdot t \cdot \sum(\vee_\perp(Y - X))[e_3 : e_4] \\ \quad\ + n_1 \cdot t \cdot \sum(\vee_\perp(X \cap Y))[e_1 : e_2] + n_2 \cdot t \cdot \sum(\vee_\perp(X \cap Y))[e_3 : e_4]\end{array}}$$

C Dafny: exclusive scan details

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

$$c = \text{sum}^{inc}\ b\ =\ [b_1,\ b_1 + b_2,\ \ldots,\ b_1 + \ldots + b_n]\ =\ [0,\ a_1,\ a_1 + a_2,\ \ldots,\ a_1 + \ldots + a_{n-1}]$$

Dafny can prove that each element in c is a partial sum over b (via the three queries $|a| > 0 \implies b[0] = 0, 0 < i < |a| \implies b[i] = a[i-1]$, and $0 \le i < |a| \implies c[i] = \sum_{k=0}^{i} b[k]$), but we could not prove that they are partial sums over the original a array: $0 \le i < |a| \implies c[i] = \sum_{k=0}^{i-1} a[k]$.

D Evaluation

D.1 Segmented parallel operations

Normalization of flags*' index function.* The full definition of flags is given below. The poscondition captures that the length of the output array is equal to the sum of the row sizes.

```
def sum (xs : []i64) : i64 =
  let ys = scan (λx y. x + y) 0 xs
  in if |xs| > 0 then ys[|xs| − 1] else 0
def flags (s : i64[] | Range s 0..∞) (xs : []i64 | Equiv |s| |xs|) : []i64 | λys. Equiv |ys| (sum s) =
  let rotated = map (λi. if i == 0 then 0 else s[i − 1]) (0..|s|)
  let offsets = scan (λx y. x + y) 0 rotated
  let indices = map (λsize i. if size > 0 then i else −1) s offsets
  let n = if |s| > 0 then s[|s| − 1] + offsets[|s| − 1] else 0
  let zeros = map (λx. 0) (0..n)
  let f = scatter zeros indices xs in f
```

Upon reaching the scatter in flags, the environment Γ includes the index functions

$$indices = \lambda\,(i_1 : 0..|sizes|)\ .\ [\,s(i_1) > 0\,] * \textstyle\sum_{j=0}^{i_1-1}(s(j)) + [\,s(i_1) \le 0\,] * -1,$$
$$n = \lambda\,()\ .\ [\,\text{true}\,] * \textstyle\sum_{j=0}^{|sizes|-1}(s(j)),$$
$$zeros = \lambda\,(i_1' : 0..\textstyle\sum_{j=0}^{|sizes|-1}(s(j)))\ .\ [\,\text{true}\,] * 0.$$

Using SCATTER3, *flags*' index function is

$$\lambda\,(i_1 : 0..|s| \times i_2 : 0..\textstyle\sum_{j=0}^{i_1}(s(j)) - \sum_{j=0}^{i_1-1}(s(j)))\ .$$
$$[\,i_2 = 0 \wedge s(i_1) > 0\,] * xs(i_1) + [\,i_2 \ne 0 \vee s(i_1) \le 0\,] * zeros(\textstyle\sum_{j=0}^{i_1-1}(s(j)) + i_2),$$

The domain simplifies to $i_1 : 0..|s| \times i_2 : 0..s(i_1)$ via algebraic rewriting (refer to the algebra layer section). The guarded expression simplifies as

(1) REWRITEBODY adds $0 < |sizes|$, Range i_1 $0..|sizes|$, $0 < s(i_1)$, and Range i_2 $0..s(i_1)$ to the environment;[9]
(2) QUERYPRED simplifies $s(i_1) > 0$ to true and $s(i_1) \le 0$ to false using the ranges added to the environment;
(3) SUB replaces indexing into *zeros* with its guarded expression ($[\,\text{true}\,] * 0$); and
(4) Boolean factoring eliminates conjunction with true and disjunction with false.

Yielding the following normalized form:

$$\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\ .\ [\,i_2 = 0\,] * xs(i_1) + [\,i_2 \ne 0\,] * 0.$$

[9] If a dimension is empty, the guarded expression is irrelevant, hence each dimension is assumed non-empty.

seg_ids. We trace how our system derives the index function for a complex segmented operation, seg_ids, which takes an array of row sizes and expands it to an array in which the row number is replicated according to the row sizes:

$$\overbrace{[2,0,3,3]}^{s} \quad \overset{\text{seg\_ids}}{\longrightarrow} \quad [0,0,2,2,2,3,3,3]$$

Note that it handles the empty row. It has index function

$$\lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\;.\;[\,\text{true}\,] * i_1$$

Applying this index function to a valid index will give you the corresponding value for i_1. This is literally $\%_{i_1:0..|s|\times i_2:0..s(i_1)}(\cdot)$ expressed in the source program!

It's implementaiton uses seg_scan, which scans each row in an irregular array independently. It has index function (source functions zip and unzip are no-ops)

$$\lambda\,(i_0 : 0..|flags|)\;.\;[\,i_0 = 0 \vee flags(i_0)\,] * xs(i_0) + [\,i_0 \neq 0 \wedge \neg flags(i_0)\,] * (\circlearrowleft + xs(i_0))$$

```
def seg_scan (flags : [n]bool) (xs : [n]i64) : [n]i64
  unzip
    (scan
      (λ(f_x, x) (f_y, y). (f_x ∨ f_y, if f_y then y else x + y))
      (false, 0)
      (zip flags xs))

def seg_ids (s : i64 | Range s 0..∞) : []i64 | λz. Equiv |z| (sum s)
  let x = map (λi. i + 1) (0..|s|)
  let f = make_flags s
  let x = map (λi. i > 0) f
  let y = map (λi. if i == 0 then 0 else i − 1) f
  in seg_scan x y
```

We show how the index function for seg_ids is derived by our system. For brevity, we focus on the function's return whose derivation is the most complicated. The intermediate variables have index functions:

$$f = \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\;.\;[\,i_2 = 0\,] * (i_1 + 1) + [\,i_2 \neq 0\,] * 0$$
$$x = \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\;.\;[\,i_2 = 0\,] * 1 + [\,i_2 \neq 0\,] * 0$$
$$y = \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\;.\;[\,i_2 = 0\,] * i_1 + [\,i_2 \neq 0\,] * 0$$

Continuing to derive the function's return (Fig. 1):

(1) Substituting the formal arguments for the actual arguments in seg_scan's index function.
(2) Since x has size $\sum_{j=0}^{|s|-1}(s(j))$, PropFlatten propagates the flattened domain from $flags_{bool}$, expressing i_0 in terms of s, i_1 and i_2.
(3) SubFlat substitutes the first occurence of $x(\sum_{j=0}^{i_1-1}(s(j)) + i_2)$
(4) Then using SolveIdx1 $\%_{i_1:0..|s|\times i_2:0..s(i_1)}(\sum_{j=0}^{i_1-1}(s(j))+i_2)$ is simplified to i_1 because $\sum_{j=0}^{i_1-1}(s(j)) \leq \sum_{j=0}^{i_1-1}(s(j)) + i_2 < \sum_{j=0}^{i_1}(s(j))$ is provable given the environment's ranges. Consequently, the sums $\sum_{j=0}^{i_1-1}(s(j))$ and $-\sum_{j=0}^{\%_{i_1:0..|s|\times i_2:0..s(i_1)}(\sum_{j=0}^{i_1-1}(s(j))+i_2)-1}(s(j))$ cancel out.
(5) Using Hoist (in the context of guards, 1 and 0 are true and false, respectively).
(6) Under the environment induced by the domain and precondition on s (Range s 0..∞ and Range i_2 0..$s(i_1)$), QueryPred falsifies $i_2 \neq 0 \wedge \sum_{j=0}^{i_1-1}(s(j)) + i_2 = 0$ in the second guard,

$$\lambda\,(i_0 : 0..|x|)\,.\,[\,i_0 = 0 \vee x(i_0)\,] * y(i_0) + [\,i_0 \neq 0 \wedge \neg x(i_0)\,] * (\circlearrowleft + y(i_0))$$

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

$$= \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.$$

$$\begin{cases} [\,\sum_{j=0}^{i_1-1}(s(j)) + i_2 = 0 \\ \quad \vee \Big([\,(\sum_{j=0}^{i_1-1}(s(j)) + i_2) - \sum_{j=0}^{\%_{i_1:0..|s|\times i_2:0..s(i_1)}(\sum_{j=0}^{i_1-1}(s(j))+i_2)-1}(s(j)) = 0\,] * 1 \\ \quad\quad + [\,(\sum_{j=0}^{i_1-1}(s(j)) + i_2) - \sum_{j=0}^{\%_{i_1:0..|s|\times i_2:0..s(i_1)}(\sum_{j=0}^{i_1-1}(s(j))+i_2)-1}(s(j)) \neq 0\,] * 0\Big)\,] * y(\sum_{j=0}^{i_1-1}(s(j)) + i_2) \\ [\,\sum_{j=0}^{i_1-1}(s(j)) + i_2 \neq 0 \wedge \neg x(\sum_{j=0}^{i_1-1}(s(j)) + i_2)\,] * (\circlearrowleft + y(\sum_{j=0}^{i_1-1}(s(j)) + i_2)) \end{cases}$$

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

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

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

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

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

$$= \lambda\,(i_1 : 0..|s| \times i_2 : 0..s(i_1))\,.\,[\,\text{true}\,] * i_1$$

Fig. 1. Rewriting seg_ids's index function to a fixed point using the presented rules.

simplifying it to false $\vee\, i_2 \neq 0 \wedge$ false (which is false and thus eliminated). Using JoinGuards, and basic Boolean algebra the guards are simplified further.

(7) Using the same rules for the second occurence of x,
(8) Similarly for y.
(9) Using SegRecReplicate.

D.2 seg_partition

The implementation is shown below. It uses the functions *sum*, *flags*, *seg_ids* and *seg_scan* defined above.

```
def seg_partition (s : []i64 | Range s 0..∞) (xs : []i64 | Equiv |xs| (sum s)) (ps : []bool | Equiv |xs| |ps|)
  : []f64 | λys. For (k : 0..|s|) Part ys xs (λi. ps[i])
  let a = map (λi. i + 1) (0..|s|)
  let f = flags s a
  let ids = seg_ids s
  let flags_T = map (λp. if p then 1 else 0) ps
  let flags_F = map (λt. 1 − t) flags_T
  let is_T = seg_scan f flags_T
  let tmp = seg_scan f flags_F
```

let *seg_ends* = scan $(\lambda x\ y.\ x + y)\ 0\ s$
let *lst* = map $(\lambda n\ i.$ if $n = 0$ then -1 else $is_T[i-1])\ s\ seg\_ends$
let is_F = map $(\lambda t\ i.\ t + lst[i])\ tmp\ ids$
let *offs* = map $(\lambda i.$ if $i > 0$ then $seg\_ends[i-1]$ else $0)\ ids$
let *is* = map $(\lambda c\ i_T\ i_F\ offset.$ if c then $offset + i_T - 1$ else $offset + i_F - 1)\ ps\ is_T\ is_F\ offs$
let *zeros* = map $(\lambda x.\ 0)\ xs$
in scatter *zeros is xs*

seg_partition takes the data array x, its non-negative row sizes s, and a boolean array p denoting the partition for each row. It returns a flat irregular array where each row has been partitioned into two by p, as illustrated by the input-output pair below.

$$\begin{aligned} &s = [2, 0, 3, 3] \\ &x = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8] \\ &p = [\text{false}, \text{true}, \text{true}, \text{true}, \text{false}, \text{true}, \text{false}, \text{true}] \end{aligned} \quad \xrightarrow{\text{seg\_partition}} \quad [x_2, x_1, x_5, x_3, x_4, x_6, x_8, x_7]$$

Preconditions require that x's length is equal to the sum of the row sizes s, and that x and p have the same length. The postcondition asserts the partition property for each row:

$$\Gamma \vdash \text{For}\ (k : 0..|s|)\ \text{Part}\ z\ x\ (\lambda i.\ p[i])$$

and aliases

$$\Gamma \vdash \text{For}\ (k : 0..|s|)\ \text{FiltPart}\ z\ x\ (\lambda i.\ \text{true})\ (\lambda i.\ p[i], \lambda i.\ \neg p[i])$$

which requires

For-FiltPart-Flat

$$\frac{\begin{array}{c}\Gamma(y) = \lambda\ (i_1 : 0..|s| \times i_2 : 0..s(i_1))\ .\ x(z^{-1}(\sum_{j'=0}^{i_1-1}(s(j')) + i_2)) \\ \text{fresh}\ j_1, j_2, j_3 \qquad \sum_{j'=0}^{i_1-1}(s(j')) + i_2\ \text{unifies with}\ \sum_{j_1=0}^{i_1-1}(s(j_1)) + i_2 \\ \Gamma \vdash \text{Query}\ \left(\sum_{j=0}^{|s|-1}(s(j)) = \sum_{i_1=0}^{|s|-1}(s(i_1))\right) \rightsquigarrow_{\text{Q}} \text{Yes} \\ \Gamma, \text{Range}\ k\ 0..|s| \vdash \sum_{j_1=0}^{k-1}(\sum_{j_2=0}^{s(j_1)-1}(\text{true})) \rightsquigarrow \sum_{j_1=0}^{k-1}(s(j_1))\end{array}}{\begin{array}{l}\Gamma \vdash \text{For}\ (k : 0..|s|) \\ (\text{InvFiltPart}\ z\ (\sum_{j_1=0}^{k-1}(s(j_1))..\sum_{j_1=0}^{k}(s(j_1)))\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \text{true})\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ p, \lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \neg p))\end{array}}$$

$$\begin{array}{l}\Gamma \vdash \text{For}\ (k : 0..|s|) \\ (\text{FiltPart}\ y\ x\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \text{true})\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ p, \lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \neg p)) \\ \rightarrow_{\text{Prop}} \\ ((\Gamma, \text{For}\ (k : 0..|s|)\ (\text{FiltPart}\ y\ x\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \text{true})\ (\lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ p, \lambda\ (i_3 : 0..\sum_{j=0}^{|s|-1}(s(j)))\ .\ \neg p))), \text{Yes})\end{array}$$

where Γ includes Range s $0..\infty$ (in Γ_{Range}), and predicates $|x| = (\sum_{j=0}^{|s|-1}(s(j)))$ and $|x| = |p|$ (in Γ_{Pred}).

This requires verifying the partition property over each row in the irregular representation. The injectivity query shown in the main paper

$$\text{Inj}\ indices'\ \textstyle\sum_{j=0}^{i_1-1}(s(j))..\sum_{j=0}^{i_1}(s(j)).$$

spawns several queries, one of which lowers to

$$\Delta_{\text{Untrans}} = \{x_1 \mapsto p(xs(\square)),\ x_2 \mapsto \neg p(xs(\square))\}$$
$$\Delta_{\bot} = \{x_1 \mapsto \{x_2\},\ x_2 \mapsto \{x_1\}\}$$
$$\Delta_{\text{Equiv}} = \{x_1[\textstyle\sum s[0 : i_1 - 1] + i_2] \mapsto 1,\ x_2[\sum s[0 : i_1 - 1] + i_2] \mapsto 0,\ x_1[\sum s[0 : i_1 - 1] + i_2'] \mapsto 0,\ x_2[\sum s[0 : i_1 - 1] + i_2'] \mapsto 1\}$$

$$\Delta_{\text{Range}} = \left\{ \begin{array}{l} \min\{0\} \le s \le \max\{\} \\ \min\{2\} \le |s| \le \max\{\} \\ \min\{0\} \le x_1 \le \max\{1\} \\ \min\{0\} \le x_2 \le \max\{1\} \\ \min\{0\} \le i_1 \le \max\{|s|\} \\ \min\{1\} \le i_2' \le \max\{s(i_1)\} \\ \min\{0\} \le i_2 \le \max\{i_2' - 1\} \end{array} \right\}$$

$$\textstyle\sum s[0 : i_1 - 1] + \sum x_1[\sum s[0 : i_1 - 1] : \sum s[0 : i_1 - 1] + i_2 - 1] \le \sum s[0 : i_1 - 1] + i_2' + \sum x_1[\sum s[0 : i_1 - 1] + i_2' + 1 : \sum s[0 : i_1] - 1]$$

Solve returns Yes for this query using the same three-step method described in Appendix B.5.3. Variables i_2, i_2' and x_1 are eliminated in succession (replaced using Δ_{Range}), before arriving at a trivial query.

D.3 max_match

The implementation of the relevant function is shown below. (The full program has been verified, however.)

```
def filter_by (cs : []bool) (xs : []t | Equiv |cs| |xs|)
    : []t  |  λys. Filt ys xs (λi. cs[i])
  Analogous to partition in the main document; uses scatter.

def get_smallest_pairs
    (es : []i64 | Range es 0..|H|)
    (idx : []i64 | Inj is (−∞)..∞, Equiv |es| |is|)
    (H : []i64)
    : ([]i64, []i64) | λ(es', is'). Inj es' (−∞)..∞, Inj is' (−∞)..∞
  let cs = map (λi j. H[i] = j) es idx
  in (filter_by cs es, filter_by cs idx)
```

We verify that scatters are safe (use unique indices) in the maximal matching graph algorithm from the Problem Based Benchmark Suite [2],[10] that iteratively filters graph edges using scatters.

We show the most challenging kernel to verify here. It takes an array of edges, *es*, an array of indices (edge numbers), *idx*, and a histogram that stores the smallest index for each edge, H. H is used to filter the edges and indices, returning two new (smaller) arrays. We wish to verify that the resulting arrays have unique values. Notably, the input *es* may have duplicates, but its filtered output must not. We automatically verify this property using the injective property on *idx*.

The index function for the filtered *es* says that it is a permutation of *es* via an uninterpreted function x (an out-of-scope variable for the permutation indices produced in filter_by) and that the array's size is equal to the number of values kept by the filter predicate.

$$es' = \lambda\left(i : 0..\textstyle\sum_{j=0}^{|es|-1}(H(es(j)) = idx(j))\right).\ es(x^{-1}(i))$$

[10] Using an existing implementation from the Futhark benchmarks found at https://github.com/diku-dk/futhark-benchmarks.

From filter_by's postcondition, es' has the property Filt es' es $(\lambda\,(i : 0..|es|)\,.\,H(es(j)) = idx(j))$. The verification of Inj es' $(-\infty)..\infty$:

FilterInj

$$\frac{\begin{array}{c}\Gamma_{\mathsf{FiltPart}}(es') = (es,\ \lambda\,(i : 0..|es|)\,.\,H(es(i)) = idx(i),\ (\lambda\,(i : 0..|es|)\,.\,\mathsf{true}, \lambda\,(i : 0..|es|)\,.\,\mathsf{false}))\\ \Gamma(es) = \lambda\,(i : 0..|es|)\,.\,[\,\mathsf{true}\,] * es(i)\\ \Gamma \vdash \lambda\,(i : 0..|es|)\,.\,[\,H(es(i)) = idx(i)\,] * es(i) + [\,H(es(i)) \neq idx(i)\,] * \bot \rightsquigarrow f\\ \text{fresh } x_1 \qquad \Gamma, x_1 \mapsto f, H(es(i)) = idx(i) \vdash \mathsf{Inj}\ x_1\ (-\infty)..\infty \rightarrow_{\mathsf{Prop}} (\Gamma', \mathsf{Yes})\end{array}}{\Gamma \vdash \mathsf{Inj}\ es'\ (-\infty)..\infty \rightarrow_{\mathsf{Prop}} ((\Gamma, \mathsf{Inj}\ es'\ (-\infty)..\infty), \mathsf{Yes})}$$

The premise Inj x_1 $(-\infty)..\infty$ is verified via its proof obligation:

$$\begin{aligned}&\forall i, j \in 0..|x_1|\ .\ x_1(i) \in Y \wedge x_1(i) = x_1(j) \Rightarrow i = j\\ &\quad \vee\ \forall i, j \in 0..|x_1|\ .\ x_1(i) \in Y \wedge x_1(j) \in Y \wedge i \neq j \Rightarrow x_1(i) \neq x_1(j)\end{aligned}$$

where the first disjunct is successful.

Which after substitution and hoisting gives rise to four queries. One of the form

$$\forall i, j \in 0..|x_1|\ .\ H(es(i)) = idx(i) \wedge H(es(j)) = idx(i) \wedge es(i) = es(j) \Rightarrow i = j$$

This is proven by first enriching the query with transitive equalities $H(es(i)) = H(es(j))$ and $is(i) = is(j)$ (both due to $es(i) = es(j)$). QueryRewriteInjEq then rewrites this to the query of the form (see Appendix B.4.4)

$$\begin{aligned}\forall i, j \in 0..|x_1|\ .\ (&H(es(j)) = idx(i) \wedge H(es(j)) = idx(i) \wedge es(i) = es(j)\\ &\wedge H(es(i)) = H(es(j)) \wedge is(i) = is(j) \wedge i = j) \Rightarrow i = j\end{aligned}$$

using the injectivity of is which implies that $i = j$. This trivial query is finally lowered to an algebraic expression which simplifies to 1 (true) after substituting in equivalences.

The other three queries are trivial because $\bot$ appears in their antecedents ($\bot \wedge \cdots \Rightarrow \ldots$), falsifiying the antecedent in each query.

D.4 FFT

The implementation is shown below.

```
def loop_body (n : i64 | Range n 1..∞) (o : []f32 | Equiv |o| 2^n) (x : []f32 | Equiv |x| 2^n) (q : i64 | Range q 0..n) =
  let iss1 = map (λk. map (λj. k * 2^(q+1) + j) 0..2^q) 0..2^(n-q-1)
  let iss2 = map (λk. map (λj. k * 2^(q+1) + j + 2^q) 0..2^q) 0..2^(n-q-1)
  let vss1 =
    map (λk.
      map (λj. let t = o[r * j] * x[k * L + j + L/2]
               in  x[k * L + j] + t) 0..L/2) 0..r
  let vss2 =
    map (λk.
      map (λj. let t = o[r * j] * x[k * L + j + L/2]
               in  x[k * L + j] - t) 0..L/2) 0..r
  let is = concat (flatten iss1) (flatten iss2)
  let vs = concat (flatten vss1) (flatten vss2)
  let x' = scatter x is vs
  in  x'

def fft (n : i64 | Range n 1..∞) (o : []f32 | Equiv |o| 2^n) (x : []f32 | Equiv |x| 2^n) =
  loop x' = x for q < n do loop_body n o x' q
```

D.5 k-means

The implementation of the key kernel is shown below.

```
def loop_body =
    (cluster : []f32)
    (indices : []i64 | Range indices 0..|cluster|)
    (values : []f32 | Equiv |indices| |values|)
    (pointers : []i64 | Range pointers 0..|values|)
    (row : i64 | Range row 0..(|pointers| - 1))
    (j : i64 | Range j 0..pointers[row + 1] - pointers[row])
    : f32 =
      let index_start = pointers[row]
      let element_value = values[index_start + j]
      let column = indices[index_start + j]
      let cluster_value = cluster[column]
      let diff = element_value - 2 · cluster_value
      let res = correction + diff · element_value
      in res

def kmeans_ker
    (cluster : []f32)
    (indices : []i64 | Range indices 0..|cluster|)
    (values : []f32 | Equiv |indices| |values|)
    (pointers : []i64 | Range pointers 0..|values|)
    (row : i64 | Range row 0..(|pointers| - 1))
    : f32 =
  loop correction = 0 for j < pointers[row + 1] - pointers[row] do loop_body cluster indices values pointers row j
```