

Automated Proofs of Reduce Specifications for Imperative Data Structures

Jamie Kai and Alexander J. Summers^[0000–0001–5554–9381]

Department of Computer Science, University of British Columbia, Vancouver, BC
Canada, V6T 1Z4

`jkai@student.ubc.ca` `alex.summers@ubc.ca`

Abstract. Inductive definitions are ubiquitous for program verification, but their recursion order often necessitates substantial manual inductive proofs to reflect state changes. This is especially unfortunate when reasoning about common properties whose meaning is not order-sensitive, such as sums, maximal elements or multiset contents of a data structure. This paper presents an alternative reasoning concept: a generic *reduce* operator, whose instantiations capture diverse properties of a variety of data structures. We define the key laws for reasoning with reduce terms, and an automated strategy for effectively applying these laws to reason about imperative verification problems without user intervention. We further define a novel *framing* technique, stylised to our new operator. Our technique is implemented and we demonstrate its efficiency and automation on a selection of verification challenges from the literature.

Keywords: Program Verification · Specifications · Proof Automation.

1 Introduction

Formal reasoning about updates to heap data structures is a well-studied problem. Many verification techniques and tools exist to tackle this problem, and typically appeal to *inductively-defined specification features*, both to define the program state constituting a data structure and properties of its current values. For example, in separation logics, inductive predicates are often used to define data structures; recursive functions are used to describe properties of their elements.

When data structures are modified, proof must capture how these inductively-defined properties *change* (or are preserved). Such arguments require *induction proofs*, which can be implicit when the program traverses a data structure following the *same recursion scheme* as the specifications. However, when code performs mutations in different iteration orders (e.g. walking a linked list from its tail), or in a random access fashion (e.g. swapping two array elements), these proofs must be explicit and easily become cumbersome: inductive lemmas are needed for each combination of property of interest and type of modification.

In this paper, we present a novel alternative formalism centred around the notion of the *reduce* of a binary operator over the elements of a data structure. Such *reduce* operations denote a computation whose result is insensitive to the

orders of evaluation and access, which is a particularly awkward case for inductive reasoning. We explain the mathematical principles underlying this idea, and show its benefits for reasoning about data structures: it allows proofs that avoid (or complement) explicit inductive reasoning, and can be instantiated uniformly for diverse data structures and properties. We then show how this reasoning can be automated; without explicit lemmas, induction or manual proof annotations. We show that efficient automation is challenging, and present a novel framing technique that substantially improves performance, enabling tractable and simple automated proofs about *reduce* properties in practice. Our key contributions are:

- The definition of a higher-order *reduce* operator that encodes *reduce* specifications for imperative data structures, and key reasoning axioms for this operator that enable simple, flexible proofs (Sec. 3).
- An automated technique for generating necessary instances of the *reduce* decomposition axioms in verification tasks, based on static analysis (Sec. 4).
- An efficiently *indexical framing* technique to reason about *reduce* terms by abstracting from the heap to a set-theoretic index domain (Sec. 5).
- An implementation of our techniques as a plugin for the Viper verification language, along with an evaluation that demonstrates the expressiveness, automation, and performance enabled by these contributions (Sec. 6).

We give a more-detailed motivation for our techniques in Sec. 2. Finally, we discuss related work (Sec. 7) and propose future developments in Sec. 8.

2 Motivation: Difficulties with Inductive Reasoning

In this section, we illustrate the verification challenges concerning updates to data structures with inductively-defined properties. Our work is aimed at verification of imperative code (modifications often in-place; aliasing memory is possible), but our techniques are designed to be compatible with most languages, techniques and tools. We give examples in pseudocode similar to the Dafny language [19].

2.1 A Running Example with Inductive Reasoning

As a simple running example, consider the `swap` method in Listing 1.1. The `multiset(A) == old(multiset(A))` specification (in which `old(e)` refers to the value of `e` in the initial state of the called method) expresses that across the execution of the method `swap`, the multiset of elements in the array `A` has not changed. Similarly, the maximum value in `A` will not have changed.

```

1 method swap(A: Array<int>, j: int, k: int)
2   requires 0 <= j, k < A.len && j != k
3   ensures multiset(A) == old(multiset(A))
4   ensures max(A) == old(max(A))
5   {
6       var tmp : int := A[j]
7       A[j] := A[k]
8       A[k] := tmp

```

9 }
}

Listing 1.1: Running Example: Array Swap

When aiming to verify these program properties, two key questions arise: (1) How can properties such as `multiset(A)` and `max(A)` be defined? (2) How can we reason about these properties as program state is modified? In most proof assistants [28, 25, 32, 22] and deductive verifiers (e.g. [18, 19, 14, 24, 5]) the answer to (1) is (user-specified) *inductive definitions*, e.g. defining a recursive function `multiset(A, l, h)` to compute the multiset of elements from indices `l` to `h-1`. Each call adds the multiset `{A[l]}` to `multiset(A, l+1, h)`, until `l==h` yields an empty multiset. A similar idea works for `max`, but avoiding empty ranges.

With inductive definitions, the answer to (2) is explicit induction proofs: since (unlike the inductive definitions) our example jumps directly to two unknown points in the array, a (nested) inductive argument will be needed to reflect the changes to the program state in these inductive definitions, justifying e.g. that `max(A, l, h)` is the largest of `max(A, l, i)`, `A[i]` and `max(A, i+1, h)`¹. Once proved, the inductive lemmas (for each of `multiset` and `max`) must also be manually applied before and after *every* modification of an array location².

2.2 An Alternative Argument Ambition

Inductive methods are powerful and general, but for notion such as multiset elements their proof principles do not capture simple or easily automatable formal arguments. A simpler argument might go as follows:

1. The multiset over all array indices equals $\{v_j, v_k\} \uplus R$: the size-two multiset from the indices j and k summed with the multiset R from all *other* indices.
2. Across lines 6-8, none of these other indices is modified; R is unchanged.
3. Across line 7, the value at index j becomes v_k (the original value at index k).
4. Across line 8, the value at index k becomes that stored in `tmp`, which is v_j .
5. The final multiset across all indices is that from the indices j and k ($\{v_j, v_k\}$) summed with the (unchanged) rest R , and so is equal to the original multiset.

That this argument requires no induction, but requires splitting and recombining our view of the multiset property flexibly over the index set (the array indices). A perfectly analogous argument works equally well for the preservation of the maximum value in the array. Our argument does require support for tracking updates to individual variables and heap locations (handled by all standard techniques), as well as some ability to *frame* properties (R , above) dependent

¹ Note the need to express the elements traversed *earlier* than the modified point; for more-general tree-like data structures this requires further auxiliary definitions and lemmas, analogous to the reasoning about *zippers* in functional languages [5].

² Depending on the verification methodology, alternative proof strategies may be feasible. For example, one can abstract an array's current content to a mathematical sequence, and define multisets and maxima in terms of this sequence. Assuming all definitions are inductive, this does not avoid the difficulties above.

on unmodified portions of the program state. Framing is usually well-supported (in substantially different ways) by modern techniques and tools, e.g. via *reads* clauses in Dafny, or separation logic preconditions in Verifast or Viper.

This paper shows how to formalise and efficiently automate arguments of this nature, applied to diverse data structures and to a variety of commonly-occurring properties (called *reduce* properties) that we define precisely in the next section.

3 The Reduce Operator and Reduce Specifications

Many useful properties concerning data structure elements can be expressed as *comprehensions*, often defined in mathematics via an iterated binary operator (e.g. the “big sum” $\sum_{i=1}^k A[i]$, “big product” $\prod_{i=1}^k A[i]$, or even “big conjunction” $\bigwedge_{i=1}^k \phi(A[i])$). In this section, we present the foundational contribution of this paper: a novel means of generalising these notions to formally specify properties over the points of *data structures*, specifically in cases where the underlying operator is *associative and commutative*. By analogy with the functional programming concept, we call these *reduce specifications*, although our notion is formally justified and reasoned with independently of any computation / evaluation order.

3.1 Background: Program States and Algebraic Preliminaries

We adopt the view of a memory *heap* as a finite partial function from *locations* (memory addresses) to *values*, defined for (at least) the locations that a program accesses [37]. We assume all programs and specifications to be well-typed.

Definition 1 (Program State and Dereference Function). *We assume a type of program states Σ , ranged over by σ , and, for every type T , a type $\text{loc}(T)$ for the locations (ranged over by ℓ) in a program state storing values of type T .*

We assume the existence of a dereference function $\text{drf} :: \Sigma \rightarrow \forall T. (\text{loc}(T) \rightarrow T)$ so that for $\sigma \in \Sigma$, the partial evaluation $\text{drf}(\sigma, \cdot) :: \forall T. (\text{loc}(T) \rightarrow T)$ is a partial function (polymorphic over T); and $\text{dom}(\sigma)$, the set of locations for which $\text{drf}(\sigma, \cdot)$ is defined, is finite. A value at location ℓ in state σ is $\text{drf}(\sigma, \ell)$; abbreviated $\text{drf}_\sigma(\ell)$.

Preliminaries. By default sets and multisets are finite in this paper. The powerset of a set X is denoted $\mathcal{P}^{\text{fin}}[X]$; the multisets of X are denoted $\mathcal{M}^{\text{fin}}[X]$. We use the brackets $\{\cdot\}$ for sets and $\{\!\!\{ \cdot \}\!\!\}$ for multisets. We use standard set operator notation (overloading \emptyset for the empty (multi)set), and \uplus for multiset sum (which adds multiplicities).

A *commutative semigroup* is a pair (X, \oplus) where X is a set and \oplus an associative, commutative binary operator on this set: for every $x, y, z \in X$ we have $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ and $x \oplus y = y \oplus x$. A *commutative monoid* (X, \oplus, e) is a commutative semigroup with an identity element $e \in X$ satisfying $x \oplus e = e \oplus x = x$ for every $x \in X$. We use only *commutative* semigroups and monoids: we hereafter refer to these simply as *semigroups* and *monoids*.

3.2 The Reduce Operator

Our novel reduce operator aims to capture properties which amount to the iteration of a semigroup’s operator over the (finite multiset) of values computed from the points stored in a heap data structure. The (customisable) ingredients of our generic operator (explained below) are illustrated pointwise as follows:

$$\begin{array}{ccccccc} \text{index} & \xrightarrow{\text{receiver}} & \text{heap} & \xrightarrow{\text{drf}_\sigma(\cdot)} & \text{location} & \xrightarrow{\text{mapping}} & \text{semigroup} \\ (\text{index set}) & & \text{location} & & \text{value} & & \text{element} \end{array}$$

Here, indices may be from any chosen type; a *receiver* function maps each index to the heap location defining a data structure element. Dereferencing this (single) heap location provides a value which is coerced into an element of a chosen semigroup by a chosen *mapping* function³. Our reduce operator lifts this idea to sets of indices, and denotes the (single) semigroup value corresponding to iterating the semigroup operator over the corresponding generated values.

Concretely, we can represent e.g. the multiset of elements in array by choosing integer indices, a receiver function to model array layout, a mapping function to inject values into singleton multisets, and the semigroup to be multisets with operator sum. Alternatively, we can represent e.g. the minimum integer value stored in the nodes of a graph by choosing the node references themselves be indices, a receiver function identifying their value locations, an identity mapping function, and the semigroup to be integers with operator (binary) minimum.

Injective Receivers. While our operator supports customisable receiver functions, we impose the restriction that a receiver must be *injective* (at least over the sets of indices it is applied to). This restriction guarantees a one-to-one correspondence between (sets of) data structure heap locations and (sets of) indices. Without this restriction, a change to a single heap location could affect an unknown and unbounded number of indices; we demonstrate in our evaluation that this restriction did not prevent support for a variety of important examples. Note that there is no such requirement for either the data structure content (which can contain duplicate values) or the mapping function (which may collapse values).

Comprehensions as homomorphisms. A key property (*cf.* Sec. 2) for our reduce operator to satisfy, is that when its index set is split in two, the results of two subcomputations can be recombined with the corresponding semigroup operator. For example, the sum of elements in an array is the sum of its odd-indexed elements *plus* the sum of its even-indexed elements. Formally, our operator should preserve this separability from disjoint pairs of index sets to subexpressions in the chosen semigroup. We now define *reduce* operators, which express this property with reference to the program heap.

Definition 2 (Reduce Operators). *Let $\sigma \in \Sigma$ be a state. For a chosen index type I , we call $F \in \mathcal{P}^{fin}[I]$ a filter. For a given state and filter, a function*

³ This name is chosen by analogy to map-reduce as a common programming idiom.

$r :: I \rightarrow \text{loc}(V)$ is a well-formed receiver function (written $\text{wf}_\zeta(r, F)$) if it acts injectively on F and its image on F is contained in $\text{dom}(\sigma)$.

Let S be a semigroup with operator $\oplus :: S \times S \rightarrow S$, and a function $m :: V \rightarrow S$ be a mapping. A reduce operator (for chosen I, r, m, S, \oplus components) is a function $\text{reduce}^{\oplus, m, r}(\sigma)(F) :: \Sigma \rightarrow (\mathcal{P}^{\text{fin}}[I] \rightarrow S)$, defined for arguments σ, F whenever $\text{wf}_\zeta(r, F)$ holds. For a given σ , the operator is defined by lifting the function $m \circ \text{drf}_\sigma \circ r : I \rightarrow S$ to set-valued arguments such that

- for every i in I with $r(i) \in \text{dom}(\sigma)$, we have $\text{reduce}^{\oplus, m, r}(\{i\}) = m(\text{drf}_\sigma(r(i)))$;
- for all disjoint sets $F_1, F_2 \in \mathcal{P}^{\text{fin}}[I]$, if $\text{wf}_\sigma(r, F_1)$ and $\text{wf}_\sigma(r, F_2)$ hold we have $\text{reduce}^{\oplus, m, r}(F_1 \cup F_2) = \text{reduce}^{\oplus, m, r}(F_1) \oplus \text{reduce}^{\oplus, m, r}(F_2)$.

We use shorthand $\text{reduce}_\sigma^{\oplus, m, r}(F)$, or $\oplus_\sigma[F]$ when \oplus and m are clear from context.

We provide a proof sketch for the existence of *reduce* operators in App. A.1.

This notion's flexibility is in its parameters: different index types and receiver mappings handle varied data structures, while different semigroups and mapping functions capture diverse properties of interest. Concrete examples include:

1. The multiset elements of array \mathbf{A} is $\text{reduce}_\sigma^{\uplus, \llbracket \cdot \rrbracket, \lambda^i. \mathbf{A}[i]}$, where $\mathbf{A}[i]$ denotes array slot locations, and the mapping $\llbracket \cdot \rrbracket$ sends values to singleton multisets.
2. The maximum integer value in array \mathbf{A} is $\text{reduce}_\sigma^{\max, \lambda^i. i, \lambda^i. \mathbf{A}[i]}$. A semigroup is necessary here: there is no unit for maximum.
3. The count of nodes in a graph \mathbf{G} satisfying predicate Q is $\text{reduce}_\sigma^{+, \mathbb{1}_Q, r}$, whose mapping is the (0/1) indicator function $\mathbb{1}_Q(n)$ and receiver is the identity.

In spite of the flexibility of our *reduce* notion, we define here a simple set of generic (for all instantiations) laws that provide its power as a reasoning principle.

Definition 3 (Reduce Laws). *The following key reasoning principles follow from the definition of (any instantiation of) our reduce operator (in each, the reduce components \oplus, m, r are implicitly for-all quantified):*

$$\forall \sigma :: \Sigma. \quad \text{reduce}_\sigma^{\oplus, m, r}(\emptyset) = e_S \quad \text{(empty)*}$$

$$\forall \sigma :: \Sigma, \forall i :: I. \quad \text{reduce}_\sigma^{\oplus, m, r}(\{i\}) = m(\text{drf}_\sigma(r(i))) \quad \text{(singleton)}$$

$$\forall \sigma :: \Sigma, \forall F, G, H :: \mathcal{P}^{\text{fin}}[I]. \quad \text{wf}_\sigma(r, H) \wedge \quad \text{(decomp)}$$

$$F \cap G = \emptyset \wedge F \cup G = H \quad \boxed{\wedge F \neq \emptyset \wedge G \neq \emptyset} \implies \\ \text{reduce}_\sigma^{\oplus, m, r}(H) = \text{reduce}_\sigma^{\oplus, m, r}(F) \oplus \text{reduce}_\sigma^{\oplus, m, r}(G)$$

$$\forall \sigma_1, \sigma_2 :: \Sigma, \forall F :: \mathcal{P}^{\text{fin}}[I]. \quad \text{wf}_{\sigma_1}(r, F) \wedge \text{wf}_{\sigma_2}(r, F) \wedge \quad \text{(extensionality)}$$

$$(\forall i \in F. \text{reduce}_{\sigma_1}^{\oplus, m, r}(\{i\}) = \text{reduce}_{\sigma_2}^{\oplus, m, r}(\{i\})) \quad \boxed{\wedge F \neq \emptyset} \implies \\ \text{reduce}_{\sigma_1}^{\oplus, m, r}(F) = \text{reduce}_{\sigma_2}^{\oplus, m, r}(F)$$

The first axiom (marked *) applies only to monoids; the conditions boxed in red (e.g. $\boxed{F \neq \emptyset}$) are necessary (to avoid empty index sets) only for semigroups.

These axioms provide the means of making our desired argument from Sec. 2.2 formal. We note, however, that our reasoning works not only to show properties are preserved, but also to general changes to the underlying data structure in question. For example, it is now simple to express and formally prove that incrementing any single array value causes its sum to increase by one. However, our end goal is not manual proof (applying these laws by hand), and we turn to their automation in the next section.

4 Automating Reduce Reasoning

Our *reduce* operator and its laws from the previous section provide a proof system one can in principle apply manually to specify and verify imperative programs. In this section, we define an automation strategy for deriving suitable applications of these laws, such that typical (all, in our evaluation) proofs are discharged automatically. We remain agnostic as to the underlying language and verification methodology, but will clarify explicitly our assumptions about what is taken care of. We do assume that programs come annotated with the specifications (including usages of our *reduce* operator) required by typical deductive verifiers (e.g. at least pre/post-conditions and loop invariants). A specification for our running swap example might, for example, look something like the following:

```

1   method swap(A: Array, j: Int, k: Int)
2   requires 0 <= j, k < A.len && j != k
3   ensures reduce(⊕, {·}, λi.A[i])([0..A.len]) ==
↪      old(reduce(⊕, {·}, λi.A[i])([0..A.len]))
4   ensures reduce(max, id, λi.A[i])([0..A.len]) ==
↪      old(reduce(max, id, λi.A[i])([0..A.len]))
5   {
6       var tmp Int := A[j]
7       A[j] := A[k]
8       A[k] := tmp
9   }
```

Listing 1.2: Running swap example with reduce specifications

We assume that operators \oplus and \max , mappings $\{\cdot\}$ and id , and receiver $\lambda i.A[i]$ are program functions that behave as expected; we discuss handling this higher-order parameterisation for tools that do not directly support it in Sec. 6. By the filter $[0..A.len]$ we mean a representation of the set $\{0, 1, \dots, A.len - 1\}$.

For the sake (only) of simplicity of presentation, in this section we will assume that all reduce terms in the problem at hand use the same receiver r and mapping m ; we will therefore adopt the shorthand notation $\overset{\oplus}{\zeta}[F]$ for $\text{reduce}_{\zeta}^{\oplus, m, r}(F)$. We define our automation strategy with respect to the *known* reduce terms: those reduce terms which have been found in the proof search so far. We write $\text{know}(\overset{\oplus}{\zeta}[F])$ to represent that the nested reduce term is known. Initially, the known reduce terms are those occurring in specifications in the program.

Our strategy will prescribe instantiations of our reduce laws (Def. 2), based on analysis of the program statements and the known reduce terms. It cannot be

totally evaluated statically (before verification), but it can be used to generate appropriate additional queries to drive the program’s verification automatically⁴.

Our strategy centres on identifying suitable *decompositions* (Axiom **(decomp)**) of the known reduce terms, such that the side effects caused by program statements can be reasoned about automatically, by connecting with underlying reasoning about the program state and by appealing to framing (when the relevant program state is untouched). In Sec. 4.1 we explain how reduce terms are decomposed and propagated across statements, as well as presenting an efficient abstraction based on the observably-relevant states. Since recursive decomposition is sometimes necessary, we take care to avoid termination pitfalls, and in Sec. 4.2 further define limited variants of our strategy which achieve better performance and retain completeness in practice. For this section, we allow the underlying verifier to freely apply the other laws of Def. 3, although we will show how to improve on this unrestricted application of **(extensionality)** in Sec. 5.

4.1 Symbolic State Labels and Statement Footprints

Our strategy requires an underlying language and verification technique which (a) can express and reliably reason about mathematical sets, (b) can identify and track precise value information for statements that are simple reads/writes to local variables and individual heap locations (aliasing expressions are permitted), (c) for each primitive statement S allow for a (possibly-symbolic) definition of its *memory footprint* $mods(S)$: a set of locations it may modify⁵. In general, this last notion can typically be defined (at least implicitly) in terms of the logic/specification features used for framing heap-based information (e.g. the “modifies” clauses in Dafny). Whatever the technique, we assume that, if σ_0 and σ_1 are the program points immediately before and after S , respectively, then for any location $\ell \in \text{dom}(\sigma_0) \setminus mods(S)$, the technique can prove that $drf_{\sigma_{S_j}}(\ell) = drf_{\sigma_{\text{pred}(S_j)}}(\ell)$ (i.e. that unmodified locations are unchanged).

Some statements cannot, by definition, change the value of a reduce term: statements S for which $mods(S)$ is always empty (such as assignments to/from only local variables), or contains only heap locations of a different *type* to those on which reduce terms depend. Based on this observation, we define a simple abstraction of the program points in the code to be verified, assigning a different (*symbolic state*) label ς_i only immediately after each statement S that might (conservatively) change the value of some reduce term in the program. We write $\varsigma_i \xrightarrow{S} \varsigma_j$ in this case, where ς_i represents the label immediately before S , and ς_j that following. Defining our strategy on pairs of labels (rather than concrete states) avoids logically-redundant applications of our **(extensionality)** axiom⁶.

⁴ For an SMT-based verifier, this can be done with quantified facts for the solver to instantiate; for a proof assistant the strategy could be implemented via tactics.

⁵ We discuss handling statements that e.g. allocate new locations later.

⁶ We will also use these labels in Sec. 5 to optimise our reasoning using Axiom **(extensionality)** further.

The statements on lines 7 and 8 both update a heap location on which reduce terms depend, and so we give a new label immediately after each. Given the receiver r of such a reduce term, we define the *index footprint* of a statement S , written fp_S^r by: $\text{fp}_S^r = \{i \in I \mid r(i) \in \text{mods}(S)\} \in \mathcal{P}^{\text{fin}}[I]$.

For example, the index footprint of (either of) the reduce terms in our running example for line 7 would be $\{j\}$ ⁷. Note that injectivity of our receivers allows us to know statically that this is the only index; this is important for connecting our reduce reasoning with underlying reasoning about individual heap locations.

Based on this notion, we can define the core idea of our decomposition strategy. The idea of this strategy is to decompose the filters of reduce terms at the boundaries of relevant statements, splitting them into the index footprints and any remaining indices; a reduce term over the latter indices should be known unchanged by extensionality and framing. This strategy is applied to each relevant statement $\varsigma \xrightarrow{S} \varsigma'$ as described by the following *update rule*:

$$\text{Upd } \varsigma \xrightarrow{S} \varsigma' \frac{\text{know}(\oplus_{\varsigma}[F] \text{ or } \oplus_{\varsigma'}[F]) \quad \text{wf}_{\varsigma}(r, F) \quad \text{wf}_{\varsigma'}(r, F) \quad \emptyset \neq \text{fp}_S^r \subsetneq F}{\left(\oplus_{\varsigma}[F] = \oplus_{\varsigma}[F \setminus \text{fp}_S^r] \oplus \oplus_{\varsigma}[\text{fp}_S^r]\right) \wedge \left(\oplus_{\varsigma'}[F] = \oplus_{\varsigma'}[F \setminus \text{fp}_S^r] \oplus \oplus_{\varsigma'}[\text{fp}_S^r]\right)}$$

This rule should be read as defining deductions we enable for the underlying verifier: if the premises hold, then the conclusion will be made available. The rule can be invoked whenever we know a reduce term either before or after the statement; in both cases the conclusions we will learn reflects two specific instantiations of our Axiom (**decomp**): in both states, we split the filter F into the corresponding index footprint and the remaining part. In practice, the premises concerning well-formedness of filters are typically satisfied automatically: since well-formedness of a receiver is downward-closed with respect to the corresponding filter, and new filters are generated only by decomposition, it suffices to check well-formedness of the original filters used in program specifications.

Returning to our running example, this strategy allows for the propagation of reduce terms both from the precondition of the method forward, and the postcondition of the method backward; for line 7, for example, these terms will be decomposed into the singleton index footprint for this statement and the remaining indices. Note that this decomposition will *recurse* in general: both the original reduce term and its two decomposed parts will e.g. be propagated forwards across line 8 by the same rule, resulting in one case in a filter with both indices i and j removed. This recursive propagation cannot, however, recurse forever: only index footprints for program statements are ever removed from a filter, and the premises of our rule include conditions to ensure that the rule cannot cause a trivial change to the filter in question: it must (provably) have some (but not all) of its indices removed.

⁷ In general, a program might also use aliasing expressions to modify relevant locations; we cannot assume the expression used in the program statement bears any relation to forms of the parameters of the reduce terms.

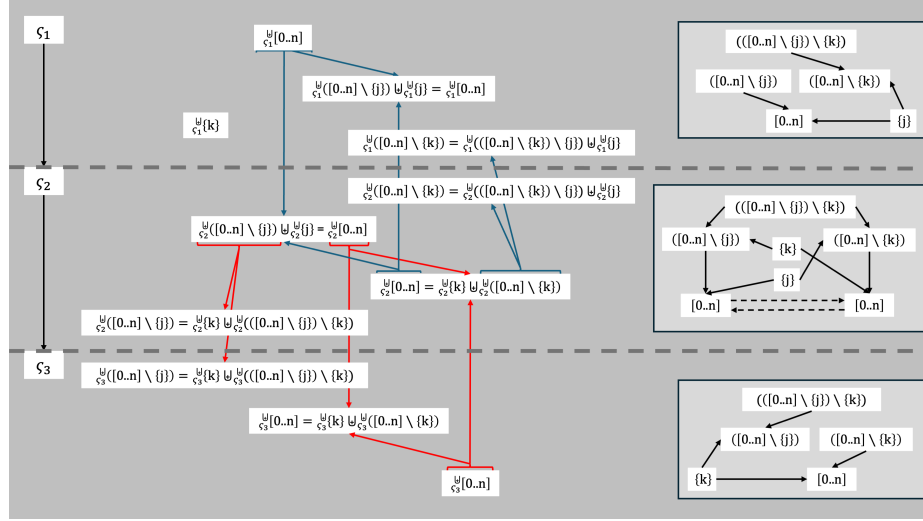


Fig. 1: Example inference for `swap` based on the automated *update rules*. Arrows indicate the automated inferences generated by *reduce* terms, starting with initial terms $\cup_{\zeta_1} [0..n]$ and $\cup_{\zeta_3} [0..n]$ from lines 3 and 4 and continuing from yielded terms. Blue arrows are instances of $[\text{Upd } \zeta_1 \rightsquigarrow \zeta_2]$, and red arrows are $[\text{Upd } \zeta_2 \rightsquigarrow \zeta_3]$.

An illustration of this strategy applied to our running example is shown in Fig. 1. Note that the decompositions enabled for this example subsume those required in our informal argument from Sec. 2.2; combined with suitable applications of **(singleton)** (applied for each singleton filter), and **(extensionality)** (applied in particular to the index set without j and k), our informal argument will be implied by the decompositions enabled. As we demonstrate in our evaluation, this (recursive) decomposition strategy is sufficient to fully automate common examples in practice; nonetheless, in the next subsection we consider restrictions focused on performance considerations.

In App. A.2, we define an extended version of this decomposition strategy to consider *frees* and *allocates* rules to decompose *reduce* terms over statements which add or remove locations from the domain of the program state; this more-general strategy is implemented in our prototype. We also extend our strategy to decompose reduce terms over single *read* locations (which can in some cases be relevant for completeness) in an analogous way to the rule presented above.

4.2 Performance, Completeness and Limited Strategies

Since our decomposition strategy can recursively decompose the same filter across the corresponding index footprints of multiple statements, it is important in practice to consider how expensive this decomposition itself could become. As already briefly discussed, the premises of our decomposition strategy impose sufficient set-theoretic conditions to guarantee that we cannot infinitely decompose

a filter (e.g. by generating new syntactically-distinct representations of the same set). We rely on these set-related queries being reliably answered by the underlying solver, but do not consider this a practical obstacle: there are, e.g., a variety of decision procedures for such finite set theories [12, 31, 17, 8, 9, 6]; recently a complete proof system for SMT was implemented for the `cvc5` solver [3].

In spite of this termination guarantee, because our rules can decompose each filter via each index statement, and potentially in various orders, in general we may still generate exponentially many *reduce* terms in the program length (technically in the number of labelled states). As a pathological example, a k -ary rotation of array values (at k distinct indices) introduces $k+1$ labelled states, but $O(2^k)$ potential decompositions of e.g. $\uplus_{s_0}[[0..n]]$ (for every order of removal of every subset of the k indices).

We observed in practice that our unrestricted strategy could become slow for such pathological examples, and as a result have defined *limited variants* of our decomposition strategy. We took inspiration from the idea of *limited functions* (originally employed in the Dafny verifier [19, 2]). The idea is to attach to each reduce term a natural number n (the *decomposition bound*), used to bound the number of decompositions of its filter. We add an extra condition to our decomposition strategy that the bound of the known term (e.g. $\oplus_{\zeta}[F]$) used must be larger than zero. If so, the analogous term in the other state (e.g. $\oplus_{\zeta'}[F]$) keeps the same bound n , but the terms (e.g. $\oplus_{\zeta}[F \setminus \mathbf{fp}_S^r]$ and $\oplus_{\zeta}[\mathbf{fp}_S^r]$) with decomposed filters get bounds $n - 1$, limiting the number of decompositions made recursively.

We obtain simple restricted variants of our decomposition strategy by choosing a depth bound n for *all* reduce terms in the program’s original specifications; this has the effect of limiting the space of decompositions possible (for example, to a linear number for $n = 1$). This substantially improves performance, but perhaps surprisingly, small values of n suffice without sacrificing completeness (automation). As a common but special case, we discovered that for examples using a *cancellative* semigroup, a bound of 1 is actually sufficient. Cancellativity is the (semigroup) property that for any fixed $z, x \in S$, if there exists $y \in S$ such that $z = x \oplus y$ then the choice of y is unique. Multiset union and integer addition are cases where cancellativity holds; it does not hold e.g. for maximum integer values. For non-cancellative examples, we found instead that a bound of 2 is always sufficient in practice. While formally justifying the sufficiency of these bounds is beyond the scope of this paper, we demonstrate in our evaluation that they work reliably in practice while achieving good performance. As an example, we show how to prove our running example with only bound 1 decompositions (a different proof from our original argument) in App. A.2.

5 Indexical Framing

Up to now, we have assumed, without a detailed analysis, that our underlying program verifier will determine the instances of Axiom (**extensionality**) to produce necessary framing arguments to verify a *reduce* specification. Since we can define the set of locations that the *reduce* operator depends on (in terms

of its receiver and filter argument), under the assumption that our verifier can prove the preservation of values for locations outside of memory footprints, it is reasonable to expect any two *reduce* applications in different states that depend on unchanged portions of the heap are provably equal.

Indeed, verification methodologies such as Dafny and Viper allow for definition of heap-dependent properties via custom *functions*, that can read from the heap and be invoked in specifications. Functions are declared along with specification of their *dependent heap locations (deps)*: the set of memory locations that a function call may possibly depend on.⁸ This gives rise to a function *framing* axiom (for each function f , and pairs of calls $f_{s_1}(\vec{x})$ and $f_{s_2}(\vec{x})$):

$$\forall s_1, s_2, \vec{x}. ((\forall \ell \in \text{deps}(f(\vec{x})). \text{drf}_{s_1}(\ell) = \text{drf}_{s_2}(\ell)) \implies f_{s_1}(\vec{x}) = f_{s_2}(\vec{x})).$$

This states that two calls to f with the same arguments, but in different states (s_1 and s_2) have equal values if the states agree on the values at all dependent heap locations.⁹ This axiom gets instantiated for all pairs of calls to f .

A body-less¹⁰ function of this kind could be used to represent our reduce feature, in particular to automate framing (Note that **(extensionality)** is implied by the framing axiom above when combined with **(singleton)**). However, we observed in practice that even simple examples encoded this way quickly became expensive. On closer analysis, this approach to framing has several performance implications: every instantiation (quadratically many in the number of calls) of the axiom above yields

- a new clause $(\exists \ell \in \text{deps}(f(\vec{x})). \text{drf}_{s_1}(\ell) \neq \text{drf}_{s_2}(\ell)) \vee f_{s_1}(\vec{x}) = f_{s_2}(\vec{x})$, equivalent to the framing axiom, and a new case-split on its disjunction;
- (if the left disjunct is not refuted) a fresh “witness” variable, i.e. *loc*-typed ℓ' that is dereferenced in both states: $\text{drf}_{s_1}(\ell')$ and $\text{drf}_{s_2}(\ell')$;
- instantiations of all other heap-related axioms relating to these dereferences.

All of these aspects slow the underlying solver. Furthermore, our automated decomposition strategy (of Sec. 4) often yields terms with filter fragments that trigger new instances of the function framing axiom (and do so needlessly, as extensionality for filter F implies extensionality for any $G \subseteq F$).

We observed that for the *reduce* operator, much of this default framing process replicates our definitions and techniques of Sec. 4. In particular, our index footprints and update rules are defined in such a way that they yield pairs of *reduce* terms for which framing arguments hold by construction. Moreover, these pairs of *reduce* terms are generated only for proof-relevant, labelled states concerning heap updates observable by their receiver component.

⁸ These are specified differently in different methodologies; e.g. in Dafny via a **reads** clause, in Viper via the separation-logic-style permissions required in a precondition. As with footprints, these may rely on arguments and conditions in program context.

⁹ These *deps* may depend only on function arguments (identical here) and specification, hence this conditional implies the dependencies are equal for both calls.

¹⁰ Specifying this with an inductive definition would defeat the motivation of our work.

In this section, we present an *indexical framing* technique that leverages index footprints and set-theoretic reasoning in the index domain to efficiently produce framing equations implied by Axiom **(extensionality)**, much in the same vein as our automated decomposition strategy produces equations implied by Axiom **(decomp)**. Indexical framing applies more generally than the case of these automated decompositions, hence we describe *framing rules* that we generate per state label in tandem with the update rules of Sec. 4. We also discuss the need to prove heap-dependent instances of **(extensionality)** when extensionality holds for reasons other than framing, and describe a strategy to allow indexical and heap-based framing to co-occur in a verification approach.

5.1 The indexical framing technique

On a technical level, our indexical framing technique is founded on a simple consequence of the footprint definitions of Sec. 4: any valid filter that is disjoint with the index footprint necessarily consists of indices that the receiver sends to the *frame* of S : those locations not in $\text{mods}(S)$.

In more detail, for a receiver $r : I \rightarrow \text{loc}(V)$ and a program statement with its pre-/post-states $\varsigma \xrightarrow{S} \varsigma'$, we assume that our verifier can describe a memory footprint $\text{mods}(S) \in \mathcal{P}^{\text{fin}}[\text{loc}(V)]$ such that for all locations ℓ :

$$\ell \notin \text{mods}(S) \implies \text{drf}_{\varsigma'}(\ell) = \text{drf}_{\varsigma}(\ell)$$

whenever both $\text{drf}_{\varsigma'}(\ell)$ and $\text{drf}_{\varsigma}(\ell)$ are defined.

For each memory footprint, we defined the index footprint $\text{fp}_S^r = \{i \in I \mid r(i) \in \text{mods}(S)\}$. Therefore, for any filter $F \in \mathcal{P}^{\text{fin}}[I]$ such that $\text{wf}_{\varsigma}(r, F)$ and $\text{wf}_{\varsigma'}(r, F)$ hold (so all dereferences are defined), we have

$$F \cap \text{fp}_S^r = \emptyset \implies (\forall i \in F. \text{drf}_{\varsigma}(r(i)) = \text{drf}_{\varsigma'}(r(i))).$$

We have as a consequence, for any *reduce* mapping component $m : V \rightarrow S$, that $\forall i \in F. m(\text{drf}_{\varsigma}(r(i))) = m(\text{drf}_{\varsigma'}(r(i)))$, which implies $\text{reduce}_{\varsigma}^{\oplus, m, r}(F) = \text{reduce}_{\varsigma'}^{\oplus, m, r}(F)$ by **(singleton)** and **(extensionality)**.

By the above argument, we can justify the right-hand-side of our **(extensionality)** Axiom by directly proving $F \cap \text{fp}_S^r$. The advantage of this ideas is that framing of reduce terms can potentially be deduced by a query purely in terms of index sets, without the need to generate terms that ultimately check the values of heap locations (as on the left-hand-side of **(extensionality)**). Combined with the fact that we can control application of this rule to only compare successor pairs of labelled states, this gives a potentially far cheaper way to deduce framing when the reason is (as is guaranteed by our decomposition strategy) that one filter has an empty index set for a given statement.

We formalise this indexical framing idea as an automatable *framing rule*, generated for every relevant statement alongside its pre-/post-states $\varsigma \xrightarrow{S} \varsigma'$, as

follows:

$$\text{Fr } \zeta \overset{S}{\rightsquigarrow} \zeta' \frac{\text{know}(\overset{\oplus}{\zeta}[F] \text{ and } \overset{\oplus}{\zeta'}[F]) \quad \text{wf}_{\zeta}(r, F) \quad \text{wf}_{\zeta'}(r, F) \quad F \cap \text{fp}_S^r = \emptyset}{\overset{\oplus}{\zeta}[F] = \overset{\oplus}{\zeta'}[F]}$$

As anticipated, since by definition $(F \setminus \text{fp}_S^r) \cap \text{fp}_S^r = \emptyset$ our update rules yield decomposed terms $\overset{\oplus}{\zeta}[F \setminus \text{fp}_S^r]$ and $\overset{\oplus}{\zeta'}[F \setminus \text{fp}_S^r]$, from which $\overset{\oplus}{\zeta}[F \setminus \text{fp}_S^r] = \overset{\oplus}{\zeta'}[F \setminus \text{fp}_S^r]$ immediately follows by the framing rule for S_j . However, this framing rule applies in *all* cases for which filter arguments can be proven disjoint from an index footprint (e.g. an update that conditionally affects a location either in the sub-data structure ranged over by F , or a location of the data structure disjoint from F , depending on program context; the framing rule applies in the latter case).

As an example, we can apply this framing rule in our swap example, with footprints $\{j\}$ and $\{k\}$, producing instances of this rule:

$$\text{Fr } \zeta_0 \rightsquigarrow \zeta_1 \frac{\text{know}(\overset{\oplus}{\zeta_0}[F] \text{ and } \overset{\oplus}{\zeta_1}[F]) \quad \text{wf}_{\zeta_0}(r, F) \quad \text{wf}_{\zeta_1}(r, F) \quad F \cap \{j\} = \emptyset}{\overset{\oplus}{\zeta_0}[F] = \overset{\oplus}{\zeta_1}[F]}$$

$$\text{Fr } \zeta_1 \rightsquigarrow \zeta_2 \frac{\text{know}(\overset{\oplus}{\zeta_1}[F] \text{ and } \overset{\oplus}{\zeta_2}[F]) \quad \text{wf}_{\zeta_1}(r, F) \quad \text{wf}_{\zeta_2}(r, F) \quad F \cap \{k\} = \emptyset}{\overset{\oplus}{\zeta_1}[F] = \overset{\oplus}{\zeta_2}[F]}$$

These rules generate all of the framing arguments needed to verify the *reduce* specification. The complete proof relies on several auxiliary concepts, the most important being two applications of Axiom (**singleton**) to the pointwise heap updates: $\overset{\uplus}{\zeta_0}[\{k\}] = \overset{\uplus}{\zeta_1}[\{j\}]$ and $\overset{\uplus}{\zeta_2}[\{k\}] = \{\text{tmp}\} = \overset{\uplus}{\zeta_0}[\{j\}]$. The introduction of the term $\overset{\uplus}{\zeta_0}[\{k\}]$ straightforwardly results from the *heap read* of $\mathbf{A}[k]$ as a right-hand value in line 7. The argument also makes use of the *right quasi-commutativity* of set difference, $(R \setminus S) \setminus T = (R \setminus T) \setminus S$, which is a useful identity given the liberal use of set difference by our update rule.¹¹

5.2 Extensionality beyond framing

Although indexical framing is a powerful tool for efficient reasoning, it does not subsume extensionality arguments in general: there can be applications of Axiom (**extensionality**) that follow despite a filter having overlap with an index footprint. These cases can arise for two reasons: non-empty footprints that (perhaps conditionally) do not actually change values in memory, and equality of updated heap values under a *reduce* mapping component.

The former reason can arise from calls to a method whose post-conditions logically guarantee value preservation, or from general statements whose preservation of values is conditional on some property that can be proved to be true. Alternatively, the latter reason can arise e.g. for a *reduce* specification defining a count of

¹¹ We aim for efficiency and minimal reliance on set theory axioms; thus, we prefer algebraic arguments, such as this right quasi-commutativity, to extensional ones.

nonzero values in a data structure, whose codomain is $(\mathbf{Int}, +, 0)$ and mapping is $\mathbf{nz} :: \mathbf{Int} \rightarrow \mathbf{Int}$ such that $\mathbf{nz}(x)$ is 1 when $x \neq 0$ and 0 when $x = 0$. A method call that multiplies an arbitrary collection of values in a data structure by 2 preserves this count, although the footprints of such a statement would be non-empty. Thus, an application of **(extensionality)** shows the count is preserved, which indexical framing cannot. Here, extensionality has a theory-specific proof: for the \mathbf{Int} type, $x = 0 \iff 2x = 0$, therefore $\mathbf{nz}(\mathit{drf}_{\zeta_j}(r(i))) = \mathbf{nz}(2\mathit{drf}_{\zeta_j}(r(i))) = \mathbf{nz}(\mathit{drf}_{\zeta_j}(r(i)))$ for valid $i \in \mathbf{fp}$ or $i \notin \mathbf{fp}$.

For these reasons, we retain extensionality, but give *priority* to indexical framing as far as possible. In the context of a proof assistant, this preference is easy to encode in a tactic; this is harder to guarantee for SMT-based verification, but we can ensure indexical framing will always be instantiated. In practice, this approach performs very well, as shown in our evaluation.

6 Implementation and Evaluation

We developed a prototype implementation (which we will open-source and provide as an artifact) of our novel *reduce* operator and accompanying automated proof techniques as a plugin for the Viper verification framework [24]. Our plugin extends the base language with syntax for the *reduce* operator itself, as well as separate declarations of the functions passed as its parameters (e.g. receiver functions). We use the standard Viper verifiers unchanged: our plugin first encodes our extended syntax into vanilla Viper, applying the automation and framing techniques presented in Sec. 4 and Sec. 5, respectively.

6.1 Implementation Overview

While our conceptual contributions are designed to be methodology-independent as far as possible, for our implementation we have to handle Viper specifics. Viper functions are not first-class and cannot be passed as arguments (to our *reduce* operator). We solve this issue by *defunctionalisation* [27]: using elements of an uninterpreted sort to represent each function, alongside axioms connecting applications of the function to its real definition (*cf.* App. B.1 for more details).

Viper memory locations are (all) field locations; our plugin identifies statements that access fields and/or change those to which permission is held and generates inlined versions of our Sec. 3.2 axioms according to the strategies of Sec. 4.1 and in terms of the abstract representations from Sec. 5 (*cf.* App. B.3). Viper employs separation-logic-style reasoning in terms of *permissions* [30], providing automatic framing e.g. for single heap updates. Viper also supports the combination of *iterated separating conjunction* [23] (defining pointwise sets of permissions), as well as *permission introspection*, providing expressions that evaluate to permission amounts currently held. Our encoding uses these features to symbolically define the sets of locations changed, freed, allocated etc. as needed for our decomposition and framing strategies (*cf.* App. B.2).

Our plugin supports a useful generalisation of our *reduce* operator: we allow its parameters to be *closures* in the sense that they can capture values (but not heap locations) from the surrounding context. For example, this allows us to express the sum over a column of a 2D-matrix using $\lambda i. A[i][j]$ as a receiver function, in which j is captured (more details in App. B.4).

6.2 Evaluation

As a means of evaluating both our implementation and our automated proof technique, we selected a set of algorithmic examples from [20], [21], [33] and the VerifyThis competition archive [35] that have properties that are well-suited to *reduce* specifications. A summary of each example follows, along with notable aspects of our *reduce* operator required to specify and verify the example. All encoded examples will be provided in our accompanying artifact.

ArraySum This method sums array A in a loop (accumulator c , iterator i), with two alternative invariants we term I1: `invariant c == sum([0..i])`, from [33], and I2: `invariant c + sum([i..A.length]) == sum([0..A.length])`, from [20]. Since A is not modified, the decompositions arise from heap reads

Swap Our running example from Sec. 2. Multiset preservation alone is proved using *fuel* with value one outlined in Sec. 4, while the addition of maximum preservation requires fuel with value two. It is used in the next two examples.

Bubble Sort This implementation of bubble sort uses nested while loops; the inner loop swaps adjacent pairs with a value inversion, and the outer repeats the inner loop until swaps occur. We proved multiset and max preservation.

Quick Sort This example uses the Lomuto partitioning scheme from [4], with recursive calls over subarrays with ranges $[lo..p-1]$ and $[p+1..hi]$ for pivot p . Our technique automatically decomposes and frames the fragments from each call.

Shear Sort This matrix sorting algorithm appeared in the 2021 VerifyThis competition. We use integer pairs as an index type, and define row- and column-specific filters, with automated decompositions by the row or columns.

Coincidence Count This algorithm, described in [10], counts the matching values across two sorted arrays A and B of varying length. We specify this with integer pairs (one for each array), summing over the indicator map $\mathbb{1}_{a=b}$. Neither A nor B are modified, all decompositions are based on conditional reads for each array.

Graph Coloring (BFS) This algorithm appeared as a verification exercise in a graduate course; it traverses a graph ($\#$ children ≤ 4) in BFS order, marking each node and adding its children to a *to-do list*. We verify a *termination measure*: either the count of unmarked nodes decreases, or the length of the to-do list does.

Constant-space Tree Coloring (DFS) This binary tree marking algorithm appeared in the 2016 VerifyThis competition. We verify a component of a termination proof: each node is visited exactly three times, so we define a counter c in each node and our *reduce* sums over the mapping $3 - c$ and prove this measure decreases.

| Example | Task | Struct | Operator | t_s | t_c | LoC/S | Fuel | Asserts |
|---------------|-----------|--------|------------------------------|-------|-------|-------|------|---------------|
| ArraySum | I1 | Array | + (Int) | 0.5 | 0.7 | 6/9 | 1 | 0 |
| | I2 | | | 0.5 | 0.7 | 6/9 | 1 | 0 |
| Swap | Verif | Array | \uplus (mset) | 1.1 | 1.2 | 4/12 | 2 | 0 |
| | | | max | | | | | |
| BubbleSort | mset only | Array | \uplus (mset) | 0.8 | 1.0 | 14/15 | 1 | 0/1 (c/s) |
| | (& max) | | max | 1.3 | 1.5 | 14/25 | 2 | 0/1 (c/s) |
| QuickSort | mset only | Array | \uplus (mset) | 1.3 | 1.5 | 20/20 | 1 | 0/1 (c/s) |
| | (& max) | | max | 3.3 | 2.5 | 20/29 | 2 | 0/1 (c/s) |
| Shearsort | mset only | Matrix | \uplus (mset) | 1.1 | 1.6 | 14/37 | 1 | 0/2 (c/s) |
| | (& max) | | max | 16.7 | 7.3 | 14/38 | 2 | 0/2 (c/s) |
| CoincCnt | Verif | Arrays | + (Int) | 0.8 | 0.9 | 13/26 | 1 | 0 |
| | | (pair) | $\mathbb{1}_{a=b}$ | | | | | |
| GraphBFS | Verif | Graph | + (Int) | 11.3 | 17.6 | 19/25 | 1 | 0 |
| (termination) | | | $\mathbb{1}_{\text{marked}}$ | | | | | |
| ConstDFS | Verif | Graph | + (Int) | 2.8 | 2.0 | 18/11 | 1 | 0 |
| (termination) | | | cnt $\in [0..3]$ | | | | | |

Table 1: Results for our Viper implementation of *reduce*. t_c and t_s are verification times (in seconds) for Viper’s Carbon (Boogie) and Silicon (symbolic execution) backends (mean over 3 trials; all had relative std. dev. < 3%). LoC/S is lines of code and lines of user specification. Fuel is the bound on decomposition depth for update rules. Asserts is number of of manual assertions needed.

6.3 Results

We tested each example using the Carbon (verification conditions) and Silicon (symbolic execution) backends of Viper, and report the verification runtime for each. Experiments were run on a MacBook Pro with Apple M2 processor and 8GB RAM. Our results are summarized in *Table 1*. Several swap examples required manual assertions with Silicon to prove that adequate permissions were held for method calls, however, no manual assertions about *reduce* decompositions were required. Because our specifications described *reduce* properties only, rather than full functional correctness, the number of lines of specification demonstrate the conciseness of our *reduce* properties. Increasing the fuel value 2 was necessary to prove sorting examples with maximum preservation, which markedly increased

verification times. However, interestingly we did not observe that the amount of fuel required did not increase beyond 2 for longer programs, which we attribute to the “selection” property of the maximum operator (i.e., $\max(a, b)$ has value either a or b ; this holds for truth values with logical operators as well) that allows case-splitting on the possible values of each term in a decomposition equation.

7 Related Work

To our knowledge, no deductive verifier before our work provides general support for reduce expressions alongside reliable automation of its proofs. Early work by Leino and Monahan [21] defines a related integer-indexed *comprehension* feature with SMT axioms; both incompletenesses and termination issues could result in practice [21]. Müller et al. [23] propose a similar injectivity restriction to our work in the context of automation for the iterated star from separation logic; their work does not address aggregate properties of the corresponding data structures. Ter-Gabrielyan [33] proposed a more-general notion of comprehensions (allowing non-commutative operators), but did not address automation. Our work takes inspiration from Tokaev [34], who proposed an encoding that can, however, be both incomplete and prohibitively slow in general; the latter likely for the reasons we address in Sec. 5.

Many verifiers and proof assistants support folds, but enforce an inductive order on their definition (and so reasoning). One exception is Nuprl [1], which provides a rich library of theorems concerning iterated binary operators. These theorems subsume our core axioms, but must be manually applied.

Several recent works address reasoning alternatives to inductive definitions. Krishna et al. [15] present *flows* as a means of handling structures with sharing and cycles; further work [16] also identified *cancellativity* as important for reasoning. Farka et al. [11] present a generalization of partial monoid homomorphisms (PCM morphisms) and describe their utility in defining and proving program properties in concurrent separation logic. This work is developed by Grandrury et al. [13] to express and concisely prove local properties of graphs using higher-order PCM morphisms. Their methods are mechanized in Roq, however they do not define automated verification methods relative to program specifications.

Recent work by Xu et al. [36] develops an automated theorem prover to derive separation logic rules through PCM morphisms and functors. Compared to our work, their tool covers a wider breadth of separation logic, comprising 300 rules to transform predicates and handle inductive definitions. Our work focuses on integration with existing verification tools and program logics, and defines a relatively lightweight set of proof rules and automation methods.

8 Conclusions and Future Work

We have introduced the *reduce* operator as a flexible means of specification and automatic verification for a wide variety of data comprehensions and non-inductively

data structures. Our automated decomposition and framing techniques are designed to be compatible with diverse verifiers, and allow for efficient verification and concise program specifications that follow mathematical intuitions.

We plan to investigate our *reduce* operator and automated decomposition and framing rules within the framework of PCM morphisms. As we support a wider class of algebraic structures (non-cancellative monoids, and semigroups), it remains to be seen which proof rules of separation logic are preserved through our *reduce* homomorphisms. A related research direction is the generalization of our approach to non-commutative operators, which would invoke *array theories*, studied for instance by Raya and Kunčák [26]. We are also interested in exploring the possibility of encoding and possibly extending refinement/liquid type systems [29], as cast within the framework of our novel reduce specifications.

References

1. Stuart F. Allen. Iterated Binary Operations. <https://nuprl-web.cs.cornell.edu/KB/show.php?ID=13>, 2004. Accessed Jan 28, 2026.
2. Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT Solver. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, pages 20–35, Cham, 2014. Springer International Publishing.
3. Kshitij Bansal, Clark Barrett, Andrew Reynolds, and Cesare Tinelli. Reasoning with Finite Sets and Cardinality Constraints in SMT. *Logical Methods in Computer Science*, Volume 14, Issue 4, November 2018. Publisher: Episciences.org.
4. Jon Louis Bentley. *Programming pearls*. ACM Press, New York, NY, 2 edition, 1999.
5. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let’s verify this with Why3. *Int. J. Softw. Tools Technol. Transf.*, 17(6):709–727, November 2015.
6. Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. Decision Problems and Some Solutions. In Domenico Cantone, Eugenio Omodeo, and Alberto Policriti, editors, *Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets*, pages 148–195. Springer, New York, NY, 2001.
7. A. H. Clifford and G. B. Preston. *Algebraic Theory of Semigroups, Volume 1*. Number v.7 in Mathematical Surveys and Monographs. American Mathematical Society, Providence, 1961.
8. Maximiliano Cristiá and Gianfranco Rossi. A Decision Procedure for Sets, Binary Relations and Partial Functions. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 179–198, Cham, 2016. Springer International Publishing.
9. Maximiliano Cristiá and Gianfranco Rossi. A Decision Procedure for Restricted Intensional Sets. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 185–201, Cham, 2017. Springer International Publishing.
10. Edsger W. Dijkstra and Wim H. J. Feijen. *A method of programming*. Addison-Wesley, Boston, MA, repr edition, 1988.
11. František Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. On algebraic abstractions for concurrent separation logics. *On Algebraic Abstractions for Concurrent Separation Logics (artefact)*, 5(POPL):5:1–5:32, January 2021.
12. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, March 1997.
13. Marcos Grandury, Aleksandar Nanevski, and Alexander Gyzlov. Verifying Graph Algorithms in Separation Logic: A Case for an Algebraic Approach. *Artifact for Verifying Graph Algorithms in Separation Logic: A Case for an Algebraic Approach.*, 9(ICFP):241:160–241:189, August 2025.
14. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
15. Siddharth Krishna, Dennis Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31, December 2017.
16. Siddharth Krishna, Alexander J. Summers, and Thomas Wies. Local Reasoning for Global Graph Properties. In Peter Müller, editor, *Programming Languages and Systems*, pages 308–335, Cham, 2020. Springer International Publishing.

17. Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, April 2006.
18. K. Rustan M. Leino. This is Boogie 2. Unpublished, June 2008.
19. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'10*, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
20. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *In FTfJP '07: Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs*, January 2007.
21. K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, New York, NY, USA, March 2009. ACM.
22. Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.
23. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *Computer Aided Verification*, Lecture notes in computer science, pages 405–425. Springer International Publishing, Cham, 2016.
24. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, page 41–62, Berlin, Heidelberg, 2016. Springer-Verlag.
25. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
26. Rodrigo Raya and Viktor Kunčak. Succinct ordering and aggregation constraints in algebraic array theories. *Journal of Logical and Algebraic Methods in Programming*, 140:100978, August 2024.
27. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference—Volume 2*, ACM '72, pages 717–740. ACM, 1972.
28. Rocq Team. The Rocq proof assistant reference manual. <https://rocq-prover.org/refman>, 2023.
29. Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, June 2008. Association for Computing Machinery.
30. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
31. Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with Cardinality Constraints in Satisfiability Modulo Theories. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 403–418, Berlin, Heidelberg, 2011. Springer.
32. Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub,

- Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Principles of Programming Languages, POPL*, 2016.
33. Arshavir Ter-Gabrielyan. *Compositional verification of rich program properties in separation logic*. PhD thesis, ETH Zurich, 2021.
 34. Peeranat Tokaeo. Automatic verification of heap-dependent folds in viper. Master's thesis, University of British Columbia, 2024.
 35. VerifyThis. Competition archive. <https://verifythis.github.io/onsite/archive/>.
 36. Qiyuan Xu, David Sanan, Zhe Hou, Xiaokun Luan, Conrad Watt, and Yang Liu. Generically Automating Separation Logic by Functors, Homomorphisms, and Modules. *The Artifact of "Generically Automating Separation Logic by Functors, Homomorphisms, and Modules"*, 9(POPL):67:1992–67:2024, January 2025.
 37. Hongseok Yang and Peter O'Hearn. A Semantic Basis for Local Reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 402–416, Berlin, Heidelberg, 2002. Springer.

A Supporting Definitions and Proofs

A.1 The Reduce Operator

We provide a proof sketch for the existence of the *reduce* operator. The fundamental idea is that the choice of mapping and operator uniquely define a homomorphism from multisets of values to the *reduce* target structure. This composes with a lifted receiver and dereference function to give a *partial homomorphism* from the monoid of index sets to the target semigroup (or monoid). This is a partial homomorphism in two senses: $(\mathcal{P}^{\text{fin}}[I], \sqcup, \emptyset)$ is a *partial commutative monoid*, and $\text{reduce}_\sigma^{\oplus, m, r}(F)$ is a *partial function* from $\mathcal{P}^{\text{fin}}[I]$ to S .

Theorem 1. *Let I be a type, (S, \oplus) be a semigroup, and $f : V \rightarrow S$ any function. There is a unique semigroup homomorphism $\phi_f^\oplus : \mathcal{M}^{\text{fin}}[V] \setminus \{\emptyset\} \rightarrow S$ that mimics f on singleton multisets, i.e. satisfying the following two conditions:*

- for every i in I , we have $\phi_m^\oplus(\{\{i\}\}) = f(i)$;
- for all $m_1, m_2 \in \mathcal{M}^{\text{fin}}[V]$, we have $\phi_f^\oplus(m_1 \uplus m_2) = \phi_f^\oplus(I_1) \oplus \phi_f^\oplus(I_2)$.

Further, for any monoid (S, \oplus, e_S) , there exists a unique monoid homomorphism $\phi_f^\oplus : \mathcal{P}^{\text{fin}}[I] \rightarrow S$ satisfying $\phi_f^\oplus(\emptyset) = e_S$ as well as the above two properties.

Proof (Sketch). Note that if a commutative semigroup has a left or right identity element, it is necessarily a two-sided identity and thus unique [7]; hence this ϕ_f^\oplus is well-defined across the semigroup and monoid cases.

This ϕ is well-defined, as there exists a unique decomposition of m into singleton multisets, i.e. $m = \{\{v_1\}\} \uplus \{\{v_2\}\} \uplus \dots \uplus \{\{v_k\}\}$ where $|m| = k$. Moreover, it is a semigroup homomorphism by definition.

Suppose there exists some ϕ' satisfying the same conditions, each of these defines a multiset expression

$$\phi_f^\oplus(m) = f(v_1) \oplus \dots \oplus f(v_k) \quad \phi'(m) = f(v'_1) \oplus \dots \oplus f(v'_k).$$

Then, by the commutativity of S , there exists a permutation of the respective expressions so that $f(v'_1) \oplus \dots \oplus f(v'_k) = f(v_1) \oplus \dots \oplus f(v_k)$.

Given a $\sigma \in \Sigma$, we can lift the receiver r and dereference $\text{drf}(\sigma)$ as follows

$$\mathcal{M}^{\mathcal{P}}[\text{drf}(\sigma, \cdot)](L) = \bigsqcup_{\ell \in L} \{\{\text{drf}(\sigma, \ell)\}\}, \quad \mathcal{P}^{\text{fin}}[r](F) = \bigsqcup_{i \in F} \{r(i)\}.$$

The *reduce* operator is then summarized as the composition of these set- and multiset-valued operators via the type-level diagram

$$\mathcal{P}^{\text{fin}}[I] \xrightarrow{\mathcal{P}^{\text{fin}}[r]} \mathcal{P}^{\text{fin}}[\text{loc}(V)] \xrightarrow{\mathcal{M}^{\mathcal{P}}[\text{drf}(\sigma, \cdot)]} \mathcal{M}^{\text{fin}}[V] \xrightarrow{\phi_m^\oplus} S.$$

A.2 Decompositions

We define more-general decomposition rules to consider *frees* and *allocates* notions: aiming to appropriately decompose *reduce* terms over statements which add or remove locations from the domain of the program state. The general idea is as follows: for a set of allocated memory locations, say $\text{allc}(S)$, we will generate asymmetric information in ζ' and any predecessor state ζ . For a term $\overset{\oplus}{\zeta'}[F]$ we yield the full decomposition of F with $\text{allc}(S)$ in ζ' , but only the term $\overset{\oplus}{\zeta}[F \setminus \text{allc}(S)]$ in the predecessor state. The rule for freed locations is similar, with the outcomes in ζ' and ζ swapped (yield full decompositions in predecessor state ζ , and only single terms in ζ'). These rules are concordant with our update rule for memory footprints, so long as we treat modified, allocated and freed locations as disjoint.

$$\text{Alloc } \zeta' \overset{\mathcal{S}}{\rightsquigarrow} \zeta \frac{\text{know}(\overset{\oplus}{\zeta'}[F]) \quad \text{wf}_{\zeta'}(r, F) \quad \text{wf}_{\zeta}(r, F \setminus \text{allc}(S)) \quad \text{allc}(S) \cap F \neq \emptyset}{\left(\overset{\oplus}{\zeta'}[F] = \overset{\oplus}{\zeta'}[F \setminus \text{allc}(S)] \oplus \overset{\oplus}{\zeta'}[F \cap \text{allc}(S)]\right) \wedge \text{know}(\overset{\oplus}{\zeta}[F \setminus \text{allc}(S)])}$$

$$\text{Free } \zeta \overset{\mathcal{S}}{\rightsquigarrow} \zeta' \frac{\text{know}(\overset{\oplus}{\zeta}[F]) \quad \text{wf}_{\zeta}(r, F) \quad \text{wf}_{\zeta'}(r, F \setminus \text{free}(S)) \quad \text{free}(S) \cap F \neq \emptyset}{\left(\overset{\oplus}{\zeta}[F] = \overset{\oplus}{\zeta}[F \setminus \text{free}(S)] \oplus \overset{\oplus}{\zeta}[F \cap \text{free}(S)]\right) \wedge \text{know}(\overset{\oplus}{\zeta'}[F \setminus \text{free}(S)])}$$

B Further Implementation and Evaluation Details

We provide here a more-detailed description of some of the key aspects of our plugin implementation. We omit details of standard concerns such as parsing and type-checking, and focus on aspects that either show up in the surface syntax supported or are non-obvious aspects of our encoding to vanilla Viper.

B.1 Defunctionalisation for higher-order operators.

Our extension to Viper allows for *reduce* components to be arbitrarily user-specified. As Viper does not allow function types, we use *defunctionalisation* to encode the *reduce* operator and its components as polymorphic *domains*, with domain functions and axioms that apply function instances to arguments. We illustrate this in Listing 1.3 with a `Reduce [I, V, S]` domain and user-defined components: receiver `rg` in line 20, mapping `countNot` in line 29 and operator `add` in line 36. We model *reduce* as *pure function* with the symbolic state label argument `rheap`.

```

1 domain Reduce [I, V, S] {
2   function red(r:Rcv [I], m:Map [V, S], op:Op [S]): Reduce [I, V, S]
3   function redApply(rheap:Int, c:Reduce [I, V, S], fs:Set [I]): S
4   function getop(c: Reduce [I, V, S]) : Op [S]
5   // more auxiliary functions...
6

```

```

7 // example axiom (trigDecomp term yielded by inline axioms)
8 axiom _decomp {
9   forall rh:Int, rd:Reduce[I,V,S], fs:Set[I], fp:Set[I] ::
10   { trigDecomp( redApply(rh, rd, fs), fp ) }
11   (fp subset fs) && !(fp == fs) ==>
12   redApply(rh, rd, fs) ==
13   opApply(getop(rd), redApply(rh, rd, fs \ fp),
14           redApply(rh, rd, fp))
15 }
16 // more axioms...
17 }
18
19 // syntax: receiver rg(G: Graph) (fun n: Node :: loc(G,n))
20 domain _rg_Receiver_Domain {
21   function rg(G: Graph): Receiver[Node]
22   axiom { forall n: Node :: recApply(rg(G), n) == loc(G,n) }
23 }
24
25 function negBooltoInt(b: Bool): Int
26   ensures (b ==> result == 0) && (!b ==> result == 1)
27
28 // syntax: mapping countNot() (fun b: Bool :: negBtoI(b))
29 domain _countNot_Mapping_Domain {
30   function countNot(): Mapping[Bool, Int]
31   axiom { forall b: Bool ::
32           mapApply(countNot(),b) == negBooltoInt(b) }
33 }
34
35 // syntax: operator add() (0, fun a: Int, b: Int :: a + b)
36 domain _add_Operator_Domain {
37   function add(): Op[Int]
38   axiom { forall a:Int, b:Int :: opApply(add(),a,b) == a+b }
39   axiom { opGetIden(add()) == 0 }
40 }
41
42 field marked: Bool
43
44 // macro for reduce specifications
45 define unmarkedCount(G,F)
46   (hreduce[add()](countNot(idrec(G).marked) | F): Int)
47
48 // e.g. spec for graph marking termination measure:
49 //   assert unmarkedCount(G,F) == 0 ||
50 //           unmarkedCount(G,F) < old(unmarkedCount(G,F))
51 }

```

Listing 1.3: reduce and operator encoding in Viper

B.2 Inferring memory footprints.

Viper’s heap model consists of a `Ref` type representing abstract memory locations, and *field access expression* of the form $e_{ref}.f$, where e_{ref} is a `Ref`-type expression and f is a uniquely named field identifier. This refines the type-driven index footprints outlined in Sec. 4.1, as index footprints are specific to a receiver *and* field identifier.

Viper supports the *iterated separating conjunction*, which can express quantified formulas asserting permissions to an unbounded set of heap locations [23]. We use this to infer memory footprints for program statements, defining new `Set[Ref]`-typed variables in terms of changes to the permissions mask. A (stylized) example is given in Listing 1.4, where the memory footprint of the `exhale` statement on lines 4-6 is inferred in lines 8-11 as `lostPerms`. The receiver domain axioms define the `Set[I]`-typed index footprints from any relevant inverse receivers in the program.

```

1 var lostPerms: Set[Ref]
2 label pre_exh
3 // exhales permission to all locations of array A
4 exhale (
5   forall k: Int :: (k in [0..len(a)]) ==> acc(A[k].f, write)
6 )
7 // infers that lostPerms is the set of lost permissions
8 assume (
9   forall r: Ref :: (r in lostPerms) <==>
10    (perm(r.f) == none && old[pre_exh](perm(r.f)) > none)
11 )

```

Listing 1.4: inferring a memory footprint in Viper

Our method for inferring footprints is polarized by Viper’s `inhale` and `exhale` statements, which can assert permissions were gained or lost, respectively. This allows for streamlined decomposition and framing axioms, detailed in App. A.2. We transform statements with simultaneous positive and negative permissions changes (e.g. method calls) to `exhale/inhale` pairs.

B.3 Inlining of Axioms

Our plugin traverses the program AST, checking field accesses in each statement against a list of all `reduce` specifications in the program. The plugin infers index footprints and annotates AST nodes with the map of the symbolic state number corresponding to each field identifier and receiver. A second pass over the AST adds *inline axioms* as Viper `assume` statements with quantified formulas as instances of the decomposition and framing axioms. An example inline axiom (again, in stylised Viper code) is given in Listing 1.5.

```

1  assume (forall rd: Reduce[Int,Int,MSet[Int]], fs: Set[Int] ::
2    // Trigger for reduce terms in symbolic state 0 or 1
3    { redApply(0, rd, fs) } { redApply(1, rd, fs) }
4
5    // Receiver check for filter fs (in current state)
6    wfRcv(getRcv(rd), fs) ==>
7
8    // Decomp with index footprint
9    let fp == recInv(getRcv(rd),A[j]) in
10   trigDecomp(redApply(0,rd,fs), fp) &&
11   trigDecomp(redApply(1,rd,fs), fp) &&
12
13   // Singleton eval A[j] = m(A[j].val) in 0; m(A[k].val) in 1
14   trigEval(redApply(0,rd,fp),mapApply(getMap(rd),A[j].val)) &&
15   trigEval(redApply(1,rd,fp),mapApply(getMap(rd),A[k].val))
16 )
17
18 assume (forall rd: Reduce[Int,Int,MSet[Int]], fs: Set[Int] ::
19   // Trigger for reduce terms in states 0 and 1 with same fs
20   { redApply(0, rd, fs) , redApply(1, rd, fs) }
21
22   wfRcv(getRcv(rd), fs) ==>
23   let fp == recInv(getRcv(rd),A[j]) in
24
25   // Indexical framing
26   (fs intersect fp == emptySet()) ==>
27   redApply(0, rd, fs) == redApply(1, rd, fs) &&
28 )
29
30 A[j].val := A[k].val

```

Listing 1.5: inline axiom for reassignment statement

Since Viper’s verification technique is modular per program method, the associativity and commutativity (and identity, if applicable) of each user-defined operator is verified with a separate method call. Additionally, a heap-dependent predicate `receiverInjCheck` conditions the axiom for each concrete *reduce* term that invokes the inline axiom; in case this check fails, the *reduce* application is undefined (in Viper, uninterpreted functions express partiality in this manner).

B.4 Extension: Context Capture

Consider the method in Listing 1.6, which sums a column of `Int`-valued $m \times n$ matrix `A`. Line 8 is a loop invariant that the accumulator variable `c` contains the partial sum of column `j`. Expressing this property requires two instances of *context capture*: the receiver body $\lambda i. A[i][j]$ contains a free variable `j` (the column number), and the filter expression `[0..i]` has free variable `i`. The state-dependent filter is simple to encode as *reduce*(`[0..old(i)]`) and *reduce*(`[0..i]`), where `old(i) + 1 == i` is provable from line 10. However, care is needed for the receiver, in order to verify that injectivity holds for arbitrary values of `j`.

```

1  method sumColumn(A: Matrix, j: Int) returns c: Int
2  requires nrows(A) == m && ncols(A) == n && 0 <= j < n
3  ensures c == reduce(+, id, λi. A[i][j]) ([0..m])
4  {
5      var i, c Int := 0
6      while (i < m)
7          invariant 0 ≤ i ≤ m
8          invariant c == reduce(+, id, λi. A[i][j]) ([0..i])
9          { c := c + A[i][j]
10         i := i + 1 }
11 }

```

Listing 1.6: context capture

We expand our definition of receivers for context-capture by changing the type to $r :: \vec{\tau} \rightarrow I \rightarrow \text{loc}(T)$, where $\vec{\tau}$ represents an arbitrary tuple of captured parameters. We require that a receiver satisfies a *local injectivity* property:

$$\forall \vec{x} :: \vec{\tau}. \forall i, j :: I. r(\vec{x})(i) = r(\vec{x})(j) \implies i = j. \quad (\text{cc-inj})$$

This (cc-inj) is satisfied by the receiver $\lambda i. A[i][j]$, which in fact satisfies a *pointwise injectivity* property for all pairs of i, j . However, for distinct \vec{x}_1, \vec{x}_2 , we could have $i_1 \neq i_2$ and $r(\vec{x}_1)(i_1) = r(\vec{x}_2)(i_2)$ and satisfy (cc-inj), e.g. with a receiver $\lambda i. A[i+j]$ that computes an offset location in array A .

We can extend mappings $m :: \vec{\tau} \rightarrow T \rightarrow S$ and operators $m :: \vec{\tau} \rightarrow S \times S \rightarrow S$ with no additional restrictions. Context-capture for these components defines a *reduce* operator for each distinct set of contextual arguments, which may be desirable.