Mechanised Hypersafety Proofs about Structured Data

VLADIMIR GLADSHTEIN, National University of Singapore, Singapore
QIYUAN ZHAO, National University of Singapore, Singapore
WILLOW AHRENS, Massachusetts Institute of Technology, USA
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA
ILYA SERGEY, National University of Singapore, Singapore

Arrays are a fundamental abstraction to represent collections of data. It is often possible to exploit *structural properties* of the data stored in an array (*e.g.*, repetition or sparsity) to develop a specialised representation optimised for space efficiency. Formally reasoning about correctness of manipulations with such structured data is challenging, as they are often composed of multiple loops with non-trivial invariants.

In this work, we observe that specifications for structured data manipulations can be phrased as *hypersafety* properties, *i.e.*, predicates that relate traces of k programs. To turn this observation into an effective verification methodology, we developed the Logic for Graceful Tensor Manipulation (LGTM), a new Hoare-style relational separation logic for specifying and verifying computations over structured data. The key enabling idea of LGTM is that of *parametrised* hypersafety specifications that allow the number k of the program components to depend on the *program variables*. We implemented LGTM as a foundational embedding into Coq, mechanising its rules, meta-theory, and the proof of soundness. Furthermore, we developed a library of domain-specific tactics that automate computer-aided hypersafety reasoning, resulting in pleasantly short proof scripts that enjoy a high degree of reuse. We argue for the effectiveness of relational reasoning about structured data in LGTM by specifying and mechanically proving correctness of 13 case studies including computations on compressed arrays and efficient operations over multiple kinds of sparse tensors.

CCS Concepts: • **Theory of computation** → **Logic and verification**; *Programming logic*; • **Software and its engineering** → *Formal software verification.*

Additional Key Words and Phrases: sparse data structures, mechanised proofs, relational logic

ACM Reference Format:
Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI, Article 173 (June 2024), 24 pages. https://doi.org/10.1145/3656403

1 INTRODUCTION

Arrays are one of the most basic yet most powerful abstractions in computer science, capable of representing virtually any kind of data. The most common way to encode data is via a *dense* array that stores data in memory *contiguously*, providing a simple interface to access it via iteration. Representing data via dense arrays is often not optimal in terms of space efficiency. For instance, a *sparse tensor* (*i.e.*, an n-dimensional matrix with most of its elements being zeros) can be encoded compactly by storing only its non-zero elements. Without this space optimisation, manipulations

Authors' addresses: Vladimir Gladshtein, National University of Singapore, Singapore, vgladsht@comp.nus.edu.sg; Qiyuan Zhao, National University of Singapore, Singapore, qiyuanz@comp.nus.edu.sg; Willow Ahrens, Massachusetts Institute of Technology, USA, willow@csail.mit.edu; Saman Amarasinghe, Massachusetts Institute of Technology, USA, saman@csail.mit.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2024/6-ART173
https://doi.org/10.1145/3656403

with tensors that contain real-world datasets becomes impossible due to the immense amount of storage that would be required by their dense representations.

Sparsity of a dense array is just one of the many structural properties of the array-stored data; others include *symmetry* (*e.g.*, an array-encoded matrix is equal to its transpose), *repetition* (data contains segments of repeating elements), or length-*irregularity* (*a.k.a.* ragged arrays). Realising such data properties allows one to choose a specialised *format* for representing *structured data*, leading to reduced size of the required storage and improved performance by avoiding redundant computations (*e.g.*, skipping through all zeros when performing summation over a sparse tensor).

Unsurprisingly, computations that involve structured data are typically much more intricate than simply iterating through a dense array, as they should take the conventions imposed by the data's format into account. The problem of automatically generating code, which is specialised for particular representations of structured data it manipulates with, has received a lot of attention in recent years. Some of the most prominent works in this direction include the influential TACO framework for code generation (Kjolstad et al. 2017) and the Finch compiler (Ahrens et al. 2023).

Problem statement. Given the intricacy of computations on structured data, it is natural to wonder: *Do they produce the same results as their non-specialised counterparts on dense data representations?* This question suggests a verification challenge that we focus on in this work.

Existing efforts on verifying computations with structured data either provide high-level *domain-specific languages* (DSL) for encoding data formats as well as format-aware computations (Arnold et al. 2010), or encode data formats as *logical predicates* in a language of a solver used to verify refinement *w.r.t.* operations on the dense representations (Dyer et al. 2019). By offering a limited set of predefined abstractions, these DSL- or predicate-based approaches do not allow for verifying formats, whose encoding does not fit into their formalisms. None of these approaches come with a formal soundness proof regarding the executable code they produce. A recent work by Kellison et al. (2023) presented a mechanised correctness proof of a particular sparse matrix computation in a general-purpose separation logic embedded into Coq (Appel 2011). That verification effort relied on a tailored encoding of the chosen data format and required several format-specific lemmas, making it difficult to reuse any of its components for verifying programs over different formats.

The goal of this work is to provide a versatile framework for formally specifying and mechanically verifying diverse computations with structured data. We aim for an approach that (a) allows for naturally-looking and easy-to-state specifications, (b) can accommodate arbitrary programmatically-defined data formats, and (c) enables a high degree of proof reuse and automation.

In the rest of the paper, we argue that *relational program logics* are the right tool for this job.

Specifying manipulations with structured data. Let us immediately dive into computations with structured data. Fig. 1 shows a high-level outline of one such computations in Python-style pseudocode. The function `mv_prod` multiplies a compressed matrix, compactly encoded by a data structure `mc`, by a compressed vector, encoded via `vc`.[1] The two random access functions, `m_get` and `v_get`, whose signatures are given, *specify* the formats used to compress the matrix and the vector, correspondingly. For instance, given a compressed matrix `mc` and two indices (row and column

```
1  def m_get (mc, i, j) -> val:
2    # matrix format; body omitted
3
4  def v_get (vc, i) -> val:
5    # vector format; body omitted
6
7  def mv_prod(mc, vc) -> val[]:
8    let ans = malloc(vc.size)
9    for mp in partition(mc):
10     inner_prod (mp, vc, ans)
```

Fig. 1. Structured matrix/vector multiplication.

[1]Compressed tensors, such as `mc` and `vc` from Fig. 1, are typically encoded as tuples of 1-dimensional arrays with their sizes, but might be also represented as linked data structures (Chou and Amarasinghe 2022).

position), `m_get` returns a value (of some type `val`) stored in the "algebraic" (*i.e.*, dense) version of the matrix at those positions, without fully reconstructing the dense representation.

An efficient implementation `mv_prod` of a matrix/vector product, therefore, operates *directly* on the compressed representations `mc` and `vc` provided as its arguments. Such computations are typically structured by partitioning the representation of the matrix into separate chunks in some format-specific way, and storing the results of submatrix/vector products into the (dense) result vector `ans`. Functions of this sort, operating on sparse tensors, compressed via multiple different formats, are standard components of popular libraries for tensor computations, such as SciPy (Python), CsMatBase (Rust), and Eigen (C++). While at the time of this writing most of such function implementations are hand-crafted, they can be also automatically generated by domain-specific compilers, such as Finch (Ahrens et al. 2023), provided the corresponding computation on dense matrices/vectors and the programmatic format specifications such as `m_get` and `v_get`.

Importantly, the programmatic format definitions `m_get` and `v_get` are *not* used directly within the implementation of `mv_prod`. Instead, they should be considered as *specifications* of the compressed data that `mv_prod` manipulates with. Given such specifications, modern domain-specific compilers are capable of producing efficient format-specific implementations from *reference* implementations of the operations phrased in terms of dense representations (Ahrens et al. 2023; Sakka et al. 2017).

How do we specify correctness of `mv_prod`, asserting that its result is the same as a multiplying an *uncompressed* version of `mc` to that of `vc`? The key observation we make in this work is that such correctness statements can be stated naturally and concisely as Hoare-style triples that *relate* executions of the programmatic format specifications, such as `m_get` and `v_get`, to that of the operation in question. A common way to capture it formally is via a *hypersafety* judgment—a logical statement that relates results of executions of *multiple* possibly different computations. Hypersafety properties can be ascribed to a product of *independently* run programs in a form of (Hoare-style) *hyper-triples*, whose pre/postconditions constrain collections of pre/post-states of the programs in question, as well as their results. Therefore, a correctness specification for `mv_prod` can be phrased as the following hyper-triple relating its result to those of `m_get` and `v_get`:

$$\left\{ P_{\mathtt{mc}}(\cdot) \ * \ P_{\mathtt{vc}}(\cdot) \right\} \left[\begin{array}{rcl} 1 & : & \mathtt{mv\_prod(mc, vc)} \\ \langle 2, i, j \rangle_{\substack{i \in [0,M) \\ j \in [0,N)}} & : & \mathtt{m\_get(mc,}\ i,\ j\mathtt{)} \\ \langle 3, j \rangle_{j \in [0,N)} & : & \mathtt{v\_get(vc,}\ j\mathtt{)} \end{array} \right] \left\{ \begin{array}{l|l} a & \exists s, \mathtt{arr}(a,\ s) \ * \\ \overline{m} & \lceil \forall i, s[i] = \textstyle\sum_j \overline{m}(i,j)\overline{v}(j) \rceil \\ \overline{v} & \end{array} \right\} \tag{1}$$

The hypersafety specification (1) relates the execution of $1 + M \times N + N$ programs, with M and N being the number of the rows and the columns in the matrix. For instance, the component $\langle 2, i, j \rangle_{i \in [0,M),\, j \in [0,N)} : \mathtt{m\_get}(m, i, j)$ of the triple represents a *collection* of independent runs of `m_get` *indexed* by tuples $\langle 2, i, j \rangle$, with i ranging from 0 to $M - 1$ and j from 0 to $N - 1$. In such cases, we will refer to the set like $\{\langle 2, i, j \rangle \mid i \in [0, M),\ j \in [0, N)\}$ as *index sets*. The results of those $M \times N$ programs are bound to a sequence (hyper-)variable $\overline{m}$ of size $M \times N$, whose individual components are retrieved in the postcondition as $\overline{m}(i, j)$. The same intuition applies to the last N program components, whose results are bound to $\overline{v}$. The postcondition of the triple, thus, ensures that the result of the first program component a, is a base pointer of an array storing a mathematical sequence s, which is element-wise equal to the dot-product of the given matrix and vector.

The precondition features two Separation Logic-style predicates, $P_{\mathtt{mc}}$ and $P_{\mathtt{vc}}$, that define the memory shape of the corresponding compressed matrix/vector representations (*i.e.*, state that they are encoded in memory by array tuples). The definitions of those predicates are quite unremarkable, as they do not capture any relations between elements of a compressed encoding and its dense counterpart: that task is left to `m_get` and `v_get`. The only noteworthy component of the precondition is the $(\cdot)$ notation, which indicates *distinct copies* of the same symbolic state across multiple state

components indexed by elements of $\{1\} \cup \{\langle 2, i, j\rangle \mid \ldots\} \cup \{\langle 3, j\rangle \mid \ldots\}$, thus, supplying a valid (*i.e.*, safe to execute from) precondition to each one of the $1 + M \times N + N$ programs in the hyper-triple.

Key idea. The specification (1) of `mv_prod` owes its brevity to the idea of *parametrised* hyper-triples. Specifically, parametrisation means that the index sets used in program components can be expressed as *functions of the input state and variables* of the programs in question. To wit, the index sets in (1) are determined by the variables M and N, which correspond to the "logical" dimensions of the involved matrix, and are constrained by the predicate $P_{\mathtt{mc}}$.[2] The benefits of parametrisation are not limited to relatively concise specifications. While very simple conceptually, the ability to vary the number of the components in a hyper-triple depending on logical- and program-level variables is surprisingly useful for verifying properties of iterative computations on non-overlapping data, such as formats for compression or sparsity. As we will show in Sec. 2, the idea of parametrised hyper-triples makes it almost straightforward to decompose proofs of (stated relationally) correctness of compressed structure-manipulating programs featuring multiple loops.

Towards practical reasoning about parametrised hypersafety. So-called *relational* program logics are a well-studied formalism to specify and reason symbolically about hypersafety properties of programs, such as information-flow security and correctness of optimisations (Barthe et al. 2011, 2012; Benton 2004; Carbin et al. 2012; Nanevski et al. 2011; Yang 2007). Alas, the vast majority of existing relational logics are limited to specifying and proving 2-properties only, while our approach, exemplified by the spec (1), requires proofs about arbitrary-arity parametrised specifications.

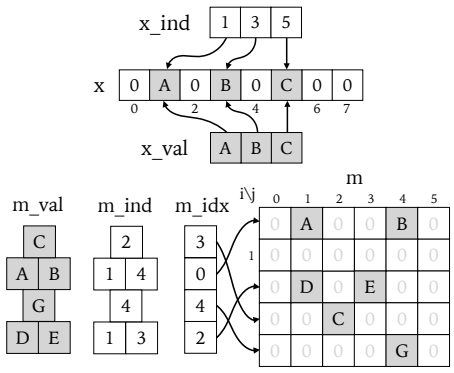
Two relatively new logics, CHL by Sousa and Dillig (2016) and LHC by D'Osualdo et al. (2022), provide proof rules for arbitrary-arity hypersafety triples. Unfortunately, CHL does not feature the necessary proof composition principles: it fixes the arity of a hyper-triple for the entire proof, making it impossible to employ specifications about subsets of the program components (or Hoare-style proofs for individual programs) in the context of a larger proof. LHC, on the other hand, allows for combining specifications of arbitrary *constant* arity in a single proof, but lacks rules for working with hyper-triples whose arity depends on *program-level variables*. This limitation of LHC turns out to be a show-stopper for constructing proofs about loops *within the logic*, requiring one to resort to proofs *in terms of semantics*. Furthermore, LHC is not a Separation Logic and, therefore, provides no support for modular reasoning about arrays and heap-based indirection. Finally, LHC only exists on paper: it is not implemented as a tool, and its soundness has not been mechanised.

In our quest of building a framework for machine-assisted verification of structured data manipulations, which is both foundational (*i.e.*, proven sound from first principles) and practical (*i.e.*, requires low annotation overhead), we are taking forward the ideas for compositional hypersafety reasoning pioneered by LHC (D'Osualdo et al. 2022), enhancing them with three new aspects:

(1) adding new rules for decomposing proofs about *arbitrary arity-parametric* hyper-triples;
(2) providing specialised *proof rules for loops* that take advantage of hyper-triple parametrisation;
(3) introducing support for *state-local* hypersafety proofs in the style of Separation Logic.

The result is the novel logical framework for Hoare-style hypersafety proofs, dubbed *Logic for Graceful Tensor Manipulation* (LGTM). We implemented LGTM as a Separation Logic (Reynolds 2002) embedded into the Coq proof assistant. The shallow nature of the embedding allowed us to use Coq's higher-order logic capabilities for implementing rules for handling parametrised triples in the form of ordinary Coq lemmas. Furthermore, the careful design of rules for hyper-triple decomposition and reasoning about loops made it possible to engineer effective proof automation resulting in very short and conceptually simple machine-assisted verification proofs.

[2] In our example, we made those variables explicit in the specification, omitting the full signature of $P_{\mathtt{mc}}$ for brevity.



(a) SV and uCSR, pictographically

```
1 def sv_get(x_ind, x_val, i):
2   ind = index(x_ind, i)
3   if (ind != -1):
4     return x_val[ind]
5   else:
6     return 0
```

(b) Sparse Vector (SV) format

```
1 def ucsr_get(m_idx, m_ind, m_val, i, j):
2   ind = index(m_idx, i)
3   if (ind != -1):
4     return sv_get(m_ind[ind], m_val[ind], j)
5   else:
6     return 0
```

(c) Unordered Compressed Sparse Row (uCSR)

Fig. 2. Sparse Vector and Unordered Compressed Sparse Row formats

Contributions and outline. To summarise, in this work we make the following contributions.

- Our first conceptual contribution is an observation that specification and verification of manipulations with structured data can be phrased in terms of reasoning about hypersafety properties.
- Our second conceptual contribution is an idea of a *parametrised* hyper-triple, whose arity depends on attributes of the data under manipulation. We show that parametrisation enables hypersafety proofs based on the structure of the *data* rather than the structure of the *program* (Sec. 2).
- Our main theoretical contribution is the Logic for Graceful Tensor Manipulation (LGTM)—a Hoare-style relational separation logic for hypersafety properties that provides effective reasoning principles about array-manipulating programs with loops (Sec. 3). We mechanised the metatheory, the rules, and the soundness proof of LGTM in the Coq proof assistant.
- Our practical contribution is a set of Coq-based automation techniques for mechanised proofs in LGTM that allow for short and intellectually manageable hypersafety proofs (Sec. 4). We have evaluated our implementation of LGTM by mechanising proofs for 13 characteristic case studies that implement computations with sparse and compressed tensors in various formats (Sec. 5). We show that, in comparison with related foundational efforts, mechanised proofs in LGTM are about 10 times shorter than those done in a general-purpose non-relational Separation Logic.

2 A TOUR OF LGTM

In this section, we showcase the reasoning principles of LGTM as a logic formalism, postponing the demonstration of its verification capabilities as a tool until Sec. 4 and 5. To do so, we specify and verify a program that computes a product of a compressed sparse matrix and a sparse vector, represented via the Unordered Compressed Sparse Row (uCSR) format and a Sparse Vector (SV), correspondingly, storing the result into an uncompressed vector represented by an array.

Fig. 2a graphically depicts the data layout in both SV and uCSR formats. The SV format is a standard representation for sparse vectors. It achieves great rates of compression for sparse (*i.e.*, mostly-zero) vectors by storing only their *non-zero* values in an array `x_val` together with their indices at the same positions in an array `x_ind`. The programmatic encoding of SV is provided by the `sv_get` function shown in Fig. 2b. This function "decompresses" a sparse vector encoded by the pair of arrays (`x_val`, `x_ind`) by retrieving the $\texttt{i}^{\text{th}}$ value of the dense representation `x` from its compressed representation via `x_ind` and `x_val`. It does so by first finding the index of the element corresponding to `i` in the array `x_ind`, calling the `index` function (line 2). The implementation of

`index` simply performs a linear scan through the contents of `x_ind`, returning a position of its second argument, if it's present, and -1 otherwise (remember, `sv_get` only serves as a format specification, hence its efficiency is not important). Depending on the search result stored into `ind`, the rest of `sv_get`'s implementation returns either the corresponding `x_val` element, or zero (lines 3-6).

The uCSR format for sparse matrices is a modification of the popular Compressed Sparse Row (CSR) format, designed to be more efficient by enabling adaptive tiling (Hong et al. 2019). uCSR adopts a compression principle similar to that of SV: rather than storing each row of the matrix, its only stores rows that have non-zero elements. To this end, uCSR uses an *unordered* array `m_idx` to store the positions of non-empty rows. Those rows are in turn compressed using SV, with associated two-dimensional arrays `m_ind` and `m_val` storing indices and values of non-empty cells for each row.[3] This intuition is mirrored in the programmatic specification of uCSR shown in Fig. 2c, which takes three arrays `m_idx`, `m_ind`, and `m_val`, as well as the dense row/column indices `i` and `j`, and retrieves the corresponding element by calling `sv_get` on its actual row encoding.

The implementation of the sparse matrix/vector product is given in Fig. 3; we have adopted it from the output produced by the Finch compiler (Ahrens et al. 2023). The function `spmspv` takes a sparse matrix m, encoded in uCSR format via three arrays: `m_idx`, `m_ind`, and `m_val`, as well as a sparse vector x, encoded via `x_ind` and `x_val`. Its result is a dense array `ans`, populated element-wise by the components of the dot-product $m \cdot x$. The implementation of `spmspv`, also uses the height `M` of the matrix, which we assume to be a global variable. The body of `spmspv` first iterates over all non-zero rows in the order determined by `m_idx` (line 3). At each iteration of the **`for`**-loop it calls `spvspv`, computing the product of two sparse vectors, `m_idx[i]`$^{\text{th}}$ row of the matrix m and the vector x. The **`while`**-loop in `spvspv` (lines 10-18) *co-iterates* over a pair of sparse vectors, visiting only non-zero values of *both* arrays. It does so by maintaining the positions in the corresponding vectors via variables `iY` and `iX`, advancing them in a way so that the result stored in `s` is changed only when indices of non-zero elements in the two vectors coincide (line 12).

```
1  def spmspv(m_idx, m_ind, m_val, x_ind, x_val):
2    ans = malloc(M)
3    for i in range(0, length(m_idx)):
4      ans[m_idx[i]] =
5        spvspv(m_ind[i], m_val[i], x_ind, x_val)
6    return ans
7
8  def spvspv(y_ind, y_val, x_ind, x_val):
9    s, iY, iX = 0, 0, 0
10   while (iY < length(y_ind)) &&
11         (iX < length(x_ind)):
12     if y_ind[iY] = x_ind[iX]:
13       s += y_val[iY] * x_val[iX]
14       iY++; iX++
15     else if y_ind[iY] < x_ind[iX]:
16       iY++
17     else if x_ind[iX] < y_ind[iY]:
18       iX++
19   return s
```

Fig. 3. Sparse Matrix/Vector product, SpMSpV (top) and Sparse Vector/Vector product, SpVSpV (bottom).

To specify the result of `spmspv`, let us introduce some notation. We will abbreviate all unmodified arrays that encode the involved data by the following Separation Logic (SL)-style predicates:

$$\begin{array}{rcl} \mathit{Arrs}_{\mathtt{m}} & \triangleq & \mathtt{arr}(\mathtt{m}_{\mathtt{ind}},\ m_{ind}) \ *\ \mathtt{arr}(\mathtt{m}_{\mathtt{val}},\ m_{val}) \ *\ \mathtt{arr}(\mathtt{m}_{\mathtt{idx}},\ m_{idx}) \\ \mathit{Arrs}_{\mathtt{x}} & \triangleq & \mathtt{arr}(\mathtt{x}_{\mathtt{ind}},\ x_{ind}) \ *\ \mathtt{arr}(\mathtt{x}_{\mathtt{val}},\ x_{val}) \\ \mathit{Arrs} & \triangleq & \mathit{Arrs}_{\mathtt{m}} \ *\ \mathit{Arrs}_{\mathtt{x}} \end{array} \tag{2}$$

That is, for instance, $\mathit{Arrs}_{\mathtt{m}}$ combines three arrays responsible for representing a sparse matrix m, with their base pointers (*e.g.*, $\mathtt{m}_{\mathtt{ind}}$) given in `monospace` font and payloads (*e.g.*, m_{ind}) given in *italic*. The disjointness of the arrays is asserted by the SL *separating conjunction* connective ($*$).

The desired specification for `spmspv` is, therefore, captured by the following hypersafety triple:

[3] In a more realistic encoding, an additional dense array `m_ptr` is used to store locations of the boundaries between rows in *one-dimensional* arrays `m_val` and `m_ind`. In this section, for simplicity of the presentation, we treat `m_ind` and `m_val` as *two-dimensional* arrays. In our mechanisation, however, we verify the encoding involving the `m_ptr` array (*cf.* Sec. 5).

$$\textsc{SeqU1}\ \frac{\{\ P\ \}\,[\iota:\ p_1]\,\{\ x\ \mid\ H\ \} \qquad \{\ H\ \}\,[\iota:\ p_1',\ S:\ \mathcal{P}_2]\,\{\ x,\overline{z}\ \mid Q(\overline{z})\ \}}{\{\ P\ \}\,[\iota:p_1;p_1',\ S:\mathcal{P}_2]\,\{\ x,\overline{y}\ \mid Q(x\overline{y})\ \}}$$

$$\textsc{SeqU2}\ \frac{\{P\}\,[\iota:\ p_1,\ S:\ \mathcal{P}_2]\,\{\ x,\overline{z}\ \mid H(\overline{z})\ \} \qquad \forall\overline{z},\{\ H(\overline{z})\ \}\,[\iota:\ p_1']\,\{\ x\ \mid\ Q(x\overline{z})\ \}}{\{\ P\ \}\,[\iota:p_1;p_1',\ S:\ \mathcal{P}_2]\,\{\ x_1,\overline{x_2}\ \mid Q(x_1\overline{x_2})\ \}}$$

$$\textsc{Product}\ \frac{\texttt{local}(\{P_i,Q_i\},i) \qquad \forall i,\{\ \lceil i\in S\rceil\ *\ P_i\ \}\,[i:\ \mathcal{P}(i)]\,\{\ x\ \mid\ Q_i(x)\ \}}{\left\{\ \mathop{*}_{i\in S} P_i\ \right\}\,[S:\ \mathcal{P}]\left\{\ \overline{x}\ \middle|\ \mathop{*}_{i\in S} Q_i(\overline{x}(i))\ \right\}}$$

Fig. 4. Rules for sequential composition and independent programs.

$$\left\{Arrs(\cdot)\right\}\begin{bmatrix} 1 & : \texttt{spmspv}(\texttt{m}_{\texttt{idx}},\texttt{m}_{\texttt{ind}},\texttt{m}_{\texttt{val}},\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}}) \\ \langle 2,i,j\rangle_{\substack{i\in[0,M)\\ j\in[0,N)}} & : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}},\texttt{m}_{\texttt{ind}},\texttt{m}_{\texttt{val}},i,j) \\ \langle 3,j\rangle_{j\in[0,N)} & : \texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j) \end{bmatrix}\left\{\begin{array}{l|l} a & \exists s,\texttt{arr}(a(1),\ s)\ * \\ \overline{m} & \lceil\forall i,s[i]=\sum_j \overline{m}(i,j)\overline{x}(j)\rceil\ * \\ \overline{x} & Arrs(\cdot) \end{array}\right\} \tag{3}$$

The spec assumes that the width N of the matrix m is a global variable. It makes use of the $(\cdot)$ notation that "replicates" a SL across all state components in the pre/postcondition. The $\lceil\cdot\rceil$ notation stands for *pure* assertions that constrain values and do not depend on the shape of the heaps. Notice that, unlike $Arrs(\cdot)$, the *spatial* part $\texttt{arr}(a(1),\ s)$ is only present in the first component's post-state.

What's next? In the rest of this section, we will verify the validity of the specification (3) via the rules of LGTM. We will start by focusing on the **for**-loop from the spmspv's body, "peeling away" all calls to ucsr_get that do not correspond to any of the loop's iterations (Sec. 2.1 – Sec. 2.2). We will then split the remaining calls into groups (one group per non-zero row) and relate the result of each one to one iteration of the **for**-loop (Sec. 2.3). We will then show how the obtained goal can be proven *compositionally* by using an independently verified specification of spvspv (Sec. 2.4). Finally, we will sketch the proof of spvspv, showcasing LGTM's rule for **while**-loops (Sec. 2.5).

2.1 Proofs about Sequential Composition and Independent Programs

To begin with the proof of (3), we unfold the definition of spmspv in the first component, and omit ranges of variables i and j, obtaining the following goal to prove:

$$\left\{Arrs(\cdot)\right\}\begin{bmatrix} 1 & : \texttt{ans} = \texttt{malloc(M)};\ldots \\ \langle 2,i,j\rangle & : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}},\texttt{m}_{\texttt{ind}},\texttt{m}_{\texttt{val}},i,j) \\ \langle 3,j\rangle & : \texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j) \end{bmatrix}\left\{\begin{array}{l|l} a & \exists s,\texttt{arr}(a(1),\ s)\ * \\ \overline{m} & \lceil\forall i,s[i]=\sum_j \overline{m}(i,j)\overline{x}(j)\rceil \\ \overline{x} & *\ Arrs(\cdot) \end{array}\right\} \tag{4}$$

Notice that the **for**-loop at lines 3-5 of Fig. 3 only "visits" rows of m that correspond to m_idx[i] for some i. In this subsection, we will show how to reduce the verification goal (4) to the one that only mentions those ucsr_get-components whose (row) argument i is present as an element in m_idx.

2.1.1 Rules for Sequential Composition. Our first step is to get to the **for**-loop in the first component of the goal (4), which will require us to symbolically step through the malloc statement, while "focusing" on the corresponding component. This focusing capability is achieved by the SeqU1 shown in Fig. 4, which allows one to split the index set of a hyper-triple into two disjoint subsets, $\{\iota\}$ and S, symbolically executing the first component p_1 of the sequential composition $p_1;p_1'$ that corresponds to the ι-indexed component of the entire hyper-triple.[4] Applying SeqU1 with $\iota = 1$ to the goal (4) allows us to execute the malloc statement using the proof rule for allocation, which is relatively standard for Separation Logic-style formalisms, so we postpone its presentation until

[4]To avoid clutter, the rules assume that execution of the "projected" components, such as $\iota:p_1$ in SeqU1, are implicitly *framed* with the corresponding hyper-heap footprint of the S-indexed component, which it keeps unmodified.

Sec. 3. The first premise of SeqU1, thus, takes the form of the following goal, where we use **0** to denote a 0-filled vector of a suitable size, evident from the context:

$$\{Arrs(\cdot)\}\ \left[\ 1 : \mathtt{ans} = \mathtt{malloc(M)}\ \right]\ \{Arrs(\cdot)\ *\ \mathtt{arr(ans(1),\ 0)}\}$$

Here the `(1)` syntax in `arr(ans(1), 0)` means that `ans` is a pointer allocated in the 1-indexed component of the symbolic state (*i.e.*, the heap where the first component runs). Abbreviating the `for`-loop as f we obtain the following triple as the result of the second obligation of SeqU1:

$$\left\{\begin{array}{c}\mathtt{arr(ans(1),\ 0)}\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll}1 & : f;\ \mathtt{return\ ans}\\ \langle 2,i,j\rangle & : \mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)\\ \langle 3,j\rangle & : \mathtt{sv\_get}(\mathtt{x_{ind}},\mathtt{x_{val}},j)\end{array}\right]\left\{\begin{array}{l|l}a & \exists s, \mathtt{arr}(a(1),\ s) *\\ \overline{m} & \lceil \forall i, s[i] = \sum_j \overline{m}(i,j)\overline{x}(j)\rceil\\ \overline{x} & *\ Arrs(\cdot)\end{array}\right\} \quad (5)$$

The goal is further simplified by stripping off the `return` statement via the SeqU2 and unfolding `arr(ans(1), 0)` into an iterated separating conjunction $\circledast_{i=0}^{M}\ \mathtt{ans(1)} + i \mapsto 0$ over a set of *points-to heaplets* (*i.e.*, symbolic heap entries), indexed by their offsets i from the base array address `ans(1)`:

$$\left\{\begin{array}{c}\circledast_{i=0}^{M}\ \mathtt{ans(1)} + i \mapsto 0\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll}1 & : \mathtt{for}\ i\ \mathtt{in\ range}(0, |m_{idx}|)\ \{p(i)\}\\ \langle 2,i,j\rangle & : \mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)\\ \langle 3,j\rangle & : \mathtt{sv\_get}(\mathtt{x_{ind}},\mathtt{x_{val}},j)\end{array}\right]\left\{\begin{array}{l|l}- & \exists s, \circledast_{i=0}^{M}\ \mathtt{ans(1)} + i \mapsto s[i] *\\ \overline{m} & \lceil \forall i, s[i] = \sum_j \overline{m}(i,j)\overline{x}(j)\rceil\\ \overline{x} & *\ Arrs(\cdot)\end{array}\right\}$$

Finally, we abbreviate the loop's body as $p(i)$, and simplify the postcondition by substituting each value of the `ans`' payload sequence $s[i]$ with the correspondent sum expression:

$$\left\{\begin{array}{c}\circledast_{i}\ \mathtt{ans(1)} + i \mapsto 0\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll}1 & : \mathtt{for}\ i\ \mathtt{in\ range}(0, |m_{idx}|)\ \{p(i)\}\\ \langle 2,i,j\rangle & : \mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)\\ \langle 3,j\rangle & : \mathtt{sv\_get}(\mathtt{x_{ind}},\mathtt{x_{val}},j)\end{array}\right]\left\{\begin{array}{l|l}- & \circledast_{i}\ \mathtt{ans(1)} + i \mapsto \\ \overline{m} & \qquad \sum_j \overline{m}(i,j)\overline{x}(j)\\ \overline{x} & *\ Arrs(\cdot)\end{array}\right\} \quad (6)$$

2.1.2 Reasoning with Component-Local Assertions. Remember that array $\mathtt{m_{idx}}$ contains indices of non-zero rows of the matrix m, so that the `for`-loop from the first component will iterate only over such indices. This suggests us to first get rid of all the components with $\mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)$ for $i \notin m_{idx}$, and try to relate the rest of those with each iteration of the `for`-loop. To implement the former, we note that for each index i outside of the payload m_{idx} the result of $\mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)$ will be 0 for any j. This fact can be stated as the following triple:

$$\{Arrs(\cdot)\}\left[\langle 2,i,j\rangle_{\substack{i\notin m_{idx}\\ j\in[0,N)}} : \mathtt{ucsr\_get}(\mathtt{m_{idx}},\mathtt{m_{ind}},\mathtt{m_{val}},i,j)\right]\left\{\begin{array}{l|l}\overline{m} & \lceil \forall i \notin m_{idx}\ \forall j, \overline{m}(i,j) = 0\rceil\\ & *\ Arrs(\cdot)\end{array}\right\} \quad (7)$$

To see how to deal with such triples, let us consider the following simple verification goal:

$$\{\ s(\cdot) \mapsto y\ \}\ [i_{0\le i<N} : \mathtt{s\ +=}\ i]\ \{\ \_\ |\ s(\cdot) \mapsto y + i\ \}$$

Intuitively, proving this triple is equivalent to proving N independent 1-safety triples of the form

$$\{\ s(i) \mapsto y\ \}\ [i : \mathtt{s\ +=}\ i]\ \{\ \_\ |\ s(i) \mapsto y + i\ \}$$

To capture this intuition LGTM features the rule Product (*cf.* Fig. 4) that allows one to divide the state assertions in the pre-/postcondition (indexed by a set S) into $|S|$ disjoint assertions, each being *local* to a particular index. The idea of locality comes from the LGTM memory model, so that its state assertions constrain not just the shapes of the heap but also the component in which they reside, so that, *e.g.*, $x(1) \mapsto 0$ and $x(2) \mapsto 0$ are not satisfied by the same heap. Postponing a more formal treatment of LGTM state till Sec. 3, here we say that an LGTM assertion is local to an index ι if it can only be satisfied by states that belong to the ι component. With the Product rule, the assumed locality of the individual specifications and independence of the program runs, it suffices to prove the correspondent triple for each individual program p from a hyper-triple.

The postcondition of the specification (7) can be rewritten using the following equivalence:

$$\lceil \forall i \notin m_{idx}\, \forall j, \overline{m}(i,j) = 0 \rceil \iff \mathop{\Large *}_{i \notin m_{idx}, j} \lceil \overline{m}(i,j) = 0 \rceil$$

With the right-hand side of the equivalence above, we can apply the Product rule, reducing (7) to

$$\left\{ \begin{array}{c} \lceil i \notin m_{idx} \wedge j \in [0,N) \rceil * \\ Arrs(\cdot) \end{array} \right\} [\langle 2,i,j\rangle : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}}, \texttt{m}_{\texttt{ind}}, \texttt{m}_{\texttt{val}}, i, j)] \left\{ m \;\middle|\; \begin{array}{l} \lceil m = 0 \rceil * \\ Arrs(\cdot) \end{array} \right\} \tag{8}$$

This is a 1-safety property, which is stated in terms of a traditional Hoare-style triple of the form $\{\ P\ \}\ [\iota : p]\ \{\ x \mid Q(x)\ \}$ for a program p. To discharge such obligations, LGTM provides the standard set of Separation Logic rules. We omit this proof, which relies on an independently proven specification of $\texttt{index}(\texttt{m}_{\texttt{idx}}, i)$; it can be found in our Coq development (Gladshtein et al. 2024a).

2.2 Focusing and Framing

At this point, one can think of LGTM as of a "Separation Logic going hyper", with the heaps being replicated over index sets. An astute reader could have noticed that the rule Product exploits the (hyper-)locality of spatial triples to provide a version of SL's *frame* rule that combines multiple independent *unary triples* into a single *hyper-triple*. In our next steps, we will keep taking advantage of the SL nature of LGTM that enables principles of modular reasoning about hypersafety.

Using (7), we can remove all runs of `ucsr_get` for m's zero-only rows of m from the goal (6). To do so, we will make use the Focus rule, shown in Fig. 5. Applying it, followed by the framing-out of some no longer useful pre-/postconditions components (via standard SL framing rules), will make us ready to handle the **for**-loop in the first component. The Focus rules allows us to advance executions of the programs within S_1 (a subset of the whole index set $S = S_1 \uplus S_2$), keeping the rest of the state unchanged. The notation $\mathcal{P}_1 \uplus \mathcal{P}_2$ stands for the disjoint union of two sets of indexed programs. Notice that the range of the result vector $\overline{y}$ of $S_2 : \mathcal{P}$ would be S_2, so we will extend it to the whole S by appending with the result of $S_1 : \mathcal{P}$, vector $\overline{x}$ of range S_1. We apply the Focus rule to the goal (6) by taking $S_1 = \{\langle 2,i,j\rangle \mid i \notin m_{idx}\}$ and $S_2 = \{1\} \cup \{\langle 2,i,j\rangle \mid i \in m_{idx}\} \cup \{\langle 3,j\rangle \mid \ldots\}$. Using the spec (7) on the S_1-indexed part earns us the conjunct $\lceil \forall i \notin m_{idx}\, \forall j, \overline{m}(i,j) = 0 \rceil$ in the precondition of the following goal, which corresponds to the second premise of the Focus:

Focus

$$\frac{\begin{array}{c} S = S_1 \uplus S_2 \\ \{\ P\ \}\ [S_1 :\ \mathcal{P}_1]\ \{\ \overline{x} \mid H(\overline{x})\ \} \\ \forall \overline{x},\ \{\ H(\overline{x})\ \}\ [S_2 :\ \mathcal{P}_2]\ \{\ \overline{y} \mid Q(\overline{x}\overline{y})\ \} \end{array}}{\{\ P\ \}\ [S :\ \mathcal{P}_1 \uplus \mathcal{P}_2]\ \{\ \overline{z} \mid Q(\overline{z})\ \}}$$

Fig. 5. Focus rule

$$\left\{ \begin{array}{c} \mathop{\Large *}_{i} \texttt{ans}(1) + i \mapsto 0 \\ * \ Arrs(\cdot) \end{array} \right\} \left[\begin{array}{cc} 1 & : \texttt{for}\ i\ \texttt{in}\ \texttt{range}(0, |m_{idx}|)\ \ \{p(i)\} \\ \langle 2,i,j\rangle_{i \in m_{idx}} & : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}}, \texttt{m}_{\texttt{ind}}, \texttt{m}_{\texttt{val}}, i, j) \\ \langle 3,j\rangle & : \texttt{sv\_get}(\texttt{x}_{\texttt{ind}}, \texttt{x}_{\texttt{val}}, j) \end{array} \right] \left\{ \begin{array}{c|l} - & \\ \overline{m} & \mathop{\Large *}_{i \in m_{idx}} \texttt{ans}(1) + i \mapsto \sum_j \overline{m}(i,j)\overline{x}(j) \\ \overline{x} & \mathop{\Large *}_{i \notin m_{idx}} \texttt{ans}(1) + i \mapsto 0 \\ & * \ Arrs(\cdot) \end{array} \right\}$$

Now, we can once again exploit the Separation Logic capabilities of LGTM, framing out the part $\mathop{\Large *}_{i \notin m_{idx}} \texttt{ans}(1) + i \mapsto 0$ from both pre- and postconditions, obtaining the following goal:

$$\left\{ \begin{array}{c} \mathop{\Large *}_{i \in m_{idx}} \texttt{ans}(1) + i \mapsto 0 \\ * \ Arrs(\cdot) \end{array} \right\} \left[\begin{array}{cc} 1 & : \texttt{for}\ i\ \texttt{in}\ \texttt{range}(0, |m_{idx}|)\ \ \{p(i)\} \\ \langle 2,i,j\rangle_{i \in m_{idx}} & : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}}, \texttt{m}_{\texttt{ind}}, \texttt{m}_{\texttt{val}}, i, j) \\ \langle 3,j\rangle & : \texttt{sv\_get}(\texttt{x}_{\texttt{ind}}, \texttt{x}_{\texttt{val}}, j) \end{array} \right] \left\{ \begin{array}{c|l} - & \\ \overline{m} & \mathop{\Large *}_{i \in m_{idx}} \texttt{ans}(1) + i \mapsto \sum_j \overline{m}(i,j)\overline{x}(j) \\ \overline{x} & * \ Arrs(\cdot) \end{array} \right\} \tag{9}$$

The triple (9) is an examplary case of a parametrised hypersafety specification: its arity is determined by the data m_{idx} that serves as a selector for the `ucsr_get` programs. Phrasing it this way allowed us to bring the iterations of `spmspv`'s **for**-loop in line with the indexing structure of the specification components, so we could handle the remaining proof using a dedicated LGTM rule for loops.

$$\frac{S = \bigcup_{i=0}^{N-1} S_i \quad \texttt{local}(R_i, S_i) \quad \forall \overline{x}, i < N, \left\{\ I(i,\overline{x}) \ *\ R_i\ \right\}\ \left[\ \iota : p,\ S_i : \mathcal{P}_i\right]\ \left\{\ z, \overline{y}\ \middle|\ I(i+1, \overline{xy})\ \right\}}{\left\{\ I(0,\overline{\emptyset}) \ *\ \mathop{\Large *}_{i=0}^{N-1} R_i\ \right\}\ \left[\ \iota : \texttt{for}\ i\ \texttt{in}\ \texttt{range}(0,N)\ \{p\},\ S : \mathcal{P}\right]\ \left\{\ z, \overline{y}\ \middle|\ I(N, \overline{y})\ \right\}}\ \textsc{For}$$

Fig. 6. A rule for `for`-iteration.

2.3 Iteration and Array Splitting

To prove the goal (9), we observe that the i^{th} iteration of the for-loop in the first component computes the dot product of the vector x and the i^{th} row of the matrix m. This suggest partitioining the verification task by *aligning* (*i.e.*, relating the result of) each i^{th} iteration of that loop with a sequence of runs of $\texttt{sv\_get}(\texttt{x}_{\texttt{ind}}, \texttt{x}_{\texttt{val}}, j)$ and $\texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}}, \texttt{m}_{\texttt{ind}}, \texttt{m}_{\texttt{val}}, i, j)$ for all j, $0 \le j < N$, as those runs compute values of x and the i^{th} row of m. This pattern of splitting a set of independent runs into "batches" to align with individual loop iterations in the "main" program is so common in structured data computations that LGTM introduces a dedicated rule For to handle it (Fig. 6).

To understand how the rule works, first notice its premise $S = \bigcup_{i=0}^{N-1} S_i$ that splits the index set S into N parts S_i. Each part corresponding to a program "batch" to be aligned with one iteration of the `for`-loop. The precondition of the conclusion features an assertion $\mathop{\Large *}_{i=0}^{N-1} R_i$ that describes a matching split of the pre-state required to run each of those batches (hence the locality side condition). The most interesting aspect of the rule is the *loop invariant* I, which is an assertion parametrised by (i) an integer that corresponds to the *next* value of the loop counter and (ii) a hyper-value that collects the results of all program batches that have been already aligned with the "prefix" of the loop. That is, in the conclusion of the rule, it is assumed that the assertion $I(0, \overline{\emptyset})$ holds initially: which corresponds to zero iterations and no aligned batches executed so far ($\overline{\emptyset}$ is an empty hyper-value). At the end, $I(N, \overline{y})$ means that the loop has been fully executed and all of the aligned programs' results are bound by $\overline{y}$. The inductive step of the rule, *i.e.*, the premise defining the goal for the body of the loop p, assumes that the invariant initially holds for some $i < N$ and a hyper-value $\overline{x}$ (of size i, which we omit from the rule for brevity), and asserts in the postcondition that at the end the invariant holds on the next index $i + 1$ and the combined hyper-value $\overline{xy}$.[5]

To proceed with our proof of (9) using the For rule, we first split the set of its indices as follows:

$$\langle 2, i, j\rangle_{i \in m_{idx}, j \in [0,N)} \cup \langle 3, j\rangle_{j \in [0,N)} = \bigcup_{i \in 0}^{|m_{idx}|-1} S_i = \bigcup_{i \in 0}^{|m_{idx}|-1} \left(\langle 2, m_{idx}[i], j\rangle_{j \in [0,N)} \cup \langle 3, j\rangle_{j \in [0,N)}\right) \quad (10)$$

Next, we define the loop invariant as follows:

$$I_{\texttt{for}}(i, \overline{mx}) \ \triangleq\ \mathop{\Large *}_{j=0}^{i-1} \texttt{ans}(1) + j \mapsto \textstyle\sum_{k=0}^{N} \overline{m}(j,k)\overline{x}(k)\ *\ \mathop{\Large *}_{j=i}^{|m_{idx}|-1} \texttt{ans}(1) + j \mapsto 0 \quad (11)$$

The invariant states that (a) a j^{th} element from the prefix of the array $\texttt{ans}(1)$, is already filled with the dot product of x and j^{th} row of m (*i.e.*, the dot-product of results of `ucsr_get` and `sv_get`), for $0 \le j < i$ and (b) that the suffix of the array is filled with zeros. After applying For and advancing the code of `spmspv` (Fig. 3) up to the line 5, we obtain the following goal, for some $i \in [0, |m_{idx}|)$:

$$\left\{\begin{array}{l}\texttt{ans}(1) + \\ m_{idx}[i] \mapsto 0 \\ *\ Arrs(\cdot)\end{array}\right\}\ \left[\begin{array}{cl} 1 & : \texttt{ans}[\texttt{m}_{\texttt{idx}}[\texttt{i}]] = \texttt{spvspv}(\ldots) \\ \langle 2, i, j\rangle_{j \in [0,N)} & : \texttt{ucsr\_get}(\texttt{m}_{\texttt{idx}}, \texttt{m}_{\texttt{ind}}, \texttt{m}_{\texttt{val}}, m_{idx}[i], j) \\ \langle 3, j\rangle_{j \in [0,N)} & : \texttt{sv\_get}(\texttt{x}_{\texttt{ind}}, \texttt{x}_{\texttt{val}}, j)\end{array}\right]\ \left\{\begin{array}{c|l} - & \texttt{ans}(1) + m_{idx}[i] \mapsto \\ \overline{m} & \quad \textstyle\sum_j \overline{m}(i,j)\overline{x}(j) \\ \overline{x} & *\ Arrs(\cdot)\end{array}\right\}$$

[5] As a curiosity, this means that I has a dependent type: the value of its first parameter determines the arity of the second.

Now we can advance lines 2 and 3 in uscr_get (Fig. 2c): $\texttt{index}(\texttt{m}_{\texttt{idx}}, m_{idx}[i])$ evaluates to i, triggering then-branch of the **if**-statement (line 4), leaving us with the following goal:

$$\left\{\begin{array}{l}\texttt{ans(1)}+\\ m_{idx}[i]\mapsto 0\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll}1 & :\texttt{ans}[\texttt{m}_{\texttt{idx}}[\texttt{i}]]=\texttt{spvspv}(\ldots)\\ \langle 2,i,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{m}_{\texttt{ind}}[\texttt{i}],\texttt{m}_{\texttt{val}}[\texttt{i}],j)\\ \langle 3,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j)\end{array}\right]\left\{\begin{array}{l|l}- & \texttt{ans(1)}+m_{idx}[i]\mapsto\\ \overline{m} & \quad\sum_j \overline{m}(i,j)\overline{x}(j)\\ \overline{x} & *\ Arrs(\cdot)\end{array}\right\} \tag{12}$$

We can further reduce the goal using SeqU1 and omitted symbolic execution rules for unary triples:

$$\left\{Arrs(\cdot)\right\}\left[\begin{array}{ll}1 & :\texttt{spvspv}(\texttt{m}_{\texttt{ind}}[\texttt{i}],\texttt{m}_{\texttt{val}}[\texttt{i}],\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}})\\ \langle 2,i,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{m}_{\texttt{ind}}[\texttt{i}],\texttt{m}_{\texttt{val}}[\texttt{i}],j)\\ \langle 3,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j)\end{array}\right]\left\{\begin{array}{l|l}s & \lceil s=\sum_j \overline{m}(i,j)\overline{x}(j)\rceil\\ \overline{m} & *\ Arrs(\cdot)\\ \overline{x} & \end{array}\right\} \tag{13}$$

2.4 Domain Substitution and Specification Composition

At this point, we have *almost* reduced the verification of the spmspv specification to verifying the correctness of its auxiliary function spvspv *w.r.t.* the respective format definition sv_get: our remaining goal (13) looks pretty much like a specification of a sparse vector/vector dot-product.

The actual desired specification of spvspv is captured via the following hyper-triple:

$$\left\{\begin{array}{l}Arrs_{\texttt{y}}(\cdot)*\\ Arrs_{\texttt{x}}(\cdot)\end{array}\right\}\left[\begin{array}{ll}1 & :\texttt{spvspv}(\texttt{y}_{\texttt{ind}},\texttt{y}_{\texttt{val}},\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}})\\ \langle 2,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{y}_{\texttt{ind}},\texttt{y}_{\texttt{val}},j)\\ \langle 3,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j)\end{array}\right]\left\{\begin{array}{l|l}s & \lceil s=\sum_j \overline{y}(j)\overline{x}(j)\rceil\\ \overline{y} & \\ \overline{x} & \end{array}\right\} \tag{14}$$

The predicate $Arrs_{\texttt{y}}(\cdot)$ abbreviates $\texttt{arr}(\texttt{y}_{\texttt{ind}}(\cdot),\ y_{ind})\ *\ \texttt{arr}(\texttt{y}_{\texttt{val}}(\cdot),\ y_{val})$, similarly for $Arrs_{\texttt{x}}(\cdot)$. To see how can we get (13) out of (14) recall that in (13), the implicitly universally quantified (outside of the entire hyper-triple) variable i ranges from 0 up to $|m_{ind}|-1$. Since i is *fixed* across all programs in the second component, we get rid of it via *domain substitution*.

A simplified version of the substitution rule Subst is presented in Fig. 7. Its key component is an *injective* mapping ϕ from the index set S to a set S', which embodies the reindexing in question. This rule applies ϕ to the index set of the triple S. It also changes $\mathcal{P}$ to $\mathcal{P}'$, which is a reindexed $\mathcal{P}$ *w.r.t.* to ϕ. Finally, it re-indexes the components of both pre- and postcondition by applying ϕ (denoted, *e.g.*, $\phi[P]$), transforming, in particular, spatial assertions of the kind $x(\iota)\mapsto v$ into $x(\phi(\iota))\mapsto v$. For composite assertions, containing separation conjunction, and big separation conjunction, ϕ just iteratively distributes over their logical connectives. This transformation does not affect the validity of the assertions, since ϕ is injective.

$$\textsc{Subst}\quad\frac{\phi : S\to S' \text{ is injective}\quad \forall i,\mathcal{P}'(i)=\mathcal{P}(\phi(i))\qquad \{\phi[P]\}\ [\phi[S]:\ \mathcal{P}']\ \{\phi[Q]\}}{\{P\}\ [S:\ \mathcal{P}]\ \{Q\}}$$

Fig. 7. Domain substitution rule

We apply Subst to (13), taking S to be the index set of (13), and ϕ to be a function removing the second element i from all tuples from the second component of S. Note that this ϕ is going to injective as i is fixed across all the programs in the second component. The obtained triple is:

$$\left\{Arrs(\cdot)\right\}\left[\begin{array}{ll}1 & :\texttt{spvspv}(\texttt{m}_{\texttt{ind}}[\texttt{i}],\texttt{m}_{\texttt{val}}[\texttt{i}],\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}})\\ \langle 2,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{m}_{\texttt{ind}}[\texttt{i}],\texttt{m}_{\texttt{val}}[\texttt{i}],j)\\ \langle 3,j\rangle_{j\in[0,N)} & :\texttt{sv\_get}(\texttt{x}_{\texttt{ind}},\texttt{x}_{\texttt{val}},j)\end{array}\right]\left\{\begin{array}{l|l}s & \lceil s=\sum_j \overline{m}(j)\overline{x}(j)\rceil\\ \overline{m} & \\ \overline{x} & \end{array}\right\} \tag{15}$$

Now, substituting $\texttt{m}_{\texttt{ind}}[\texttt{i}]$ for y in (14), we get a hyper-triple that immediately implies (15).

2.5 A Rule for While-Loops

What is left now is to verify correctness of spvspv stated as a hyper-triple (14). Below, we sketch its proof, primarily focusing on how to deal with the **while**-loop in the body of spvspv.

$$\textsc{While}\quad \frac{S = \bigcup_{i=0}^{N-1} S_i \quad \texttt{local}(R_i, S_i) \qquad \begin{array}{l} \forall \overline{x}, \left\{\ I(N,\overline{x})\ \right\}\ [\ \iota : c\ ]\ \left\{\ b\ \middle|\ \lceil b = \mathit{false}\rceil\ \right\} \\ \forall \overline{x}, i < N, \left\{\ I(i,\overline{x})\ *\ R_i\ \right\}\ [\ \iota : \texttt{if}\ c\ \texttt{then}\ p,\ S_i : \mathcal{P}_i]\ \left\{\ z, \overline{y}\ \middle|\ I(i+1, \overline{xy})\ \right\} \end{array}}{\left\{\ I(0,\overline{\emptyset})\ *\ \circledast_{i=0}^{N-1} R_i\ \right\}\ [\ \iota : \texttt{while}\ (c)\ \{p\},\ S : \mathcal{P}]\ \left\{\ z, \overline{y}\ \middle|\ I(N, \overline{y})\ \right\}}$$

Fig. 9. A rule for `while`-iteration.

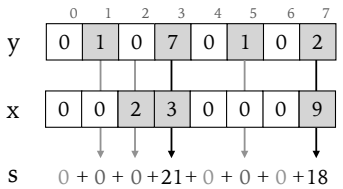


Fig. 8. SpVSpV pictographically

The proof is enabled by an important fact: the **`while`**-loop in spvspv processes *only non-zero* values of input vectors. An illustration of how spvspv works is depicted in Fig. 8. Given the vectors y and x, the algorithm visits all positions where at least one vector has a non-zero value (1, 2, 3, 5, and 7 in this example). For each such position, spvspv will increase the value of s only if *both* positions are non-zero (such situations are depicted with solid black lines in Fig. 8). Getting back to (14), this suggests us to align each iteration of the **`while`**-loop with the corresponding element from the sequence of non-zero indexes in payloads of either of the two vectors $y_{ind} \cup x_{ind}$. For all other indices j, results of both `sv_get` functions, $\overline{y}(j)$ and $\overline{x}(j)$ will be zero, hence they can be safely elided from the summation $\sum_j \overline{y}(j)\overline{x}(j)$. This can be captured by the following triple:

$$\left\{\begin{matrix} Arrs_{\mathtt{y}}(\cdot)\ * \\ Arrs_{\mathtt{x}}(\cdot) \end{matrix}\right\}\ \left[\begin{matrix} \langle 2, j\rangle_{j \notin y_{ind} \cup x_{ind}} : \mathtt{sv\_get}(\mathtt{y_{ind}}, \mathtt{y_{val}}, j) \\ \langle 3, j\rangle_{j \notin y_{ind} \cup x_{ind}} : \mathtt{sv\_get}(\mathtt{x_{ind}}, \mathtt{x_{val}}, j) \end{matrix}\right]\ \left\{\begin{matrix} \overline{y} \\ \overline{x} \end{matrix}\ \middle|\ \begin{matrix} \lceil \forall j \notin y_{ind} \cup x_{ind}, \overline{y}(j) = 0\rceil\ * \\ \lceil \forall j \notin y_{ind} \cup x_{ind}, \overline{x}(j) = 0\rceil \end{matrix}\right\} \tag{16}$$

The hyper-triple (16) can proven in a way similar to the specification (7) of `ucsr_get`. First, using the Product rule we reduce it to a set of unary triples, and then, noticing that the `index` function in the body of `sv_get` will return `-1` for any j that is not an element of the set $x_{ind} \cup y_{ind}$. Now applying the Focus rule, and taking $ind \triangleq y_{ind} \cup x_{ind}$, we reduce (14) to the following goal:

$$\left\{\begin{matrix} Arrs_{\mathtt{y}}(\cdot)\ * \\ Arrs_{\mathtt{x}}(\cdot) \end{matrix}\right\}\ \left[\begin{matrix} 1 : \mathtt{spvspv}(\mathtt{y_{ind}}, \mathtt{y_{val}}, \mathtt{x_{ind}}, \mathtt{x_{val}}) \\ \langle 2, j\rangle_{j \in ind} : \mathtt{sv\_get}(\mathtt{y_{ind}}, \mathtt{y_{val}}, j) \\ \langle 3, j\rangle_{j \in ind} : \mathtt{sv\_get}(\mathtt{x_{ind}}, \mathtt{x_{val}}, j) \end{matrix}\right]\ \left\{\begin{matrix} s \\ \overline{y} \\ \overline{x} \end{matrix}\ \middle|\ \lceil s = \sum_{j \in ind} \overline{y}(j)\overline{x}(j)\rceil\right\} \tag{17}$$

Finally, we are ready to align each iteration of the **`while`**-loop with the corresponding element of ind by applying the While rule shown on Fig. 9. While is (predictably) similar to the For rule. The intuition for this rule comes from the fact that each loop `while` (c) $\{p\}$ that terminates after no more than N steps is equivalent to just N iterations of `if` c `then` p. Therefore, if we divide our index set into N batches, we can align each batch with one execution of `if` c `then` p. Note that this will also handle the case when the loop terminates after $K < N$ steps: in that case all runs of `if` c `then` p will become idle, starting from the K^{th} step. A new premise on the top right corner will ensure that the loop will terminate once we have exhausted all batches.[6] To proceed with the proof of (17) by making use of the While rule, we divide our index set as follows:

$$\langle 2, j\rangle_{j \in ind} \cup \langle 3, j\rangle_{j \in ind} = \bigcup_{j=0}^{|ind|-1} \{\langle 2, ind[j]\rangle, \langle 3, ind[j]\rangle\}$$

[6] LGTM is implemented as logic for *total* correctness, yet the While rule does not require an explicit termination measure. The termination will follow from the fact that, each iteration is aligned with its own batch from a finite set of index batches.

Locations	$x,\ y,\ \ldots$	alpha-numeric strings
Constants	n	unbounded integer literals
Expressions	$e ::=$	$n \mid \text{true} \mid \text{false} \mid x \mid e = e \mid e < e \mid e \wedge e \mid \neg e \mid e \oplus e$
Command	$c ::=$	$!x \mid x[d] \mid x := e \mid x[e] := e \mid \texttt{for}\ x\ \texttt{in}\ \texttt{range}(e,e)\ \{c\} \mid \texttt{while}\ (e)\ c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\{c\} \mid$ $\texttt{let}\ x := c\ \texttt{in}\ c \mid \texttt{alloc}(n) \mid \texttt{malloc}(n) \mid \texttt{free}(x) \mid \texttt{mfree}(x) \mid \texttt{length}(e) \mid \texttt{return}\ e$

Fig. 10. LGTM programming language

Here, to reference to the j^{th} component of *ind* we treat as an sorted sequence without duplicates. Let us show the most important part of the loop invariant and sketch the rest of the prove:

$$\begin{aligned} I_{\texttt{while}}(j, \overline{yx}) \triangleq \exists i_y, i_x,\ &\texttt{iY} \mapsto i_y\ *\ \texttt{iX} \mapsto i_x\ * \\ &s \mapsto \textstyle\sum_{k<j} \overline{y}(ind[k])\,\overline{x}(ind[k])\ * \\ &\lceil \min(x_{ind}[i_x], y_{ind}[i_y]) = ind[j] \rceil\ *\ \ldots \end{aligned} \tag{18}$$

The first line of the invariant simply gives names to the values stored in the vector counters (i_y and i_x). The second line captures the fact that s stores the accumulated intermediate result of the dot product. The third line is the most subtle: it states that at each iteration, the current position in the *ind* sequence, is equal to a *minimum* of the current positions in both vectors ($y_{ind}[i_y]$ and $x_{ind}[i_x]$). If $y_{ind}[i_y]$ and $x_{ind}[i_x]$ are same, then they are both equal to $ind[k]$. Hence line 6 of spvspv will increase the sum by $\overline{y}(ind[j])\,\overline{x}(ind[j])$, preserving the second line of $I_{\texttt{while}}$. In other cases, using the third line of the invariant, we will be able to show one of $y_{ind}[i_y]$ or $x_{ind}[i_x]$ to be zero, so the sum will indeed remain unchanged, matching the code on lines 8-9 and 10-11 of Fig. 3.

2.6 Putting It All Together

And that concludes our tour of LGTM! As a quick recap, the right side of Fig. 13 summarises the main milestones of our proof of spmspv's specification (3) (with pre-/posts elided), showing the corresponding sections and changes in the proof goals. Once again, we emphasise the key LGTM aspects that enabled the proof: *separation* for modular reasoning (Sec. 2.1.2), *parametrisation* for deconstructing loops (Sec. 2.3 and 2.5), and *compositionality* for proof reuse (Sec. 2.4). We next briefly outline LGTM's soundness argument, postponing the presentation of its implementation as a verification tool till Sec. 4, in which we will show the Coq mechanisation of our spmspv example.

3 LGTM, FORMALLY

The programming language of LGTM (Fig. 10) is chosen to be generic enough to model tensor computations in C, Python, and Julia. Its commands include location and array access ($!x$ and $x[e]$), allocation/deallocation for both individual memory locations (via alloc/free) and arrays (via malloc/mfree). The let-expressions are also used to chain sequences of commands, thus encoding a sequential composition. For the sake of the presentation, we assume that all the considered programs are well-typed, integers are unbounded, and division is not present amongst the operations.

3.1 Semantics

We adopt a standard big-step semantics for our language. Let c be a command and s be a *state*, *i.e.*, a finite mapping from locations Loc to values Val. The evaluation relation $\langle c, s\rangle \Downarrow \langle v, s'\rangle$ is defined inductively on the structure of c. It means that a command c starting from input store s terminates with the return value $v \in \text{Val}$, and store s'.[7] In this work, we only focus on deterministic program executions, *i.e.*, such that allocation does not exercise randomness. For a fixed index set S, given two *hyper-heaps* h and h' defined as mappings from $\text{Loc} \times S$ to values, a hyper-value $\overline{v}$ and an

[7]We only consider terminating programs, and require termination measure given for programs with **while**-loops.

S-indexed program product $\mathcal{P}$, the semantics of a product execution is defined as follows:

$$\langle \mathcal{P}, h\rangle \Downarrow_S \langle \overline{v}, h'\rangle \triangleq \forall \iota \in S, \langle \mathcal{P}(\iota), h|_\iota\rangle \Downarrow \langle \overline{v}(\iota), h'|_\iota\rangle$$

That is, for every $\iota \in S$ and individual program $\mathcal{P}(\iota)$ executed on the corresponding projection $h|_\iota$ of the initial hyper-heap (defined as $h|_\iota \triangleq \lambda l \in \mathsf{Loc}.\ h(l,\iota)$), its execution results in a new state $h'|_\iota$ and a value $\overline{v}(\iota)$. Similarly to the ordinary (unary) programs, we assume all programs in a product to be deterministic and terminate (the latter is guaranteed for all programs that verify in LGTM). The details of the semantics can be found in our Coq mechanisation (Gladshtein et al. 2024a).

Hyper-safety triples in LGTM are defined in terms of the weakest precondition predicate:

$$\mathsf{wp}\ [S :\ \mathcal{P}]\ \{Q\} \triangleq \lambda h,\ \exists h'\ \overline{v},\ \langle \mathcal{P}, h\rangle \Downarrow_S \langle \overline{v}, h'\rangle \wedge Q(\overline{v})(h') \tag{19}$$

The weakest preconditions are relative to a provided postcondition (*i.e.*, a predicate on hyper-heaps) Q and are defined as predicates on input heaps h, where Q is a function from hyper-value to hyper-heap assertions. We next define a hyper-triple $\{P\}\ [S : \mathcal{P}]\ \{Q\}$ as a notation for $P \vdash \mathsf{wp}\ [S :\ \mathcal{P}]\ \{Q\}$. In plain language, this means that an assertion $P(h)$ implies $\mathsf{wp}\ [S :\ \mathcal{P}]\ \{Q\}(h)$ for any hyper-heap h. We will use the expanded version of the postcondition whenever we want to emphasise its assertion Q to be a function of the result hyper-value: $\{H\}\ [S : \mathcal{P}]\ \{\overline{v}\ \mid Q(\overline{v})\}$ and we will omit the index set S whenever it is clear from the context. Note that, due to the termination requirement on the programs in products, all triples in LGTM ensure total correctness by definition.

3.2 Support for Reasoning with Parametrisation and Separation

Many essential rules of LGTM take their origins in the Logic for Hyper-triple Composition (LHC) by D'Osualdo et al. (2022). LGTM's *structural* and *lockstep* rules directly adjust those of LHC to work with parametrised triples that use Separation Logic (SL) connectives and predicates. Below, we outline the key LGTM's enhancements on top of LHC, supporting parametrisation and separation.

3.2.1 Parametrisation. One example of an LGTM rule that is not derivable in LHC is the Product rule (Fig. 4). Product takes inspiration from LHC WpConj rule, depicted on Fig. 11. LHC triples do not feature the index set explicitly, as all programs in LHC triples can only be indexed with natural numbers. This rule allows one to conjoin specifications of two program products $\mathcal{P}_1$ and $\mathcal{P}_2$ with possibly overlapping components, yielding $\mathcal{P}_1 + \mathcal{P}_2$, if $\mathcal{P}_1$ and $\mathcal{P}_2$ agree on the overlapping part. Even though LHC's rules are phrased in terms of triples with an arbitrary number of programs, this number must be *fixed* prior the proof. This is not a problem in LHC: for program products of a fixed known size n, one would have to apply this rule n times.

$$\frac{\mathrm{idx}(Q_1) \subseteq \mathrm{supp}(t_1)\quad \mathrm{idx}(Q_2) \subseteq \mathrm{supp}(t_2)}{\mathsf{wp}\ \mathcal{P}_1\{Q_1\} \wedge \mathsf{wp}\ \mathcal{P}_2\{Q_2\}}\ \textsc{Wp-Conj}$$
$$\mathsf{wp}\ (\mathcal{P}_1 + \mathcal{P}_2)\ \{Q_1 \wedge Q_2\}$$

Fig. 11. WpConj rule from LHC

Things become problematic with parametrised triples: to get the effect of LGTM's Product rule, one would have to apply Wp-Conj *the size of* S number of times. To support cases when $|S|$ depends, *e.g.*, on a program parameter, this would require LHC support a "constant-space" iterated rule application (as in $\mathcal{R}^n()$ instead of $\mathcal{R}(\mathcal{R}(\ldots))$ - repeated explicitly n times) of Wp-Conj. This ability does, indeed, come "for free" for an embedding into a higher-order logic such as one of Coq (hence the importance of Coq formalisation in our work). It is, however, missing from LHC, but is explicitly captured in LGTM rules. The same idea applies to For and While rules (Sec. 2.3–2.5). In both those rules we divide the index set S into n batches, where n is a parameter coming from the specification, so that one iteration of the `for`-/`while`-loop handles only one such batch.

3.2.2 Separation. LGTM extends LHC's *lockstep* inference rules with Separation Logic specific-rules (*e.g.*, for pointer and array allocation). Adding required adjusting a unary

$$\{\mathtt{emp}\}\ [S : \mathtt{alloc}(v)]\ \left\{\overline{x}\ \middle|\ \mathop{\Large *}_{i\in S} \overline{x}(i) \mapsto v\right\}$$

Fig. 12. LGTM rule for allocation

SL proof system to work with hyper-heaps, as, for instance, is exemplified by Alloc rule presented on Fig. 12. Alloc states that the results of a family of pointer allocations are these pointers $\overline{x}$, pointing to the assigned value. Other LGTM lockstep rules follow the same intuition and are available in the appendix of the extended version of the paper (Gladshtein et al. 2024b).

3.3 Soundness and Mechanisation

We say that LGTM triple $\{P\}\ [S : \mathcal{P}]\ \{Q\}$ is *derivable* and write $\vdash_{\text{LGTM}} \{P\}\ [S : \mathcal{P}]\ \{Q\}$ if this triple can be obtained using the rules of the logic. We say that the triple $\{P\}\ [S : \mathcal{P}]\ \{Q\}$ is *semantically valid* and write $\models \{P\}\ [S : \mathcal{P}]\ \{Q\}$ if and only if

$$\forall h, P(h) \Rightarrow \exists h'\ \overline{v}, \langle \mathcal{P}, h\rangle \Downarrow_S \langle h', \overline{v}\rangle \wedge Q(\overline{v})(h')$$

That is, we ask that for every hyper-heap h satisfying the precondition, if we run $\mathcal{P}$ on it, it terminates, and the resulting hyper-heap and hyper-value satisfy Q. The derivability of a logic triple in LGTM is connected to its semantic validity by the following standard soundness result:

Theorem 3.1 (Soundness). *For all hyper-heap assertions P, functions from hyper-value to hyper-heap assertions Q, and program products $\mathcal{P}$ indexed by a set S,*

$$\vdash_{\text{LGTM}} \{P\}\ [S : \mathcal{P}]\ \{Q\} \Rightarrow \models \{P\}\ [S : \mathcal{P}]\ \{Q\}$$

We have mechanised the meta-theory of LGTM and its semantics in the logic of Coq proof assistant. In our mechanisation, we defined the Hoare triples semantically in terms of the weakest precondition predicates, which, by their definition (19) encode semantic validity. We then defined each LGTM rule as a lemma and proved that it holds, thus establishing the result of Theorem 3.1. Our mechanisation of LGTM is built by extending CFML (Charguéraud 2011, 2020)—a mechanised sequential Separation Logic. Our changes to the original CFML including the theory of hyper-heaps, the wp-calculus for program products, LGTM rules, and automation, are totalling at 20kLOC.

4 LGTM AS A VERIFICATION TOOL

Let us now showcase LGTM as a verification framework by walking through a mechanisation of the `spmspv` case study from Sec. 2 and explaining its most essential proof automation components.

4.1 Structuring Mechanised Proofs

Fig. 13 presents a mechanised proof of `spmspv` (left) next to the informal outline of the proof derivation from Sec. 2 (right), with gray rectangles depicting the transformations of the proof goals.

One noticeable technical difference between the two proofs is the representation of index sets in the mechanisation. In the Coq specification (lines 1-7), each component of the program product has an explicit *tag*, separate from the rest of the index set definition. For example, the tag of `uscr_get` is `2`. We tried several designs for LGTM triples and found the "tagged union" of index sets to be the most user-friendly when it comes to the local per-component reasoning via focusing (Sec. 2.2).[8]

The proof immediately makes use of the index set tags: the `xin` *tactical* (*i.e.*, a higher-order tactic) at line 9 takes the tag `1` as its argument, thus applying a version of the SeqU1 rule (Sec. 2.1.1). Using `xin`, one can advance the proof of a composite hyper-triple by symbolically executing instructions in its sub-components in a lockstep manner by passing a sequence of tactics implementing sequential (or, in general, lockstep) rules after the `:` separator. In this case, we pass two tactics to the `xin` tactical. The first one, `xmalloc`, symbolically executes the `malloc` instruction in the first component. The second, `xret`, strips off the last `return` statement in the body of `spmspv`.

[8]The tag-based encoding does not affect expressivity, since index sets can be arbitrarily mapped using Subst (Fig. 7).

```
1  Lemma spec_spmspv (m_idx m_ind m_val x_ind x_val : loc) :
2  {{ Arrs }} [{
3    [1| _  ∈ {0}            ⇒ spmspv m_idx m_ind m_val x_ind x_val];
4    [2| ij ∈ [0,M) x [0,N) ⇒ uscr_get m_idx m_ind m_val ij.1 ij.2];
5    [3| j  ∈ [0,N)          ⇒ sv_get x_ind x_val j];
6  }] {{ a m x, arr(a, fun i ⇒ \sum_(j ∈ [0, N))m(i,j) * x(j)) \* Arrs }}
7  Proof. xstart. (* proof start *)
8    (* § 2.1.1 *)
9    xin 1 : xmalloc p⇒ //; xret.
10   (* § 2.2 *)
11   xfocus 2 (indom (m_idx x [0, N) )).
12   (* § 2.1.2 *)
13   xapp spec_ucsr_get.
14   - case⇒ ??; indomE; tauto.
15   xclean_unused. (* clean up unused assertions *)
16   (* frame out unused part of ans array *)
17   xframe_array p m_idx.
18   - move⇒ *; erewrite <-Sum0s at 1; apply/SumEq⇒ ?.
19     by indomE⇒ /=; case_if.
20   (* partition index set into Si to apply For-rule *)
21   have E: [0,M)x[0,N) ∩ m_idx x[0,N) = m_idx x[0,N)
22   - rewrite prod_intr_list; f_equal; exact/intr_list.
23   rewrite E fset_of_list_nodup prod_Union_distr.
24   rewrite -(Union_same |m_idx| [0, N)).
25   (* § 2.3 *)
26   xfor with Inv := I_for.
27   - xin 2: xapp spec_index; xstep; xif⇒ [?|]; xgo.
28     xin 1: xgo.
29     rewrite index_nodup //; try math. (* simplify goal *)
30     (* § 2.4 *)
31     xsubst (fun (2,i,j) ⇒ (2,j)).
32     (* § 2.5 *)
33     xapp spec_spvspv.
34     xsimpl⇒->; xstep; xsimpl*. (* store result of spvspv *)
35   (* solve arithmetic to adjust big summations in precondition *)
42 Qed.
```

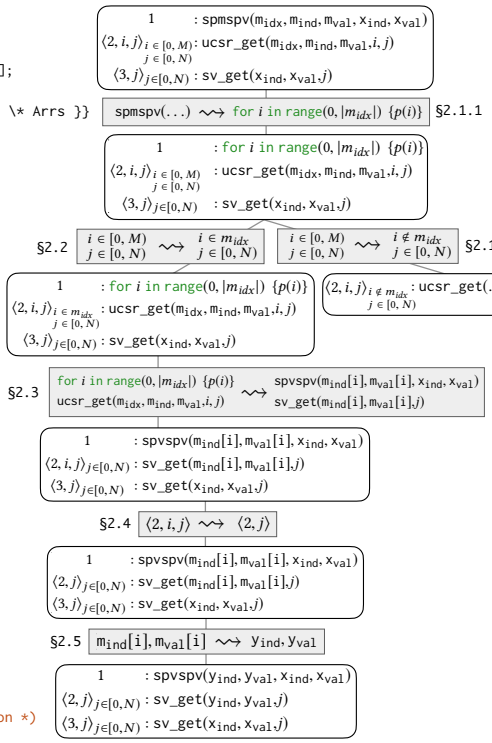


Fig. 13. Mechanised LGTM proof of spmspv (left) and the corresponding paper-and-pencil derivation (right).

Moving on with the proof, we achieve the effect of the Focus rule from Sec. 2.2 by calling `xfocus` tactic at line 11. As the result, the index set of the second component will be divided into two groups, as per the tactic's second argument: inside $m_{idx} \times [0, N)$ and outside of it. Subsequently, the lines 12-13 verify the runs of `uscr_get` outside of $m_{idx} \times [0, N)$, using the specification of `uscr_get` via `xapp` tactic. In the rest of the proof, we first divide the index set into a union of $|m_{idx}|$ batches, to apply the For rule (lines 21-24). Next, at line 26 we apply the `xfor` tactic, replicating the steps from Sec. 2.3, followed by an application of the `xsubst`, which performs the domain substitution from Sec. 2.4. We conclude the proof by using an independently verified specification of spvspv (line 33), discharging the residual obligations about index set manipulations (lines 34-42).

4.2 Proof Automation for Separation and Parametrisation

The two enabling features of LGTM, modularity via separation and parametrisation, come with additional obligations in rules that exercise them. Those obligations have to do with (a) proving *assertion locality* (Sec. 2.1.2) when reducing triple arity and (b) *changing indices within assertions* when performing domain substitution with arbitrary index sets (Sec. 2.4). Below, we describe two proof automation techniques that eliminate the proof burden associated with those obligations.

4.2.1 Discharging Locality Obligations. Tactics that generate proof obligations for locality of hyper-heap assertions include `xprod` (which implements the Product rule) and `xfor` (which implements For). For instance, the application of `xfor` at line 26 of Fig. 13 generates a number of subgoals requiring one to establish locality of the assertions R_i used as preconditions for programs in the respective components (*cf.* Fig. 6). Even a subgoal for the third component looks a bit intimidating:

$$\texttt{local}\left(\left(\mathop{\Large\ast}_{i=0}^{N-1} \texttt{arr}(\texttt{x}_{\texttt{ind}}\,\langle 3,i\rangle,\ x_{ind}) \ \ast\ \mathop{\Large\ast}_{i=0}^{N-1} \texttt{arr}(\texttt{x}_{\texttt{val}}\,\langle 3,i\rangle,\ x_{val})\right),\ \{\langle 3,i\rangle : i \in [0,N)\}\right)$$

To dispatch such goals, we provide the `xlocal` tactic shown Fig. 15. In a nutshell, this tactic, structurally decomposes the hyper-heap assertion in a goal, reducing locality of a composite assertion to locality of its parts. For instance, lines 3-4 of the tactic reduce locality of $P_1 \ast P_2$ to locality of both P_1 and P_2, calling `xlocal` recursively on those subgoals. Lines 5-6 we reduce locality of $\mathop{\Large\ast}_{i\in S} H_i$ to the locality of P_i for each $i \in S$. Finally, whenever `xlocal` reaches terminal goals such as $\texttt{local}(\texttt{arr}(p(i), l), S)$, it applies the `local_array` lemma, resulting in the obligation $i \in S$, proven by the `indomE` membership solver from the MathComp library (Mahboubi and Tassi 2022).

```
1  Ltac xlocal := intros;
2    match goal with
3    | |- local (P1 \* P2) _ ⇒
4         apply local_conj; xlocal
5    | |- local (\*_(i ∈ s) Pi) _ ⇒
6         apply local_big_conj; xlocal
7    | |- local (arr(p(i), l)) s ⇒
8         apply local_array; indomE
9    ...
10   end
```

Fig. 14. `xlocal` tactic

4.2.2 Automating Index Set Substitution. The need for proof automation in the presence of parametrisation manifests whenever we want to perform reindexing involving arbitrary index sets via the Subst rule. As Subst applies a substitution function ϕ to both pre- and postcondition, ϕ must be distributed over the connectives of the LGTM including, separation conjunction and its iterated version (big-$\ast$). One naïve approach to implement such distribution automatically would involve proving a lemma of the form $\phi\left[\mathop{\Large\ast}_{i\in S} H_i\right] = \mathop{\Large\ast}_{i\in S} \phi\,[H_i]$ for distributing the substitution over $\ast$, and use it for repeated rewriting in the assertions. Unfortunately, this solution does not combine well with the shallow embedding of "big" operators into Coq, that implement indexing over i via Coq's lambda-functions. As the inner assertions H_i of $\mathop{\Large\ast}_{i\in S} H_i$ might depend on the lambda-bound i, rewriting them would require independently instantiating the equality lemmas such as $\forall P, Q.\ \phi[P \ast Q] = \phi[P] \ast \phi[Q]$ with assertions that depend on i and are not well-formed outside of the lambda. Therefore, after we rewrite an assertion involving $\ast$ using the above equality once, the application of ϕ will occur in the scope of the binder i, preventing its further rewrites.

We overcome this hurdle with the following trick. When performing domain substitution via ϕ, we start with replacing the precondition ϕ`[P]` (*resp.* ϕ`[Q]`) with a fresh existential variable `?H`, adding a subgoal ϕ`[P] = ?H` to the set of proof obligations. This subgoal will be repeatedly transformed as we gradually propagate ϕ inside `P`, elaborating `?H` as we go. This idea is implemented by `xsubst` tactic shown in Fig. 15. To understand its effect on the proof, consider the lines 6-8 of its definition that deal with the big-$\ast$ operator. The tactic implementation first propagates ϕ under the big-star by performing the rewrite via the `subst_big_conj` lemma, followed by application of the `big_conj_eq` lemma of type $(\forall i, H_i = P_i) \Rightarrow \mathop{\Large\ast}_{i\in S} H_i = \mathop{\Large\ast}_{i\in S} P_i$, which is essentially a specialised version of the functional extensionality axiom. Applying this lemma instantiates the existential variable `?H` with $\mathop{\Large\ast}_{i=0}$(`?Hi i`), producing a subgoal $\forall i, (\phi[P_i] =$ `?Hi i`), which will require further elaboration of `?Hi`. The tactic subsequently fixes the binder i by moving it to the proof context using `intros`, followed by a recursive call of `xsubst`.

```
1  Ltac xsubst := match goal with
2    | |- φ[P1 \* P2] = ?H ⇒
3         rewrite subst_conj;
4         apply conj_eq; xsubst
5    | |- φ[\*_(i ∈ s) Pi] = ?H ⇒
6         rewrite subst_big_conj;
7         apply big_conj_eq; intros; xsubst
8    | |- φ[arr(p(i), l)] = ?H ⇒
9         rewrite subst_array; reflexivity
10   ...
11   end
```

Fig. 15. `xsubst` tactic

When `xsubst` reaches primitive heap assertions such as $\texttt{arr}(p(i), l)$, it rewrites them using suitable lemmas (*e.g.*, turning $\texttt{arr}(p(i), l)$ into $\texttt{arr}(p(\phi(i)), l)$ via `subst_array`), dispatching the residual goal by using Coq's standard `reflexivity` tactic that instantiates the remaining existential variable.

Table 1. Statistics for the verified programs. For each program, we specify its operation in Einstein notation, the formats of the involved operands (D is for *dense*), and the return type, followed by the size of the code and of the mechanised LGTM proof, along with the proof/code ratio. We also indicate the characteristic rules of LGTM the auxiliary specifications used by the proof, as well the time it takes Coq to check it.

#	Operation	Formats (Operands)	Return Type	Size (LOC)		Ratio	Prominent rules				Uses	Time (sec)
				Code	Proof		For	While	Focus	Subst		
1	$\sum_i x_i$	SV $(x)^{1,2}$	`int`	9	22	2.4	✓		✓			5.42
2	$\sum_i x_i y_i$	SV (x), D $(y)^{1,2}$	`int`	12	22	1.8	✓		✓			6.96
3	$\sum_i x_i y_i$	SV (x), SV $(y)^{1,2}$	`int`	31	237	7.6		✓	✓			20.73
4	$\sum_{i,j} A_{i,j}$	COO (A)	`int`	7	25	3.6	✓		✓			5.40
5	$\sum_{i,j} A_{i,j}$	CSR $(A)^{1,2}$	`int`	10	11	1.1	✓			✓	#1	7.91
6	$\sum_j A_{i,j} x_j$	CSR (A), D $(x)^{1,2}$	`int[]`	10	12	1.2	✓			✓	#2	8.99
7	$\sum_j A_{i,j} x_j$	CSR (A), SV $(x)^{1,2}$	`int[]`	10	16	1.6	✓			✓	#3	11.02
8	$\sum_{i,j} A_{i,j}$	uCSR (A)	`int`	10	29	2.9	✓		✓	✓	#1	12.62
9	$\sum_j A_{i,j} x_j$	uCSR (A), D $(x)^3$	`int[]`	11	33	3.0	✓		✓	✓	#2	18.69
10	$\sum_j A_{i,j} x_j$	uCSR (A), SV $(x)^3$	`int[]`	11	30	2.7	✓		✓	✓	#3	20.05
11	$\sum_i x_i$	RL $(x)^2$	`int`	16	18	1.1	✓					5.90
12	$\sum_i \alpha x_i + \beta y_i$	RL (x), RL $(y)^2$	`int`	64	143	2.2		✓				41.66
13	$\sum_j A_{i,j} x_j$	CSR (A), D $(x)^4$	`double[]`	19	36	1.9	✓		✓	✓		31.76

[1] From (Kjolstad et al. 2017) [2] From (Ahrens et al. 2023) [3] From (Hong et al. 2019) [4] From (Kellison et al. 2023)

5 LGTM UNDER THE SPOTLIGHT: EMPIRICAL EVALUATION

We conducted an empirical evaluation of LGTM to answer the following research questions:

- **RQ1**: Is LGTM expressive enough to reason about real-world computations on structured data?
- **RQ2**: How LGTM proofs are influenced by the involved formats and kinds of computations?
- **RQ3**: How does LGTM fare against state-of-the-art approaches not based on relational logic?

Benchmarks. We have assembled our benchmark suite by adopting case studies from different sources, including programs produced by the TACO (Kjolstad et al. 2017) and Finch (Ahrens et al. 2023) compilers, notable benchmarks from the Sparse Suite collection (Hong et al. 2019), and an example from a recent effort on verifying sparse matrix/vector multiplication (Kellison et al. 2023). The conversion of the selected programs from their implementation languages, C and Julia, to the language of LGTM (Fig. 10) has been done manually, but could be easily automated.

The resulting collection of 13 programs is summarised in Tab. 1. The programs were selected to feature diverse range of data formats and operations. The considered formats can be broadly classified into three groups: (1) *primitive* sparse tensor formats, such by SV and Coordinate List (COO) (Popoola et al. 2023), (2) *composite* sparse tensor data formats, CSR and uCSR, that incorporate primitive formats as their parts, and (3) a data format for non-sparse structured data, represented by Run-Length Encoding (RL) (Donenfeld et al. 2022). For each such format group, we considered the most common operations with the respective data implemented in real-world kernels: dot-product and alpha-blending for vector formats, and matrix/vector multiplication for matrix formats. All the verified programs listed in Tab. 1 are parametrised with arbitrary sizes of matrices/vectors.

5.1 RQ1: Expressivity of the Logic

We answer RQ1 by providing quantitative evidence of LGTM's effectiveness and efficiency when verifying characteristic programs manipulating with structured data listed in Tab. 1.

A quick glance at the table's fifth and sixth column should convey that LGTM's proofs are relatively compact, especially for an extrinsic verifier, such as Coq, with the average proof/code ratio (in LOC) being 2.5. One notable outlier in this aspect is a dot-product of two sparse vectors (#3),

whose proof we will discuss in Sec. 5.2. The last column of Tab. 1 reports the proof checking times (averaging at 15.2s), obtained via Coq 8.18 on a 3.2 GHz Apple M1 MacBook Air with 16GB RAM. The time it takes Coq to check LGTM proofs is directly influenced by the number of uses of tactics that implement domain-specific automation, such as `xsubst` and `xlocal` (*cf.* Sec. 4.2.2).

5.2 RQ2: Verification Trends

Besides quantitative characteristics of LGTM, such as proof sizes and proof checking time, we are also interested in *verification trends*. For instance, what rules one should expect to use when dealing with a particular format or a kind of structured data computation?

To answer this question, we start by considering the programs #1-#4 involving primitive formats SV and COO for sparse vectors and matrices, which are used as "building blocks" by other formats The verification trends for these programs are summarised in the seventh column of Tab. 1, featuring (1) the Focus rule, required to focus *only on non-zero elements* of the input tensor and (2) the For or While rules, depending on the type of the loop in the code, required to group remaining non-zero elements with correspondent loop iterations. Next, we focus on the case studies #5-#10 with composite formats, CSR and uCSR, which are obtained as combinations of a dense representation, SV (for exploiting sparsity), and COO (for featuring random permutation of the components). The key proof principle of LGTM that shows up prominently in those proofs is Subst. This should not come as a surprise: because of the composite nature of the formats, one should expect to make use of the specifications that involve their sub-formats (*e.g.*, SV is a part of CSR). Using Subst allows one to adjust intermediate subgoals within the proof to match specifications of "auxiliary" operations (most of which are represented by case studies #1-#4) so they could be used in the proof in a compositional manner. As the penultimate column of Tab. 1 indicates, the case studies #5-#10 that involve composite data formats, reuse a proof of some case study #1-#3, involving only primitive formats. Finally, we consider the case studies #11-#12, in which we have verified operations on RL encoding that manipulate with compressed arrays containing repeated data. In those proofs, we made use only of LGTM's loops rules, as the RL format is primitive (so we don't need Subst) and is not sparse (so we don't need to focus on non-zero elements via Focus).

We conclude this subsection with an observation about the only example from our collection that has a relatively large proof/code ratio (7.6): a dot-product of two SV vectors (#3). This figure can be explained by the fact that the textbook implementation of SV/SV product is heavily optimised by *skipping* as many unnecessary computations (involving zeros) as possible. Hence the main complexity in the proof is because of the need to justify the *absence* of code, rather than its *presence*.

5.3 RQ3: Comparison with Non-Relational Proofs

The landscape of tools for verified computations on structured data is somewhat scarce, with the prior work mostly focusing on verified compilers for sparse tensors (Arnold et al. 2010; Dyer et al. 2019; Kovach et al. 2023). Those tools, which we discuss in detail in Sec. 6, can only produce a subset of computations that can be verified in LGTM, and they are not guaranteed to produce the most performant format-specific code. The various existing logics for k-safety either do not offer means to work with array computations (Dardinier and Müller 2024; Sousa and Dillig 2016), or are not even mechanised (D'Osualdo et al. 2022), thus making it impossible to conduct a meaningful comparison with them as tools. Therefore, we only compare verification in LGTM with formal reasoning about structured data in a mechanises general-purpose non-relational Separation Logic.

Recent work by Kellison et al. (2023) presented a mechanised proof of a sparse matrix/dense vector product function implemented in C, taking a sparse matrix in the CSR format. The C implementation is specified and verified using the Verified Software Toolchain (VST) (Appel 2011), a Separation Logic framework embedded into Coq. Kellison et al.'s proofs follows a relatively conventional

```
Inductive crs_row_rep {t} : ∀ cols (vals:list
(ftype t)) col_ind (v:list (ftype t)), Prop :=
| crs_row_rep_nil: crs_row_rep 0%Z nil nil nil
| crs_row_rep_zero: ∀ cols vals col_ind v,
  crs_row_rep (cols-1) vals (map pred col_ind) v ->
  crs_row_rep cols vals col_ind (Zconst t 0 :: v)
| crs_row_rep_val: ∀ cols x vals col_ind v,
  crs_row_rep (cols-1) vals (map pred col_ind) v ->
  crs_row_rep cols (x::vals) (0::col_ind) (x::v).
```

(a) Inductive predicate for relating a vector v with length cols and its corresponding sparse vector represented by col_ind and vals.

```
Definition crs_rep_aux {t} (m_val: matrix t) cols
(vals: list (ftype t)) col_ind row_ptr: Prop :=
(* propositions about sortedness and lengths *) ∧
∀ j, 0 <= j < Zlength m_val ->
crs_row_rep cols (sublist (Znth j row_ptr)
  (Znth (j+1) row_ptr) vals)
(sublist (Znth j row_ptr) (Znth (j+1) row_ptr)
  col_ind) (Znth j m_val).
```

(b) Predicate for relating a dense matrix mval modelled as a list and a sparse matrix in CSR, represented by row_ptr, col_ind, and vals.

Fig. 17. Encoding of CSR format in VST by Kellison et al. (2023); ftype t is a type for floating point numbers.

two-layer paradigm for verifying imperative code (Appel 2022): first a *functional* model of sparse matrices is defined, along with its operations (*e.g.*, computing the dot-product of a matrix row and a dense vector); next, the C function is ascribed a specification in terms of a functional model, and the verification amounts to proving that the imperative function *refines* the functional operation.

Fig. 16 presents a slightly simplified CSR sparse matrix/dense vector product implementation verified in Kellison et al.'s LAProof framework. The outer loop iterates through the rows of a compressed CSR matrix m and the inner loop sums up all non-zero m elements in each row, multiplied by the respective values of a dense vector v. The functional model of CSR is defined using a family of inductive predicates. First, the inductive predicate crs_row_rep (Fig. 17a) relates the contents of a dense vector and its sparse representation, capturing the shape of the SV format. Next, crs_rep_aux (Fig. 17b) uses crs_row_rep to express the relation between a dense matrix and the corresponding CSR matrix. Finally, an SL *representation predicate* crs_rep (omitted for brevity) relates a dense matrix to its sparse counterpart (constrained by crs_rep_aux) stored as a C struct in the memory. The predicate crs_rep is used in the pre/postconditions of the specification, and the postcondition states that the returned vector is exactly the result of multiplying the input dense matrix and a dense vector. The LAproof verification proceeds as follows:

```
def spmv(m_idx, m_ind, m_val, v):
  for i in range(0, length(m_idx)):
    for j in range(m_idx[i], m_idx[i+1]):
      ans[i] += m_val[j] * v[m_ind[j]]
```

Fig. 16. CSR matrix/dense vector product

(1) Prove that the inner loop accumulates into ans[i] all values in the array m_val corresponding to the i^{th} row, multiplied by respective values of v using the invariant

$$I(j) \triangleq \mathsf{ans}[i] = \sum_{k=\mathtt{m}_{\mathtt{idx}}[i]}^{j} \mathtt{m}_{\mathtt{val}}[k] \cdot v[\mathtt{m}_{\mathtt{ind}}[k]]$$

(2) For any i at the end of the loop, replace the sum at the right hand side of the equality in the invariant with the product of the corresponding rows of the logical "dense" matrix m and the vector v. This rewrite requires leveraging the properties of the representation predicate csr_rep, connecting the contents of m_idx, m_ind, and m_val with the elements of the logical matrix m (including zeros), via a set of predicate-specific lemmas.

(3) Verify the outer loop using an invariant stating that at i^{th} iteration, for each $k < i$, $\mathsf{ans}[k] = \mathtt{m}[k] \cdot \mathtt{v}$, where m[k] is an k^{th} row of the logical matrix m.

The most laborious part of this proof is its second step. It requires a whole separate file with 350 LOC of Coq code to prove a number of bespoke facts about csr_rep. Those facts are needed to "align" the representation predicate, which necessarily describes the entire contents of the matrix m, with the logic of the inner loop of the code above, which only operates on m's non-zero values.

In contrast, our approach does not require one to design such tailored functional model for each new sparse format. Instead, with LGTM, one can use access functions (*e.g.*, `sv_get`) for interpreting the formats, relating their results to that of the main function being specified. Those access functions can be executed "on demand" in lockstep with the respective parts of the operation's implementation and verified with a comparatively low proof overhead. This highlights a conceptual advantage of our approach: it liberates the user from the burden associated with (*i*) designing functional format definitions and (*ii*) proving their properties that are typically required in refinement proofs.

To further demonstrate how our approach facilitates proofs about sparse matrices, we conducted the verification task from Kellison et al.'s work in LGTM, which is marked as the case study #13 in Tab. 1. In the interest of fair comparison, the program #13 differs from a similar implementation #6 in the following aspects: (a) it works on the domain of IEEE 64-bit floating point numbers instead of integers, and (b) it does not call sub-functions (*e.g.*, #2) to keep the full resemblance to the C program by Kellison et al., which does not depend on sub-procedures that operate with sparse vectors. To verify the inner loop of `smpv` in Fig. 16 in LGTM, we first replace the manipulations with the matrix contents via `m_ind` and `m_val` by manipulations directly with the contents of its logical row $\texttt{m}[i]$, so that the remaining goal is conceptually reduced to a vector/vector product. And then align each i^{th} iteration of the inner loop with a matching access function call for the i^{th} non-zero element of the matrix's row $\texttt{m}[i]$, and prove that the "logical" iterations corresponding to zeros can be ignored. Verification of the outer loop is similar to the one presented in Sec. 2.

To summarise, we note that, instead of inventing a data representation predicate and proving lemmas about it, LGTM proof is mostly about "aligning" calls to access functions with the code being verified. That is why, from Tab. 1, our proof, even without composition, is only 36 LOC. In Kellison et al.'s work, the proofs about CSR-specific properties alone take 210 LOC, and the refinement proof takes additional 204 LOC, making our proof *less than a tenth* of their total length.

6 RELATED WORK

Our work connects several lines of research on (a) program logics for relational safety properties, (b) compilers for structured data, and (c) verification of structured data manipulations.

Relational program logics. Benton's seminal work (2004) introduced Relational Hoare Logic (RHL) to capture safety properties of *pairs of programs* and prove them using lockstep rules. RHL's extension to Separation Logic (Yang 2007) has been used to specify and verify complex relational properties of heap-manipulating programs, such as equivalence of heap-based graph-marking algorithms. Barthe et al. (2011) proposed a method to encode reasoning about 2-safety properties in unary Hoare logic by considering *product programs* that simulate the execution steps of both their constituents. More recently, Sousa and Dillig (2016) introduced Cartesian Hoare Logic (CHL) that allows one to express and prove k-safety properties of imperative programs for arbitrary but fixed k. CHL comes with the automated verifier Descartes that proves hypersafety specifications by reducing them to unary Hoare triples. None of these approaches support reasoning about families of programs indexed by elements of an arbitrary finite set that can change within the proof.

A recent work by Dardinier and Müller (2024) introduced Hyper Hoare Logic (HHL)—a framework to reason about k-safety properties of arbitrary (even infinite) arity k. Unlike LGTM, HHL only allows for k-safety specifications constraining multiple runs of *the same* program. Similarly to LGTM, in HHL the arity of a judgement can change during the proof, for instance in its Cons rule. This feature is, however, not exploited by HHL, which does not provide specialised rules for reasoning about loops, such as LGTM's For and While. The closest to our work is Logic for Hyper-triple Composition (LHC) by D'Osualdo et al. (2022). We provided a detailed discussion on the improvements LGTM makes over LHC in terms of expressivity in Sec. 3.2.

Compilers for structured data manipulations. Sparsity is a well-studied example of structured data, with wide-ranging applications from machine learning to scientific computing. A popular strategy for sparse tensor compilation is to generate low-level code directly from high-level abstractions (Kjolstad et al. 2019). The TACO tensor compiler is emblematic of the approach, defining sparse tensor formats through an iterator interface for non-zero elements (Chou et al. 2018, 2020), generating an implementation of an operation for the sparse representations from the same operation on dense arrays. Later works generalised TACO's format abstraction to capture dynamic data structures (Chou and Amarasinghe 2022), multiple simultaneous formats (Ye et al. 2023), repetition (Donenfeld et al. 2022), or ragged-style irregularity (Fegade et al. 2022). Many such approaches were unified and generalised by the Finch compiler (Ahrens et al. 2023), which goes beyond sparsity and allows one to define arbitrary iteration strategies for producing efficient looping code.

Verified computations with structured data. Arnold et al. (2010) proposed a high-level functional language LL for expressing computations with sparse tensors that facilitates automation of correctness proofs about matrix manipulations in Isabelle/HOL. The approach by Arnold et al. is, however, only limited to sparse formats that can be encoded in LL. Furthermore, it assumes correctness of the *compilation* from the LL encoding to the C representation (Arnold 2011). That is, the correctness guarantees provided by the approach only apply to LL programs, but *not* to their C counterparts.

Dyer et al. (2019) suggested to encode sparse formats and computation as Alloy predicates, phrasing verification of sparse tensor computations as a search for counterexamples. Such verification is unsound, in the sense that Alloy tries to find only small counterexamples up to a certain bound. Moreover the way to encode stateful programs in Alloy proposed by Dyer et al. is only capable to capture programs with nested `for`-loops and could not express, *e.g.*, `spmspv` from Sec. 2.

Liu et al. (2022) developed a Coq library for certified optimisations of tensor-manipulating program via verified rewrites. Those rewrites do not exploit the structure of specific data formats, and, hence, are not guaranteed to produce optimised implementations. Finally Kovach et al. (2023) presented Etch, a verified (in Lean) compiler for sparse data computations. Etch is limited to operations on structures that are expressible as mappings over indexed streams, so its optimisations can be expressed as stream fusion, hierarchical iterations, and control over interaction order (Kiselyov et al. 2017). This restricts the range of operations Etch can synthesise: for instance, it cannot generate computations on data with unordered layers, such as COO and uCSR formats.

Unlike the listed above efforts, LGTM is not a compiler, but is a verification framework: it does not synthesise computations, but offers means to mechanically verify the results produced by highly fine-tuned unverified synthesisers, such as TACO. This makes it a suitable verification back-end for certifying sparse data compilers, which are yet to be developed. Furthermore, the applications of LGTM are not limited to just sparse data: for example, our case studies #11-12 from Tab. 1 are about the run-length encoding for arrays with repeated values.

7 CONCLUSION AND FUTURE WORK

In this work we presented LGTM: a program logic for hypersafety properties, built around the idea of parametrised k-safety specifications. We argued that parametrisation enables intuitive safety proofs about intricate computations over structured data (*e.g.*, sparse tensors), and substantiated this claim by mechanically verifying thirteen case studies using LGTM embedding into Coq.

In our immediate future work, we are planning to automate common reasoning patterns of LGTM to facilitate push-the-button construction of machine-assisted proofs for parametrised hypersafety specifications. As our long-term agenda, we are aiming to use LGTM as a foundation for building a *certifying* compiler for structured data manipulations similar to Finch (Ahrens et al. 2023) by adopting the ideas of proof-producing program synthesis (Watanabe et al. 2021).

ACKNOWLEDGMENTS

We thank Andrew Appel, Kiran Gopinathan, Yunjeong Lee, Peter Müller, and George Pîrlea for their feedback on drafts of this paper. We also thank the anonymous PLDI'24 PC and AEC reviewers for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

DATA AVAILABILITY

The software artefact accompanying this paper is available online (Gladshtein et al. 2024a). The artefact contains the source code and build scripts for the Coq mechanisation of LGTM, as well as the collection of case studies that can be used to reproduce the results described in Sec. 5.

REFERENCES

Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman P. Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *CGO*. ACM, 41–54. https://doi.org/10.1145/3579990.3580020

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel. 2022. Coq's vibrant ecosystem for verification engineering (invited talk). In *CPP*. ACM, 2–11. https://doi.org/10.1145/3497775.3503951

Gilad Arnold. 2011. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes*. Ph. D. Dissertation. University of California, Berkeley, USA. http://www.escholarship.org/uc/item/2pw6165p

Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and verifying sparse matrix codes. In *ICFP*. ACM, 249–260. https://doi.org/10.1145/1863543.1863581

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM (LNCS, Vol. 6664)*. Springer, 200–214. https://doi.org/10.1007/978-3-642-21437-0_17

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *POPL*. ACM, 97–110. https://doi.org/10.1145/2103656.2103670

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *POPL*. ACM, 14–25. https://doi.org/10.1145/964001.964003

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*. ACM, 169–180. https://doi.org/10.1145/2254064.2254086

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *ICFP*. ACM, 418–430. https://doi.org/10.1145/2034773.2034828

Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

Stephen Chou and Saman P. Amarasinghe. 2022. Compilation of dynamic sparse tensor algebra. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1408–1437. https://doi.org/10.1145/3563338

Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 123:1–123:30. https://doi.org/10.1145/3276493

Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. 2020. Automatic generation of efficient sparse tensor format conversion routines. In *PLDI*. ACM, 823–838. https://doi.org/10.1145/3385412.3385963

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI (2024), 207:1–207:24. https://doi.org/10.1145/3656437

Daniel Donenfeld, Stephen Chou, and Saman P. Amarasinghe. 2022. Unified Compilation for Lossless Compression and Sparse Computing. In *CGO*. IEEE, 205–216. https://doi.org/10.1109/CGO53902.2022.9741282

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 289–314. https://doi.org/10.1145/3563298

Tristan Dyer, Alper Altuntas, and John W. Baugh Jr. 2019. Bounded Verification of Sparse Matrix Computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 36–43. https://doi.org/10.1109/Correctness49594.2019.00010

Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. In *MLSys*. mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/d3d9446802a44259755d38e6d163e820-Abstract.html

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024a. *LGTM: the Logic for Graceful Tensor Manipulation*. https://doi.org/10.5281/zenodo.10951930

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024b. Mechanised Hypersafety Proofs about Structured Data: Extended Version. *CoRR* abs/2404.06477 (2024). https://doi.org/10.48550/ARXIV.2404.06477

Changwan Hong, Aravind Sukumaran Rajam, Israt Nisa, Kunal Singh, and Ponnuswamy Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*. ACM, 300–314. https://doi.org/10.1145/3293883.3295712

Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs. In *ARITH*. IEEE Computer Society, 36–43. https://doi.org/10.1109/ARITH58626.2023.00021

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *POPL*. ACM, 285–299. https://doi.org/10.1145/3009837.3009880

Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *CGO*. 180–192. https://doi.org/10.1109/CGO.2019.8661185

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. https://doi.org/10.1145/3133901

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (2023). https://doi.org/10.1145/3591268

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498717

Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. https://doi.org/10.5281/zenodo.7118596

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *32nd IEEE Symposium on Security and Privacy*. IEEE Computer Society, 165–179. https://doi.org/10.1109/SP.2011.12

Tobi Popoola, Tuowen Zhao, Aaron St. George, Kalyan Bhetwal, Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2023. Code Synthesis for Sparse Tensor Format Conversion and Optimization. In *CGO*. ACM, 28–40. https://doi.org/10.1145/3579990.3580021

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 76:1–76:30. https://doi.org/10.1145/3133900

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying k-Safety Properties. In *PLDI*. ACM, 57–69. https://doi.org/10.1145/2908080.2908092

Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473589

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *ASPLOS*. ACM, 660–678. https://doi.org/10.1145/3582016.3582047

Received 2023-11-16; accepted 2024-03-31