Infinitary Relational Logic

VLADIMIR GLADSHTEIN, National University of Singapore, Singapore
QIYUAN ZHAO, National University of Singapore, Singapore
YUXI LING, National University of Singapore, Singapore
SEAN WANG*, Princeton University, USA
ILYA SERGEY✉, National University of Singapore, Singapore

Relational program logics are a popular formalism for stating and proving properties that relate executions of several computations. We present Infinitary Relational Logic (IRL)—the first Hoare-style Separation Logic that allows one to state and prove relational properties of possibly *infinite* families of arbitrary programs. The key insights behind IRL are to (a) generalise relational program specifications in the style of Separation Logic triples to families of programs indexed by arbitrary infinite sets, and (b) provide general proof rules that support reasoning principles guided by the structure of these index sets.

We have implemented IRL as a foundational embedding and verification tool on top of the Lean proof assistant. We demonstrate its power by showcasing both the practical and theoretical advances IRL brings to the state of the art in deductive program verification. To show the former, we use IRL to specify and prove the correctness of a series of previously unverified algorithms from computer graphics and geo-spatial information systems that iterate over array-encoded continuous objects. In doing so, we show that specifying representations of implicitly continuous data using *code* rather than traditional *state invariants* offers pragmatic benefits in the form of concise and reusable proofs, while retaining full compatibility with conventional non-relational Hoare-style reasoning. To show the latter, we use IRL to specify and verify a novel notion we call *Weird Machine Realisability*, providing the first conceptual framework that formally characterises the space of unintended behaviours permitted by a vulnerable program. All our case studies are formalised in Lean.

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

Additional Key Words and Phrases: mechanised proofs, relational logic, weird machines, Lean

ACM Reference Format:
Vladimir Gladshtein, Qiyuan Zhao, Yuxi Ling, Sean Wang, and Ilya Sergey. 2026. Infinitary Relational Logic. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 388 (October 2026), 28 pages. https://doi.org/10.1145/3839520

1 Introduction

Relational Program Logics are a Hoare-style formalism for specifying properties of sets of programs. Initially proposed by Benton (2004), relational logics offer a convenient way to state and verify relations between runs of two programs, making them an effective tool for proving correctness of program optimisations [5, 9, 16]. The power of such logics for phrasing and proving properties such as program equivalence and refinement comes from the *extensional* nature of their specifications:

*Work done during a research internship at National University of Singapore.

Authors' Contact Information: Vladimir Gladshtein, National University of Singapore, Singapore, Singapore, vgladsht@comp.nus.edu.sg; Qiyuan Zhao, National University of Singapore, Singapore, Singapore, qiyuanz@comp.nus.edu.sg; Yuxi Ling, National University of Singapore, Singapore, Singapore, yuxiling@u.nus.edu; Sean Wang, Princeton University, Princeton, USA, sw9414@cs.princeton.edu; Ilya Sergey, National University of Singapore, Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2026/10-ART388
https://doi.org/10.1145/3839520

they describe *what* the relation between inputs/outputs of the programs in question should be, rather than saying *how* the programs achieve it. For example, the Hoare-style triple (1) below states the equivalence of the two programs, A and B, *w.r.t.* their effect on the mutable variable x.

$$\left\{\ x(A) = x(B)\ \right\} \left[\begin{array}{l} A : \texttt{x := x + 1; x := 2 * x} \\ B : \texttt{x := x * 2 + 2} \end{array}\right] \left\{\ x'(A) = x'(B)\ \right\} \tag{1}$$

The notation $x(A)$ and $x(B)$ in the precondition of the triple refers to the value of the variable in the respective symbolic state component, manipulated by the program A or B, correspondingly, while the "primed" $x'(\cdot)$ in the postcondition denote the final values of the respective variables. From the pre/postconditions alone, one can conclude the equivalence of the two programs *w.r.t.* their effect on the value of the variable x without bothering to know how exactly those programs manipulate x. The proof rules supplied by relational logics allow one to establish Hoare-style triples such as (1) by means of syntax-driven decomposition of the involved programs. While the original Relational Hoare Logic by Benton only provided rules to reason about pairs of related programs, subsequent works have generalised them to reason about finite families of k arbitrary programs [16, 20, 44].

An intriguing (and often overlooked) aspect of relational Hoare-style specifications is that, when interpreted *intensionally*, they also provide a convenient way to capture the behaviour of one program in terms of another one, assuming the constraints stated by the precondition. For example, one way to interpret the triple (1) is as a specification of the program A in terms of a simpler program B. One can, indeed, argue that, for this particular example, it is easy to capture an equivalent specification for A alone in an ordinary, unary, Hoare-style triple just by adding a *functional specification* $x' = 2x + 2$ to the postcondition, thereby summarising A's effect as a logical formula. However, as the complexity of the program's behaviour or the shape of its input increases, this approach for reasoning about imperative programs, known as the *two-layer paradigm* [2, 27, 49], might incur what we believe is an unreasonably high formalisation overhead, requiring the proof to "align" the functional specification with the imperative code, while capturing the data invariants as state predicates. In contrast, specifying programs by stating their relation to *other programs* (not necessarily the equivalence) often offers a significantly more concise description of their intended behaviour, more reusable specifications, and straightforward syntax-driven proofs.

The idea of using relational specifications to express the desired program behaviour also naturally extends to *families* of imperative computations. As an example, consider the Hoare triple (2) that states that the set A_n of programs computes a right Riemann sum approximation of $\int_0^1 x^2 dx = \frac{1}{3}$:

$$\left\{\ \forall n \in \mathbb{N}, t(A_n) = s(A_n) = 0\ \right\} \left[\begin{array}{l} A_{n\in\mathbb{N}} : \texttt{for i = 1 to n:} \\ \qquad\qquad\quad \texttt{t := i / n} \\ \qquad\qquad\quad \texttt{s := s + t}^2 \texttt{ / n} \end{array}\right] \left\{\ \lim_{n\to\infty} s'(A_n) = \int_0^1 x^2 dx\ \right\} \tag{2}$$

In the triple above, the family of programs A_n and their respective symbolic states in the pre-/postcondition of the triple are *indexed* by elements of the set $\{\langle A, n\rangle \mid n \in \mathbb{N}\}$, where A is a unique uninterpreted element serving simply as a tag. To show that the desired equality involving the limits in the postcondition holds, it would suffice to show that for any $n \in \mathbb{N}$ the outcome of A_n satisfies a certain closed form, as a function of n, completing the proof by taking its limit for $n \longrightarrow \infty$.

Notice that the triple (2) has an *infinite arity* (*i.e.*, it relates infinitely many programs), and, therefore, it cannot be directly proven in any of the existing relational logics to date, which are limited to either expressing properties of finite program families [16, 20, 44] or to reasoning about (possibly infinite) sets of executions of the *same* program [13]. Furthermore, to prove (2) by following the informal argument above, one would need such a logic to feature a rule that allows for exploiting the *structure of the index set* (*i.e.*, every A_n has the same body in the `for`-loop parametrised by n), making it possible to reduce the proof of (2) to proving individual unary Hoare triples for every A_n.

In this work, we present Infinitary Relational Logic (IRL)—the first Hoare-style program logic for expressing and proving relational properties of *infinite* sets of arbitrary programs, featuring proof decomposition principles that exploit the structure of the index sets. We showcase the utility of IRL in two application domains: computational geometry and computer security. For the former, we use IRL to specify and verify programs iterating over array-encoded continuous objects, showing that its proofs are quantifiably more concise and reusable than those in unary Hoare logics. For the latter, we demonstrate how IRL provides a way to state and verify *Weird Machine Realisability* (WMR)—a new class of security-related properties beyond the reach of existing specification methodologies.

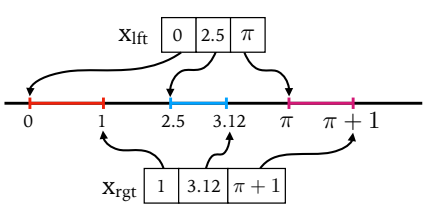
Key ideas and challenges. Our starting point for developing IRL was the recent relational separation-style program logic LGTM [20], which allowed one to state program specifications as relational Hoare triples of arbitrary but *finite* arity, providing a convenient way to specify and verify programs performing complex manipulations over multi-dimensional arrays. The key idea of this work is to generalise LGTM by lifting the finiteness restriction on its index sets, allowing for specifications that capture properties of *infinitely many* programs indexed by elements of sets such as $\mathbb{N}$ or $\mathbb{Z}$, and even uncountable ones, *e.g.*, $\mathbb{R}$ or $[0, 1]$. This conceptually simple generalisation makes it possible to specify, in a concise way, program families such as the one shown in (2), while also allowing for novel applications of infinitary relational reasoning (more on that in Sec. 5). Contrary to our initial expectations, generalising LGTM to IRL was not straightforward at all. Due to infiniteness, the meta-theory of IRL required an approach, crucially relying on the axiom of choice. Furthermore, to provide useful reasoning principles for infinite families of indexed programs in the presence of non-deterministic allocation, we had to make a very careful choice of semantics for heap-manipulating programs, constituting the main technical innovation for this work (more on that in Sec. 4).

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

- We present IRL—a new Hoare-style Separation Logic for stating and proving relational properties of infinite sets of programs. We argue for the utility of IRL by using its reasoning principles to deliver the first specifications and proofs for a number of algorithms over array-encoded continuous data from computer graphics and geo-spatial information systems, including implementations of bilinear interpolation [25] and of the count query procedure [46] (Sec. 2).
- To demonstrate the compatibility of IRL specifications with established verification techniques for heap-manipulating programs and show its quantitative advantages in terms of proof sizes and reusability, we consider a proof of a unary (*i.e.*, *non-relational*) spec for an algorithm over continuous data following the two-layer paradigm [2]. We use it to show how proofs of unary specifications can be derived from, or reduced to, proofs of relational IRL triples (Sec. 3).
- We describe the semantic foundations and the soundness argument for IRL (Sec. 4).
- We further showcase IRL with a security-inspired case study—proving Weird Machine Realisability (WMR), a novel property that formally characterises the space of unintended behaviours allowed by a vulnerable program (Sec. 5). We show that the derived in IRL domain-specific proof rules for WMR facilitate reasoning about sets of programs represented by context-free grammars.
- We implemented IRL meta-theory, rules, soundness proof, and proof automation in Lean [14], making it, to the best of our knowledge, the first feature-full Lean implementation of a relational Separation Logic to date. All our case studies are mechanised in Lean.

2 Hoare-Style Reasoning about Manipulations with Continuous Data

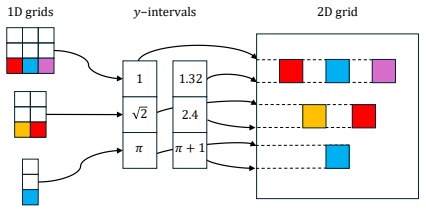
We begin the demonstration of IRL reasoning principles by showing how to use it to specify and verify programs that operate on continuous data encoded by a compressed array-based representation. As our running example, we will consider a program that computes the colour interpolation



(a) 1D and 2D grid, pictographically

```
1 def grid1_get(xlft, xrgt, xval, x):
2   ind = search(xlft, xrgt, x)
3   if (ind != -1):
4     return xval[ind]
5   else:
6     return 0
```

(b) Retrieving elements of 1D grid




```
1 def grid2_get(ylft, yrgt, xlft, xrgt, xval, x, y):
2   ind = search(ylft, yrgt, y)
3   if (ind != -1):
4     return grid1_get(xlft[ind], xrgt[ind], xval, x)
5   else:
6     return 0
```

(c) Retrieving elements of 2D grid

Fig. 1. 1D and 2D grids and their access functions

of a compressed continuous 2D grid [32]—one of many algorithms widely used in 3D deep learning to efficiently recover three-dimensional scenes from two-dimensional images.

Fig. 1a shows a graphical depiction of the memory layout in 1D and 2D grids. The one-dimensional grid consists of a collection of non-overlapping intervals with a specific colour. Each interval is represented in memory by (1) its left and right boundaries, which are stored in the arrays `xlft` and `xrgt` respectively, and (2) its colour, which is stored in the array `xval`. Following the specification approach of LGTM [20], we define the encoding of coloured intervals via *accessor functions* (*a.k.a.* getters). For instance, the accessor function `grid1_get` shown in Fig. 1b retrieves the colour of the point `x` on the one-dimensional grid. To do so, it identifies the interval that contains the point `x` by calling `search` (whose implementation we omit for brevity) at line 2 and returns the associated colour (lines 3-4). If the point does not belong to any interval, `grid1_get` returns zero (lines 5-6).

A two-dimensional grid generalises the one-dimensional case by extending intervals into rectangles. As in 1D, we define a collection of non-overlapping intervals, this time over the y-axis, using the arrays y_{lft} and y_{rgt}. Each y-interval may be shared by multiple axis-aligned rectangles that differ in their x-coordinates. For example, in Fig. 1a, the three upper rectangles share the same interval on the y-axis. To model this, we use three additional 2D arrays: x_{lft}, x_{rgt}, and x_{val}. For each y-interval defined by $y_{lft}[i]$ and $y_{rgt}[i]$, two arrays $x_{lft}[i]$ and $x_{rgt}[i]$ store the encoding of x-coordinates of *all* rectangles associated with this interval, while $x_{val}[i]$ stores their colours. The function for retrieving the colour at a point `(x, y)` in the 2D grid is shown in Fig. 1c. It works similarly to the 1D grid retrieval function, which is called at line 4.

For both axes, we assume that the boundary arrays are *well-formed*, satisfying two conditions that we elide from the assertions below for brevity but make explicit in our Lean development. First, every interval is *valid*: its left boundary does not exceed its right one, *i.e.*, $x_{lft}[k] \leq x_{rgt}[k]$ (and, likewise, $y_{lft}[k] \leq y_{rgt}[k]$). Second, consecutive intervals along an axis do *not overlap*: the right boundary of one interval precedes the left boundary of the next, *i.e.*, $x_{rgt}[k] < x_{lft}[k+1]$ (and, likewise, for the y-axis). We do not require the left- and right-boundary arrays of an axis to have equal length.

```
1 def bilinInterp(ylft, yrgt, xlft, xrgt, xval):
2   ans = 0
3   for i in range(0, len(ylft)):
4     l = linInterp(xlft[i], xrgt[i], xval[i])
5     ans = ans + l * (yrgt[i] - ylft[i])
6   return ans
```

(a) Bilinear interpolation

```
1 def linInterp(xlft, xrgt, xval):
2   ans = 0
3   for i in range(0, len(xlft)):
4     ans += xval[i] * (xrgt[i] - xlft[i])
5   return ans
```

(b) Linear interpolation

Fig. 2. Bilinear and linear interpolations of two- and one-dimensional grids correspondingly

Fig. 2a shows the implementation of the bilinear interpolation function `bilinInterp`. This function calculates the average colour of a grid within a specified region by summing up the areas of rectangles in that region, each weighted by its colour. For simplicity, in this example, the region is assumed to be the entire grid and to have a total area of 1. The code of `bilinInterp` iterates over the y-coordinate intervals of each rectangle in the grid, adding up the area of each rectangle within that y-range, weighted by its colour (lines 3-5). To compute the area of a rectangle within a single y-interval, it calls the `linInterp` function (line 4), shown in Fig. 2b. In turn, `linInterp` computes the sum of the width of each interval in the one-dimensional grid multiplied by its colour (lines 3-4).

2.1 Specifying Bilinear Interpolation with an Infinitary Relational Triple

Before we show a specification of `bilinInterp` in IRL, let us introduce a few handy notations. We will abbreviate all unmodified arrays that encode the involved data by the following *symbolic heap* predicates in the style of Separation Logic (SL) [36, 41]:

$$\begin{aligned} \mathit{Arrs}_{\mathtt{y}} &\triangleq \mathtt{arr}(\mathtt{y}_{\mathtt{lft}},\ y_{lft}) \ast \mathtt{arr}(\mathtt{y}_{\mathtt{rgt}},\ y_{rgt}) \\ \mathit{Arrs}_{\mathtt{x}}(i) &\triangleq \mathtt{arr}(\mathtt{x}_{\mathtt{lft}}[\mathtt{i}],\ x_{lft}[i]) \ast \mathtt{arr}(\mathtt{x}_{\mathtt{rgt}}[\mathtt{i}],\ x_{rgt}[i]) \ast \mathtt{arr}(\mathtt{x}_{\mathtt{val}}[\mathtt{i}],\ x_{val}[i]) \\ \mathit{Arrs} &\triangleq \mathit{Arrs}_{\mathtt{y}} \ast \left(\mathop{\ast}_{i=0}^{|y_{lft}|-1} \mathit{Arrs}_{\mathtt{x}}(i)\right) \end{aligned} \tag{3}$$

In the definitions (3), $\mathtt{arr}(\mathtt{x}_{\mathtt{lft}},\ x_{lft})$ is an SL predicate asserting that in a heap constrained by it, $\mathtt{x}_{\mathtt{lft}}$ (given in monospace) is a base pointer of an array that stores a sequence x_{lft} (given in *italic*). Now, following the idea of specifying programs via relations to other programs, we will provide `bilinInterp` with a spec stated in terms of an *uncompressed* version of a given 2D grid. Such an uncompressed representation can be seen as mathematical function $\mathit{Grid}(x, y)$ which, given two real-valued coordinates x and y, returns the colour of a grid rectangle containing the point (x, y). With such representation, we can express the result of bilinear interpolation simply as $\iint_{\mathbb{R}^2} \mathit{Grid}(x, y)\, dx\, dy$. This integral is the sum of the areas of all rectangles in the grid weighted by their colour, *i.e.*, the value of the $\mathit{Grid}(x, y)$ *at each point* (x, y), which we can define as the output of `grid2_get` function called on *all* $(x, y) \in \mathbb{R}^2$. Therefore, the correctness specification of `bilinInterp` can be expressed as the following Hoare triple relating its result to those of `grid2_get`:

$$\left\{\mathit{Arrs}(\cdot)\right\}\left[\begin{array}{ll} 1 & : \mathtt{bilinInterp}(\mathtt{y}_{\mathtt{lft}}, \mathtt{y}_{\mathtt{rgt}}, \mathtt{x}_{\mathtt{lft}}, \mathtt{x}_{\mathtt{rgt}}, \mathtt{x}_{\mathtt{val}}) \\ \langle 2, x, y\rangle_{\substack{x\in\mathbb{R}\\ y\in\mathbb{R}}} & : \mathtt{grid2\_get}(\mathtt{y}_{\mathtt{lft}}, \mathtt{y}_{\mathtt{rgt}}, \mathtt{x}_{\mathtt{lft}}, \mathtt{x}_{\mathtt{rgt}}, \mathtt{x}_{\mathtt{val}}, x, y) \end{array}\right]\left\{\frac{a,}{\mathit{Grid}} \;\middle|\; \left[a = \iint_{\mathbb{R}^2} \overline{\mathit{Grid}}(x, y)\, dx\, dy\right]\right\} \tag{4}$$

As this is the first IRL triple we encounter, let us explain the anatomy of its notation. Reading from left to right, a triple consists of four parts: (1) a *precondition* (here, $\mathit{Arrs}(\cdot)$); (2) a list of *indexed program components* of the form $\iota : p$, each associating an index ι (or a set of indices) with a program p; (3) a comma-separated list of *result-binding variables*, one per program component; and (4) a *postcondition* that may mention those variables, separated from the binders by a vertical bar "|". In (4), the two binders are stacked vertically: the top one, a, binds the result of `bilinInterp`,

while the bottom one, $\overline{Grid}$, binds the (infinitely many) results of the `grid2\_get` runs. The comma separating them stresses that these are two distinct bindings rather than a single compound expression. The specification (4) relates the execution of an uncountable number of programs indexed by elements of the set $\{1\} \cup \{2\} \times \mathbb{R}^2$. The collection of all programs can be split in two sets: (1) a single program `bilinInterp` (indexed by 1) and (2) $\mathbb{R}^2$ runs of `grid2_get`, at each specific point on a 2D plane. For example, the second set of programs is represented as a collection of runs $\texttt{grid2\_get}(\texttt{y}_{\texttt{lft}}, \texttt{y}_{\texttt{rgt}}, \texttt{x}_{\texttt{lft}}, \texttt{x}_{\texttt{rgt}}, \texttt{x}_{\texttt{val}}, x, y)$ taken for *all* triples $\langle 2, x, y\rangle \in \{2\}\times\mathbb{R}\times\mathbb{R}$. The precondition of the triple (4) asserts $Arrs(\cdot)$ predicate over all infinitely many initial heaps (*a.k.a.*, a *hyper-heap*) used to run the respective program components. The $(\cdot)$ notation indicates *distinct copies* of the predicate *Arrs* across all state components indexed by elements of $\{1\} \cup \{\langle 2, i, j\rangle \mid i, j \in \mathbb{R}\}$.

The postcondition of the triple (4) constrains the matching set of program outcomes (it's convenient to think of them as an infinite vector with $1 + |\mathbb{R}^2|$ components). That is, the result value of the `bilinInterp` program is a single real number, bound in the postcondition by the variable a. At the same time, the results of $|\mathbb{R} \times \mathbb{R}|$ runs of `grid2_get` are bound to a $\overline{Grid}$ function (*i.e.*, an infinite vector indexed by pairs of reals), whose individual values are retrieved as $\overline{Grid}(x, y)$ in the postcondition. The rest of the postcondition states the relation between the outputs of `bilinInterp` and the many runs `grid2_get` functions. Specifically, it asserts that the result of the `bilinInterp` function is equal to a mathematical representation of bilinear interpolation expressed as a double integral of the $\overline{Grid}$ function. The $\lceil\cdot\rceil$ notation stands for *pure* mathematical facts that do not depend on the program state, while the *spatial* part of the postcondition, constraining the heaps, is omitted for the sake of demonstration, since none of the programs change the arrays they manipulate.

An astute reader might wonder if we are allowed to make use of integration in the postcondition without proving that $\overline{Grid}(x, y)$ is integrable on $\mathbb{R}$ (*i.e.*, its integral over $\mathbb{R}$ exists and is finite).[1] We observed that, in practice, the integrability side condition only arises when we need to perform algebraic manipulations with an integral or to compute its concrete value. To avoid the "premature" burden of discharging it early, once and for all, we treat the result of integration symbolically throughout our proofs until the steps that crucially rely on algebraic properties of integration.

In the rest of this section, we will verify the validity of the specification (4) via the rules of IRL.

2.2 Extending Relational Proofs to Infinitely Many Programs

We start our walk through the proof of (4) with a primer on the reasoning principles of LGTM, the finite-arity precursor of IRL, and discuss how to generalise them for verifying the correctness of relational Hoare triples with infinite arity. We will start by using LGTM rules to (a) partially symbolically execute the first program component in the triple (4) and (b) filter out the runs of `grid2_get` called on a grid area *outside* of its rectangles, since they do not contribute to the integral.

To get to the `for`-loop in the first component of the triple, we first unfold definition of `bilinInterp`. For brevity, we omit the encoding-specific arguments $\texttt{x}_{\texttt{lft}}$, $\texttt{x}_{\texttt{rgt}}$, ..., of the calls to `grid2_get`.

$$\left\{ Arrs(\cdot) \right\} \left[\begin{array}{ll} 1 & : \texttt{ans} = \texttt{alloc(0)}; \ldots \\ \langle 2, x, y\rangle_{(x,\,y)\,\in\,\mathbb{R}^2} & : \texttt{grid2\_get}(x, y) \end{array} \right] \left\{ \frac{a,}{\overline{Grid}} \;\middle|\; \left\lceil a = \iint_{\mathbb{R}^2} \overline{Grid}(x, y)\, dx\, dy \right\rceil \right\} \tag{5}$$

Now we can symbolically execute the `alloc` statement[2] by *focusing* on the first component of the triple by using the SeqU1 rule, shown in Fig. 3 (a consolidated summary of all IRL proof rules is provided in Appendix A). This rule is applicable in a scenario where the index set of a triple can be split into two *disjoint* sets $\{\iota\}$ and S. In this case, SeqU1 rule allows one to advance the first

[1] In our Lean mechanisation we use the definition of Bochner integral and Bochner integrability from mathlib [31].
[2] There are no stack variables in IRL's programming language; values for newly declared variables are heap-allocated.

$$\textsc{SeqU1}\ \frac{\{\ P\ \}\,[\iota:\ p_1]\,\{\ x\ \mid\ H\ \} \quad \{\ H\ \}\,[\ \iota:\ p_1',\ S:\ \mathcal{P}_2\ ]\,\{\ x,\overline{z}\ \mid Q(\overline{z})\ \}}{\{\ P\ \}\,[\ \iota:p_1;p_1',\ S:\mathcal{P}_2\ ]\,\{\ x,\overline{y}\ \mid Q(x\overline{y})\ \}} \qquad \textsc{SeqU2}\ \frac{\{P\}\,[\ \iota:\ p_1,\ S:\ \mathcal{P}_2\ ]\,\{\ x,\overline{z}\ \mid H(\overline{z})\ \} \quad \forall\overline{z},\{\ H(\overline{z})\ \}\,[\iota:\ p_1']\,\{\ x\ \mid\ Q(x\overline{z})\ \}}{\{\ P\ \}\,[\ \iota:p_1;p_1',\ S:\ \mathcal{P}_2\ ]\,\{\ x_1,\overline{x_2}\ \mid Q(x_1\overline{x_2})\ \}}$$

$$\textsc{Focus}\ \frac{S = S_1 \uplus S_2 \quad \{\ P\ \}\,[S_1:\ \mathcal{P}_1]\,\{\ \overline{x}\ \mid\ H(\overline{x})\ \} \quad \forall\overline{x},\ \{\ H(\overline{x})\ \}\,[S_2:\ \mathcal{P}_2]\,\{\ \overline{y}\ \mid\ Q(\overline{x}\cdot\overline{y})\ \}}{\{\ P\ \}\,[S:\ \mathcal{P}_1\uplus\mathcal{P}_2]\,\{\ \overline{z}\ \mid\ Q(\overline{z})\ \}}$$

Fig. 3. IRL rules adopted from LGTM verbatim

component of the sequential composition $p_1; p_1'$ in the ι-indexed component of an overall triple. Here, we apply SeqU1 taking $\iota = 1$ and $S = \{\langle 2, x, y\rangle \mid x, y \in \mathbb{R}\}$. The first premise of SeqU1 in our example is instantiated with a pointer-allocating command.

$$\{Arrs(\cdot)\}\ \left[\ 1 : \texttt{ans} = \texttt{alloc(0)}\ \right]\ \{Arrs(\cdot)\ \ *\ \texttt{ans(1)} \mapsto 0\}$$

The `(1)` notation in the assertion $\texttt{ans(1)} \mapsto 0$ indicates that the `ans` pointer is allocated in the 1-indexed component of a symbolic (hyper-)heap. Abbreviating the `for`-loop with f and the loop body with $p(i)$, the second premise of SeqU1, which we now need to prove, becomes:

$$\left\{\begin{array}{c}\texttt{ans(1)} \mapsto 0\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{cc} 1 & : f;\ \texttt{return ans}\\ \langle 2,x,y\rangle_{(x,\,y)\,\in\,\mathbb{R}^2} & : \texttt{grid2\_get}(x,y)\end{array}\right]\left\{\begin{array}{c|c} a, & \left[a = \iint_{\mathbb{R}^2} \overline{Grid}(x,y)\,dx\,dy\right]\\ \overline{Grid} & \end{array}\right\} \tag{6}$$

This can be further simplified by chopping off the `return` statement via SeqU2:

$$\left\{\begin{array}{c}\texttt{ans(1)} \mapsto 0\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{cc} 1 & : \texttt{for } i \texttt{ in range}(0, |y_{lft}|)\ \ \{p(i)\}\\ \langle 2,x,y\rangle_{(x,\,y)\,\in\,\mathbb{R}^2} & : \texttt{grid2\_get}(x,y)\end{array}\right]\left\{\begin{array}{c|c} -, & \texttt{ans(1)} \mapsto\\ \overline{Grid} & \iint_{\mathbb{R}^2} \overline{Grid}(x,y)\,dx\,dy\end{array}\right\} \tag{7}$$

2.2.1 Separating Conjunctions over Infinite Sets. Recall that the `for`-loop in (7) iterates over all y-coordinate intervals of the grid encoding. This means that the loop will skip all the points of the grid with y-coordinates outside of such intervals. This observation means we can get rid of all runs of $\texttt{grid2\_get}(x, y)$ where $y \notin \left[y_{lft}[i], y_{rgt}[i]\right)$ for any $i < |y_{lft}|$. Furthermore, note that for all such y, the output of the call to the `grid2_get` function is zero. We can express this observation as follows:

$$\{Arrs(\cdot)\}\left[\begin{array}{cc}\langle 2,x,y\rangle_{x\,\in\,\mathbb{R}} & : \texttt{grid2\_get}(x,y)\\ y \notin \bigcup_k \left[y_{lft}[k], y_{rgt}[k]\right) & \end{array}\right]\left\{\begin{array}{c|c} \overline{Grid} & \left[\forall x\,y,\ \overline{Grid}(x,y) = 0\right]\\ & *\ Arrs(\cdot)\end{array}\right\} \tag{8}$$

Reasoning about such triples is enabled via the new InfProd rule of IRL shown in Fig. 4. The intuition behind InfProd rule is that if the pre-/postconditions of an IRL triple can be split into a collection of independent *unary* assertions in Separation Logic (SL), we should reduce it to a collection of unary Hoare triples with SL-style pre-/postconditions. InfProd rule is the first IRL rule that is conceptually different from its LGTM analogue Prod, which has almost the same shape as InfProd with every use of $\fatslash$ replaced by the *iterated* separating conjunction operator $\ast$. In the Prod rule, $\ast_{i\in S} P_i$ denotes a series of separately conjuncted facts P_i over a *finite* index set S, where each P_i asserts facts only about the i-indexed copy of the heap. However, this rule may not apply in IRL where an index set can be infinite. To address this issue, we introduce an *infinite* version of the iterated separating conjunction, which we call *septegral* and denote as $\fatslash$. For a collection of unary SL assertions P_i indexed by elements of a (possibly infinite) set S, $\fatslash_{i\in S} P_i$ is a hyper-heap assertion where each P_i constrains a distinct copy of the symbolic heap indexed by i. If a unary SL predicate P is the same across all indices then the meaning of a septegral of P is the same as $P(\cdot)$.

Coming back to the InfProd rule, if the pre-/postconditions of a relational triple can be presented as a septegral over its index set S, one can reduce it to a single unary SL triple quantified over an

$$\textsc{InfProd}\ \frac{\forall i, \left\{\ \lceil i \in S\rceil\ *\ P_i\ \right\}[\mathcal{P}(i)]\left\{\ x\ \middle|\ Q_i(x)\ \right\}}{\left\{\ \circledast_{i\in S} P_i\ \right\}[S:\mathcal{P}]\left\{\ \overline{x}\ \middle|\ \circledast_{i\in S} Q_i(\overline{x}(i))\ \right\}} \qquad \textsc{InfInfProd}\ \frac{\forall j \in \mathcal{I}, \left\{\ \circledast_{i\in S_j} P_i\ \right\}[S_j:\mathcal{P}]\left\{\ \overline{x}\ \middle|\ Q_j(\overline{x})\ \right\}}{\left\{\ \circledast_{i\in \biguplus_{j\in\mathcal{I}} S_j} P_i\ \right\}\left[\biguplus_{j\in\mathcal{I}} S_j : \mathcal{P}\right]\left\{\ \overline{x}\ \middle|\ \forall j \in \mathcal{I}, Q_j(\overline{x})\ \right\}}$$

Fig. 4. IRL rules for disjoint sets of programs. Side conditions in InfInfProd about Q are omitted for brevity.

index $i \in S$. The precondition of (8) can be thus rewritten using the following equivalence (9):

$$\left\lceil \forall x\, y,\ \overline{Grid}(x,y) = 0 \right\rceil\ *\ Arrs(\cdot) \iff \circledast_{x,y} \left\lceil \overline{Grid}(x,y) = 0 \right\rceil\ *\ Arrs \tag{9}$$

With the right-hand side of the equivalence (9), we can apply the InfProd rule, reducing (8) to

$$\left\{\lceil y \notin \textstyle\bigcup_k \left[y_{lft}[k], y_{rgt}[k]\right)\rceil\ *\ Arrs\right\}\ [\texttt{grid2\_get}(x,y)]\ \left\{g\ \middle|\ \lceil g = 0\rceil\ *\ Arrs\right\}, \tag{10}$$

which is just a unary triple. To reason about such triples, IRL provides standard SL rules.

The InfProd rule is, in fact, derived from the more general IRL rule InfInfProd (also in Fig. 4), which allows one to reduce an IRL triple on disjoint sets of programs to a collection of IRL triples on each set. We will show another application of InfInfProd in Sec. 5.3.3.

2.2.2 Focusing on Essential Triple Components. Next, let us see how, by making use of the proven triple (8), we can remove all idle runs of `grid2_get` from (7). by applying the Focus rule from Fig. 3. We apply the rule to the triple (7), taking $S_1 := \left\{\langle 2, x, y\rangle \mid x \in \mathbb{R},\ y \notin y_{proj}\right\}$, where y_{proj} abbreviates $\bigcup_k \left[y_{lft}[k], y_{rgt}[k]\right)$. One subgoal is discharged by (8) and the other remains:

$$\left\{\begin{array}{l}\texttt{ans(1)} \mapsto 0\ * \\ \left\lceil \forall x \in \mathbb{R}\ y \notin y_{proj}, \overline{Grid_0}(x,y) = 0\right\rceil \\ \quad * \ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll} 1 & : \texttt{for} \ldots \\ \langle 2,x,y\rangle_{\substack{x\in\mathbb{R}\\ y\in y_{proj}}} & : \texttt{grid2\_get}(x,y)\end{array}\right]\left\{\begin{array}{l|l} \_, & \texttt{ans(1)} \mapsto \\ \overline{Grid} & \iint \left(\overline{Grid} \cup \overline{Grid_0}\right)(x,y)\,dx\,dy\end{array}\right\}$$

In the triple above, $\overline{Grid_0}$ binds all outputs of idle (*i.e.*, 0-producing) `grid2_get` runs. We can simplify the postcondition by splitting the integral into two parts, abbreviating the set $\mathbb{R} \setminus y_{proj}$ with y^c_{proj}:

$$\begin{aligned}\iint_{\mathbb{R}^2} \left(\overline{Grid} \cup \overline{Grid_0}\right)(x,y)dx\,dy &= \int_{y_{proj}}\int_{\mathbb{R}} \overline{Grid}(x,y)\,dx\,dy + \int_{y^c_{proj}}\int_{\mathbb{R}} \overline{Grid_0}(x,y)\,dx\,dy \\ &= \int_{y_{proj}}\int_{\mathbb{R}} \overline{Grid}(x,y)\,dx\,dy + \int_{y^c_{proj}}\int_{\mathbb{R}} 0\,dx\,dy = \int_{y_{proj}}\int_{\mathbb{R}} \overline{Grid}(x,y)\,dx\,dy\end{aligned}$$

The first equality in this series of rewrites is justified by the additive integration law and disjoint domains of $\overline{Grid}$ and $\overline{Grid_0}$ functions. The second one follows from the assertion in the precondition, as $\overline{Grid_0}$ is always 0. To establish this equality we need to show additionally that (a) $\lambda y. \int_{\mathbb{R}} \overline{Grid}(x,y)\,dx$ is integrable on y_{proj} and that (b) $\lambda y. \int_{\mathbb{R}} \overline{Grid_0}(x,y)\,dx$ is integrable on y^c_{proj}. The latter one is trivial since $\overline{Grid_0}$ is a constant zero function. We postpone the proof of the former until a bit later.
By using the simplification above and erasing the pure assertion about $\overline{Grid_0}$, we obtain:

$$\left\{\begin{array}{l}\texttt{ans(1)} \mapsto 0 \\ \quad * \ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{ll} 1 & : \texttt{for}\ i\ \texttt{in}\ \texttt{range}(0, |y_{lft}|)\ \ \{p(i)\} \\ \langle 2,x,y\rangle_{\substack{x\in\mathbb{R}\\ y\in y_{proj}}} & : \texttt{grid2\_get}(x,y)\end{array}\right]\left\{\begin{array}{l|l} \_, & \texttt{ans(1)} \mapsto \\ \overline{Grid} & \int_{y_{proj}}\int_{\mathbb{R}} \overline{Grid}(x,y)\,dx\,dy\end{array}\right\} \tag{11}$$

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

Fig. 5. An LGTM rule for `for`-iteration

2.2.3 Aligning Loop Iterations with the Corresponding Getters. To prove (11) we observe that the k^{th} iteration of the loop computes the interpolation of all grid rectangles with y coordinates in $\left[y_{lft}[k], y_{rgt}[k]\right)$. We exploit this fact by rewriting the integral in the postcondition as follows:

$$\int_{y_{proj}} \int_{\mathbb{R}} \overline{Grid}(x,y)\, dx\, dy = \sum_k \int_{\left[y_{lft}[k], y_{rgt}[k]\right)} \int_{\mathbb{R}} \overline{Grid}(x,y)\, dx\, dy \tag{12}$$

Notice that the equation (12) also simplifies our pending obligation (a) regarding integrability of $\lambda y. \int_{\mathbb{R}} \overline{Grid}(x,y)\, dx$ on y_{proj}, reducing it to proving the integrability of that function on $\left[y_{lft}[k], y_{rgt}[k]\right)$ for each k. Let us leave it once again for now and revise later in Sec. 2.3.

Next, we note that the k^{th} iteration should increase the value of the `ans(1)` by the value of the k^{th} component of summation (12), and in its turn, the k^{th} member of the summation (12) depends only on the outputs of `grid2_get`(x,y) invoked with y from the interval $\left[y_{lft}[k], y_{rgt}[k]\right)$. This suggests that we can *align* the k^{th} iteration of the `for`-loop with the set of all runs of `grid2_get`(x,y) on the k^{th} interval of y_{proj}. We can do so using the standard LGTM For rule from Fig. 5. The premise of the rule divides the index set S into N parts, S_i. Each part corresponds to a "batch" of programs aligned with one loop iteration. The loop invariant I is a statement parametrised by two elements: (*i*) an integer representing the *next* value of the loop counter and (*ii*) a hyper-value (*i.e.*, a vector) populated with the results of all programs from the batches executed "in sync" with the preceding iterations of the `for`-loop. The conclusion of the rule assumes that the invariant $I(0,\overline{\emptyset})$ holds prior to the loop execution, meaning that no iterations took place ($\overline{\emptyset}$ denotes an empty hyper-value). By the end, $I(N,\overline{y})$ indicates that the loop has completed all iterations, and the results of all aligned program batches are contained in $\overline{y}$. The rule's inductive step presumes that the invariant holds at the start for some $i < N$ and a hyper-value $\overline{x}$. It then asserts in the postcondition that, at the end, the invariant will hold for the subsequent index $i+1$ and the aggregated hyper-value $\overline{x} \cup \overline{y}$.
To proceed with our proof of (11), we split the index set we define the loop invariant as follows:

$$\left\{\langle 2,x,y\rangle \,\middle|\, x \in \mathbb{R},\ y \in y_{proj}\right\} = \bigcup_{k=0}^{|y_{lft}|-1} \left\{\langle 2,x,y\rangle \,\middle|\, x \in \mathbb{R},\ y \in \left[y_{lft}[k], y_{rgt}[k]\right)\right\} \tag{13}$$

$$I_{\texttt{for}}(n, \overline{Grid}) = Arrs(\cdot)\ *\ \texttt{ans(1)} \mapsto \sum_{k=0}^{n-1} \int_{\left[y_{lft}[k], y_{rgt}[k]\right)} \int_{\mathbb{R}} \overline{Grid}(x,y)\, dx\, dy \tag{14}$$

The invariant states that (a) all arrays encoding the grid are unchanged and (b) `ans` accumulates the interpolation of grid rectangles with y coordinates in $\left[y_{lft}[k], y_{rgt}[k]\right)$ for all $k < n$. After applying the rule For to (11), and replacing the unchanged part of the summation in $I_{\texttt{for}}$ with s we get :

$$\left\{\begin{array}{l}\texttt{ans(1)} \mapsto s \\ *\ Arrs(\cdot)\end{array}\right\} \left[\begin{array}{ll} 1 & : \texttt{let l = linInterp(...) in} \\ & \ \ \ldots \\ \langle 2,x,y\rangle_{x \in \mathbb{R}} & : \texttt{grid2\_get}(x,y) \\ y \in \left[y_{lft}[k], y_{rgt}[k]\right) & \end{array}\right] \left\{\begin{array}{l|l} -, & \texttt{ans(1)} \mapsto s + \\ \overline{Grid} & \int_{\left[y_{lft}[k], y_{rgt}[k]\right)} \int_{\mathbb{R}} \overline{Grid}(x,y)\, dx\, dy \end{array}\right\}$$

To further simplify this triple, note that the first command in the `grid2_get` is `search(`$\texttt{y}_{\texttt{lft}}, \texttt{y}_{\texttt{rgt}}, y$`)` function, which is now called on all values y inside the same $\left[y_{lft}[k], y_{rgt}[k]\right)$ interval. Remember

$$\textsc{LetU}\ \frac{\{P\}\left[\iota:\ p_1,\ S:\ \mathcal{P}_2\right]\{\ x,\overline{z}\mid H(x,\overline{z})\ \}\quad \forall x,\overline{z},\{\ H(x,\overline{z})\ \}\left[\iota:\ p_1'(x)\right]\{\ x\mid Q(x\overline{z})\ \}}{\{\ P\ \}\left[\iota:\texttt{let x} = p_1\ \texttt{in}\ p_1'(x),\ S:\mathcal{P}_2\right]\{\ x,\overline{y}\mid Q(x\overline{y})\ \}}$$

$$\textsc{Merge}\ \frac{\texttt{local}(\{P_1,Q\},S_1)\quad \mu:S_2\to S_2' \text{ is surjective}\quad \forall i\in S_2,\mathcal{P}_2(i)=\mathcal{P}_2'(\mu(i))\quad \left\{P_1 * \circledast_{S_2'} P_2\right\}\left[\begin{array}{c}S_1:\mathcal{P}_1\\ S_2':\mathcal{P}_2'\end{array}\right]\{\overline{x},\overline{y}\mid Q(\overline{x},\overline{y}\circ\mu)\}}{\left\{P_1 * \circledast_{S_2} P_2\right\}\left[\begin{array}{c}S_1:\mathcal{P}_1\\ S_2:\mathcal{P}_2\end{array}\right]\{\overline{x},\overline{y}\mid Q(\overline{x},\overline{y})\}}$$

Fig. 6. LetU and Merge rules

that `search(lft, rgt, x)` returns the index of the interval in which `x` is located. Hence, for all calls of `grid2_get`, the output of the `search` in our triple is the same, so we simplify the triple to (15):

$$\left\{\begin{array}{c}\texttt{ans(1)}\mapsto s\\ *\ Arrs(\cdot)\end{array}\right\}\left[\begin{array}{cc}1 & :\texttt{let l = linInterp(...) in...}\\ \langle 2,x,y\rangle_{x\in\mathbb{R}} & :\texttt{grid1\_get}(\texttt{x}_{\texttt{lft}}[\texttt{k}],\texttt{x}_{\texttt{rgt}}[\texttt{k}],\texttt{x}_{\texttt{val}}[\texttt{k}],x)\\ y\in\left[y_{lft}[k],y_{rgt}[k]\right) & \end{array}\right]\left\{\begin{array}{c}-,\\ \overline{Grid}\end{array}\ \middle|\ \begin{array}{c}\texttt{ans(1)}\mapsto s+\\ \iint\overline{Grid}(x,y)\,dx\,dy\end{array}\right\}\tag{15}$$

2.3 Merging Specifications of Infinitely Many Programs

So far, we have seen how the IRL rule InfProd (Fig. 4), which *reduces* verification of a set of programs to verifying just one. Let us now demonstrate the second technical innovation of IRL, which has been proven invaluable for verifying infinite relational triples: a rule for *merging* an infinite set of syntactically equivalent programs that deliver the same outcome (independent of the individual programs' indices) into a single program. To explain the rule, we continue with our running example, which we have left at the state of proving the triple (15). This triple should be provable from the specification of *linear* interpolation stated as follows:

$$\left\{Arrs(\cdot)\right\}\left[\begin{array}{cc}1 & :\texttt{linInterp}(\texttt{x}_{\texttt{lft}},\texttt{x}_{\texttt{rgt}},\texttt{x}_{\texttt{val}})\\ \langle 2,x\rangle_{x\in\mathbb{R}} & :\texttt{grid1\_get}(\texttt{x}_{\texttt{lft}},\texttt{x}_{\texttt{rgt}},\texttt{x}_{\texttt{val}},x)\end{array}\right]\left\{\begin{array}{c}a,\\ \overline{Grid}\end{array}\ \middle|\ \texttt{a}=\int_{\mathbb{R}}\overline{Grid}(x)\,dx\right\}\tag{16}$$

To finish the proof, one might expect to be able to apply the standard LetU rule of LGTM from Fig. 6 to (15), making use of (16). Unfortunately, the goal (15) cannot be tackled this way as the index sets in the triples' second components *do not match*. That is, the triple (15) has a call of `grid1_get` for each $x\in\mathbb{R}$ and $y\in\left[y_{lft}[k],y_{rgt}[k]\right)$, while in (16), this function is called "only" once for each $x\in\mathbb{R}$. This kind of problem appears frequently when combining specifications of programs that operate with geometric data of different dimensionality. To solve it, IRL introduces the Merge rule (Fig. 6), which replaces components of a triple that correspond to runs of the *same program* by a single program. In this particular example, Merge reduces the cardinality of the second component of (15), by merging all duplicated calls of `grid1_get` for each $x\in\mathbb{R}$ into one.

To apply the Merge rule, we first need to split the programs in our triples into two distinct sets: $S_1:\mathcal{P}_1$ and $S_2:\mathcal{P}_2$. Intuitively, $S_2:\mathcal{P}_2$ is going to be a set of programs with potential duplicates. Next, we pick a surjective "merging" function $\mu:S_2\to S_2'$, which is allowed to squash only those elements of S_2 that correspond to equal programs in $\mathcal{P}_2$. The latter is enforced by the condition $\forall i\in S_2,\mathcal{P}_2(i)=\mathcal{P}_2'(\mu(i))$, where $\mathcal{P}_2'$ is set of "merged" programs in a new triple. Finally, the pre-condition of the original triple to prove should be split into (1) P_1 which is *local* to set S_1 (*i.e.*, only contains assertions over states indexed by elements in S_1), and (2) $\circledast_{S_2}P_2$ which is a unary heap assertion P_2, to be replicated over S_2. After applying the rule, P_1 remains unaffected (as μ only operates on S_2) and $\circledast_{S_2}P_2$ turns into $\circledast_{S_2'}P_2$, indicating that we have merged those assertions in the collection P_2 that correspond to duplicate programs. For simplicity of reasoning, the rule also

Table 1. Statistics for the verified programs. For each program, we specify its operation in mathematical notation, and the return type, followed by the size of its specification statement (*Spec*), the sizes of the code in its original form (C_O) and after ANF transformation (C_A), the mechanised proof (Pf), and the proof/code ratios *w.r.t.* both forms of the code. We also indicate the characteristic IRL rules and the auxiliary specifications used by the proof. The correctness proofs of the auxiliary getter functions are *reused* across all case studies sharing a data format and are not counted in Pf: the `search` proof shared by the grid-format cases 1–4 is 75 LOC, and the getter proof shared by the point-counting cases 5–7 is 111 LOC.

#	Operation	Return Type	Size (LOC)				Ratio		Prominent rules used		Relies on
			Spec	C_O	C_A	Pf	C_O	C_A	InfProd	Merge	
1	$\int_{\mathbb{R}} Grid(x)\ dx$	$\mathbb{R}$	11	9	14	43	4.8	3.1	✓		
2	$\iint_{\mathbb{R}} Grid(x,y)\ dx\ dy$	$\mathbb{R}$	11	11	24	64	5.8	2.7	✓	✓	#1
3	$\int_0^1 Grid(p+x)\ dx$	$\mathbb{R}$	12	22	34	102	4.6	3.0	✓		
4	$\iint_0^1 Grid(p+x,q+y)\ dx\ dy$	$\mathbb{R}$	13	27	42	128	4.7	3.0	✓	✓	#3
5	$\lvert Ps(i,x) \cap Interval(x)\rvert$	$\mathbb{Z}[]$	11	16	22	89	5.6	4.0	✓		
6	$\lvert Ps(i,x,y) \cap Box(x,y)\rvert$	$\mathbb{Z}[]$	12	20	35	117	5.9	3.3	✓		#5
7	$\lvert Ps(i,x,y) \cap Circle(x,y)\rvert$	$\mathbb{Z}[]$	10	46	66	250	5.4	3.8	✓		

requires the postcondition predicate $Q(\overline{x},\overline{y})$ to be local to the non-merged indices S_1, which means that it also stays unchanged. This is the case in all of the case studies we have considered.

The Merge rule allows for reducing the proof of an IRL triple for a set of programs over S_2 to another over a smaller set S_2' that contains the same programs. Using a similar line of reasoning, the converse should also hold: if a triple has been proved over S_2', then we can derive a corresponding triple over S_2, as it contains the same programs. This converse pattern is captured by the Expand rule, whose formulation is essentially obtained by swapping the two triples in the Merge rule. We will describe one use of the Expand rule in Sec. 5.3.4.

To proceed with the proof of (15), we first rewrite its precondition using the equivalence:

$$\texttt{ans(1)} \mapsto 0 \ *\ Arrs(\cdot) \iff (\texttt{ans(1)} \mapsto 0 \ *\ Arrs(1)) \ *\ \fint_{j \in \left\{\langle 2,x,y\rangle \mid x\in\mathbb{R},\ y\in\left[y_{lft}[k],y_{rgt}[k]\right)\right\}} Arrs(j)$$

Taking $S_2 \triangleq \left\{\langle 2,x,y\rangle \mid x\in\mathbb{R},\ y\in\left[y_{lft}[k],y_{rgt}[k]\right)\right\}$ and $\mu \triangleq \lambda(x,y).x$ reduces (15) to the following specification (17) (leaving out the pointer arguments to `grid1_get` for simplicity):

$$\left\{\begin{array}{l}\texttt{ans(1)} \mapsto s\\ *\ Arrs(\cdot)\end{array}\right\} \left[\begin{array}{ll} 1 & : \texttt{let l=linInterp(...) in}\\ & \ldots\\ \langle 2,x\rangle_{x\in\mathbb{R}} & : \texttt{grid1\_get}(x)\end{array}\right] \left\{ \_,\ \overline{Grid} \;\middle|\; \begin{array}{l}\texttt{ans(1)} \mapsto s + \\ \fint_{\left[y_{lft}[k],y_{rgt}[k]\right)} \int_{\mathbb{R}} \overline{Grid}(x)\ dx\ dy\end{array}\right\} \tag{17}$$

Note that in the postcondition, the outermost integral over y produces a constant factor for $\int_{\mathbb{R}} \overline{Grid}(x)\ dx$. Using standard integration rules we can simplify it to $(y_{rgt}[k]-y_{lft}[k]) \int_{\mathbb{R}} \overline{Grid}(x)\ dx$. Since the function $\lambda y.\ \int_{\mathbb{R}} \overline{Grid}(x)\ dx$ is constant on $\left[y_{lft}[k],y_{rgt}[k]\right)$, it is trivially integrable on $\left[y_{lft}[k],y_{rgt}[k]\right)$, assuming $\lambda x.\overline{Grid}(x)$ is integrable on $\mathbb{R}$, thus, further simplifying our integrability side condition. We can now apply LetU using the specification (16) of the linear interpolation:

$$\left\{\begin{array}{l}\texttt{ans(1)} \mapsto s\ *\\ \left[\texttt{l} = \int_{\mathbb{R}} \overline{Grid}(x)\ dx\right]\end{array}\right\} \left[\texttt{1 : ans += l} * (y_{rgt}[k]-y_{lft}[k])\right] \left\{ - \;\middle|\; \begin{array}{l}\texttt{ans(1)} \mapsto s + \\ (y_{rgt}[k]-y_{lft}[k]) \int_{\mathbb{R}} \overline{Grid}(x)\ dx\end{array}\right\} \tag{18}$$

The triple (18) is trivially provable using standard SL rules. The only facts left to prove are the triple (16) and integrability of $\lambda x.\overline{Grid}(x)$ on $\mathbb{R}$. Their proofs exercise a subset of all techniques we have just shown and can be found in the Lean development accompanying this paper [21].

2.4 Other Geometric Case Studies

To evaluate our Lean implementation of IRL as a verification tool, we mechanised several additional examples: more realistic versions of linear and bilinear interpolations, as well as count queries for points on a plane in a box and a circle [46]. The statistics summarising the verification effort and the used features of the logic for the resulting collection of seven case studies is shown in Tab. 1. The proof-effort statistics (Pf) report only the program-specific proof lines; we account for the theorem specifications and for the correctness proofs of the auxiliary functions used by the format getters (such as `search` in Fig. 1b) separately, since the latter are *reused* across all case studies that share a data format. To make this concrete, the specification of bilinear interpolation (case 2) is about 11 LOC, and it relies on roughly 75 LOC of auxiliary lemmas establishing the specification of the `search` function. This `search` proof is shared across all four grid-format case studies (cases 1–4), so its cost is amortised over the whole suite rather than paid per case study; the point-counting case studies (cases 5–7) likewise share the correctness proof of their common getter.

The seven case studies fall into two groups, corresponding to the two application domains highlighted in the abstract. The first four (cases 1–4) are *computer-graphics* workloads that interpolate an image encoded as a grid: cases 1 and 2 are exactly the linear and bilinear interpolations specified by (16) and (4) over the entire grid, as discussed above, while cases 3 and 4 are their more realistic variants that interpolate over an arbitrary unit interval $[p, p{+}1]$ (resp. rectangle $[p, p{+}1] \times [q, q{+}1]$) rather than the whole grid. Their efficient implementations use a `while`-loop (Sec. 4.3) that stops scanning once the current interval leaves the region of interest. The remaining three (cases 5–7) are *geo-spatial* queries [46] that count how many points of a stored set fall inside a query region: an interval on a line (case 5), an axis-aligned box (case 6), and a circle (case 7). The box and circle queries first scan the y-coordinates of the points and then, for each relevant y, scan their x-coordinates, reusing the interval query. All seven case studies rely on the Focus and InfProd rules to discard the (infinitely many) getter runs that fall outside the region of interest, and both bilinear interpolations additionally use the Merge rule to collapse the duplicated calls to the linear getter, as reflected in the last two columns of Tab. 1.

All programs are verified in their A-normal form (ANF) [18] so that intermediate computation steps are exposed, thereby facilitating forward-style reasoning. Thanks to Lean's metaprogramming support [38], the ANF transformation is performed at the expression level during program definition, without requiring any manual effort. Our proof scripts place each tactic that applies a single IRL rule on its own line; the additional proof overhead mainly comes from the mathematical reasoning.

3 From Unary to Relational Specifications and Back Again

Even though IRL brings the benefits of short specifications and proofs, it introduces a new formalism, so a sceptical reader might wonder, whether it would be compatible with conventional unary Hoare-style specifications and proofs in Separation Logic. Luckily, the answer to that is yes. To show the compatibility of IRL with standard SL, let us revisit the bilinear interpolation algorithm verified in Sec. 2, now with the goal of establishing a more traditional *unary* specification for it. We explore two strategies to achieve this goal: the first follows the well-established two-layer paradigm [2, 27] and uses only non-relational, standard SL rules; the second leverages the relational triple (4) already proven in Sec. 2 and, by applying a series of rules from IRL, derives a proof for the unary specification. Through this example, we illustrate how proofs of unary specifications and IRL specifications for the same program can be systematically transformed into one another.

3.1 A Unary Specification and Its Proof in Unary Separation Logic

Intuitively, an ordinary specification for `bilinInterp` should assert in the postcondition that the result of `bilinInterp` is exactly the double integral of the underlying function $f(x, y)$ that defines the 2D grid. Since f is not expressed explicitly, but rather encoded via a collection of arrays, we need to relate their contents to f properly. Following the existing work [27], we define a *representation predicate* $grid\_rep(f, y_{lft}, y_{rgt}, x_{lft}, x_{rgt}, x_{val})$ to capture how these arrays encode f. Consistent with the earlier presentation in Sec. 2, y_{lft} and y_{rgt} indicate (mathematical) lists of reals that record the y-coordinates of the two boundaries of each rectangle, while x_{lft}, x_{rgt} and x_{val} are nested lists of reals. For each i, $x_{lft}[i]$ and $x_{rgt}[i]$ record the x-coordinates of each rectangle's boundaries, and $x_{val}[i]$ record these rectangles' colours accordingly. All rectangles represented by $x_{lft}[i], x_{rgt}[i]$ and $x_{val}[i]$ have the same y-coordinates of boundaries, namely $y_{lft}[i]$ and $y_{rgt}[i]$. With *grid_rep* in hand, we can assign `bilinInterp` with the following unary specification, abbreviating the pure assertion $\lceil grid\_rep(f, y_{lft}, y_{rgt}, x_{lft}, x_{rgt}, x_{val}) \rceil$ as I_{grid} to highlight its role as a representation invariant.

$$\left\{I_{grid} * Arrs\right\} \left[\texttt{bilinInterp}(\texttt{y}_{\texttt{lft}}, \texttt{y}_{\texttt{rgt}}, \texttt{x}_{\texttt{lft}}, \texttt{x}_{\texttt{rgt}}, \texttt{x}_{\texttt{val}})\right] \left\{ a \;\middle|\; \left\lceil a = \iint_{\mathbb{R}^2} f(x,y)\,dx\,dy \right\rceil * I_{grid} * Arrs \right\} \tag{19}$$

The complete definition of the predicate *grid_rep* is rather intricate, and relies on a number of auxiliary constructions, which can be found in full in the accompanying artefact [21]. The proof of the spec (19) is done by finding a function that expresses the integral from its postcondition *explicitly* and stating it through the state components constrained by *grid_rep*. We will comment on the quantitative aspects of this proof in comparison to the one conducted in IRL in Sec. 3.4.

3.2 Deriving a Relational Triple from a Unary Specification

We can derive the IRL specification (4) from the unary spec (19) as follows. First, we prove that the getter procedure $\texttt{grid2\_get}(x, y)$ indeed returns the function value $f(x, y)$:

$$\left\{I_{grid} * Arrs\right\} \left[\texttt{grid2\_get}(\texttt{y}_{\texttt{lft}}, \texttt{y}_{\texttt{rgt}}, \texttt{x}_{\texttt{lft}}, \texttt{x}_{\texttt{rgt}}, \texttt{x}_{\texttt{val}}, x, y)\right] \left\{ a \;\middle|\; \lceil a = f(x,y) \rceil * I_{grid} * Arrs \right\} \tag{20}$$

Second, we apply the InfProd rule from Fig. 4 on (20) to obtain the following relational triple:

$$\left\{I_{grid} * Arrs(\cdot)\right\} \left[\langle 2, x, y\rangle_{(x,\,y)\,\in\,\mathbb{R}^2} \;:\texttt{grid2\_get}(x, y)\right] \left\{ \overline{Grid} \;\middle|\; \left\lceil \overline{Grid} = f \right\rceil * I_{grid} * Arrs(\cdot) \right\} \tag{21}$$

Finally, we apply the Focus rule from Fig. 3 on (19) and (21) with S_1 being $\{1\}$ (thereby implicitly "lifting" the unary triple (19) to a relational triple with a singleton index set) and S_2 being $\{2\} \times \mathbb{R}^2$. This results in the following relational triple as desired.

$$\left\{ \begin{matrix} Arrs(\cdot) \\ * I_{grid} \end{matrix} \right\} \left[\begin{matrix} 1 & :\texttt{bilinInterp}(\texttt{y}_{\texttt{lft}}, \texttt{y}_{\texttt{rgt}}, \texttt{x}_{\texttt{lft}}, \texttt{x}_{\texttt{rgt}}, \texttt{x}_{\texttt{val}}) \\ \langle 2, x, y\rangle & :\texttt{grid2\_get}(x, y) \end{matrix} \right] \left\{ \begin{matrix} a \\ \overline{Grid} \end{matrix} \;\middle|\; \begin{matrix} I_{grid} * \left\lceil \overline{Grid} = f \right\rceil * Arrs * \\ \left\lceil a = \iint_{\mathbb{R}^2} f(x,y)\,dx\,dy \right\rceil \end{matrix} \right\} \tag{22}$$

The general recipe for deriving a relational triple about a "main" program and getters from the unary triple of the main program would be to (1) connect the imperative getter to some mathematical object by proving a unary triple, (2) "lift" such unary triple to IRL with the InfProd rule, and (3) "combine" the unary proof about the main program with the relational spec of getters via Focus.

3.3 Deriving a Unary Specification from a Relational Triple

Surprisingly, it is also possible to derive the unary specification (19) from the relation IRL triple (4)! The intuition is that post-conditions of (4) and (21) together contain enough information to reconstruct the unary triple. First, (4) says that the result of `bilinInterp` is the double integral of getters outcomes $\overline{Grid}$. And second, (21) says this $\overline{Grid}$ is *exactly* the function f from I_{grid}.

The key property of IRL, which allows us to exploit the above intuition, is that IRL semantics is total, and hence adding new programs to triples is sound. This gives rise to the derived Extension rule (we omit its formulation), which is basically the inverse of the Focus rule (Fig. 3); it is sound because adding new programs only strengthens the triple. By applying the Extension rule, we can extend the unary triple (19) with the relational triple of getter procedures (21) to obtain (22), as the new proof obligation. To prove (22), we employ another rule to add the pure facts about getters from the postcondition of (21) to the postcondition of (4); this rule is also sound since the spatial assertions in the pre/postconditions of (21) are the same, and thus adding more pure facts about getters "does not hurt." Applying the extended (4) to the proof goal (22) concludes the proof.

3.4 On Pragmatic Benefits of Relational Proofs

Now that we have seen that the bilinear interpolation can be specified both in IRL and in a unary Separation Logic, it is natural to wonder: which of the two specification and proof styles leads to shorter and more reusable proofs? Quantitative comparisons of this sort between program logics are quite rare (we are not aware of any), but since we implemented IRL as a foundational verifier in Lean, with a fully-fledged implementation of a unary SL as its component,[3] we were equipped with the tools to conduct this experiment. We provide a quantitative assessment of the three distinct proof styles for the bilinear interpolation program, as presented in Sec. 3.1 (completely unary), Sec. 3.3 (unary-via-relational), and Sec. 2 (completely relational).

Tab. 2 summarises the number of lines of code (LOC) required for each category (listed on the left-most column) of proof under the three proof styles. The categories are listed in *reusability decreasing order*. Since the unary-via-relational proof effort (abbreviated as "UvR") builds on the purely relational one, the latter's LOC statistics is counted into the former's total lines of code. The first row indicates how many lines each proof style contributes to the lemmas that are reusable across verification efforts of programs operating on *different formats* of continuous data. Among the three proof styles, only the relational proof provides a `search` function (Fig. 1) that is payload-agnostic: its spec can be reused for other formats with a layered structure and potentially different payload. The second row corresponds to the parts that are reusable for *the same data format*. The third row accounts for the parts that can be reused when performing the same operation on the same data format. This includes, for example, a lemma from the unary proof, which shows that the representation justifies the double integral on a 2D grid. In the relational proof, however, no analogous lemma is needed, as the corresponding reasoning has been *internalised* into the logic itself via dedicated rules, especially the loop aligning ones (*e.g.*, the For rule from Fig. 5). The fourth row represents the non-reusable parts—those that each proof style must pay individually. These are essentially the proofs of `bilinInterp` carried out using either standard Separation Logic rules or IRL rules. Beyond proof size, the burden on the user also differs significantly. In the unary approach, the user must first devise a representation invariant and then construct the rest of the proof around it. Although this invariant and its proofs are reusable within the same data format (*i.e.*, fall under the "Format" category), it does not reach the level of generality found in the relational proof's "Layer" category. From a practical standpoint, even if the user ultimately seeks a unary spec, using the unary-via-relational approach may still be more economical (500 *vs.* 465 LOC).

Table 2. Proof effort (LOC) by category and style

	Unary	UvR	Relational
Layer	0	0	110
Format	180	120	26
Operation	200	0	0
Other	120	45	164
Total	500	165 + 300	300

[3]We have released our Lean implementation of the unary Separation Logic as an open-source repository [19].

$$\begin{array}{lll} \text{Expressions} & e ::= n \mid r \mid \mathsf{true} \mid \mathsf{false} \mid v \mid e = e \mid e < e \mid e \wedge e \mid \neg e \mid e \oplus e \mid \texttt{length}(e) \\ \text{Command} & c ::= !x \mid x[d] \mid x := e \mid x[e] := e \mid \texttt{for}\ v\ \texttt{in}\ \texttt{range}(e,e)\ \{c\} \mid \texttt{while}\ (e)\ c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\{c\} \mid \\ & \quad e\ ;\ e \mid \texttt{let}\ v := c\ \texttt{in}\ c \mid x := \texttt{alloc}(v)\ \texttt{in}\ c \mid x := \texttt{malloc}(n)\ \texttt{in}\ c \mid \texttt{let}\ v := f(\overline{arg})\ \mid \texttt{return}\ e \end{array}$$

Fig. 7. IRL language. Locations (ranged over by x, y) and *variables* (v, u) are encoded as alpha-numeric strings; *arguments* ($\overline{arg}$) are lists of locations and variables. Constants are unbounded integers (n) or reals (r).

These observations collectively suggest that relational proofs not only reduce the overall mechanised verification effort but also offer a more scalable and reusable approach, particularly valuable when reasoning about programs iterating over continuous data, and, more generally, arrays.

4 Meta-Theory and Soundness of Infinitary Relational Logic

In this section, we present the meta-theory of IRL. We first discuss the key challenges arising from the infinitary nature of IRL (Sec. 4.1). We then describe the IRL language along with its semantics and additional logic rules (Sec. 4.2–Sec. 4.3), concluding with the soundness theorem.

4.1 From Finite to Infinite Meta-Theory

Hyper-heaps in LGTM are defined as partial finite maps (PFMs) from an index set S to heaps (also PFMs) [20], capturing an arbitrary but *finite* number of heaps across independent runs. Having hyper-heaps defined in terms of PFMs allows for carrying out meta-theory proofs in LGTM using *induction* over the size of heap domains. In IRL, where index sets can be infinite, a very different proof approach is needed, crucially relying on the axiom of choice. To demonstrate the difference, consider the distributivity of septegrals over the existential quantifier of hyper-heap assertions. Given a set of indices S and a mapping $H : S \to A \to \mathit{hProp}$, the lemma states:

$$\circledast_{i \in S} \exists^{h} x,\ H_i(x) \iff \exists^{hh} x,\ \circledast_{i \in S} H_i(x)$$

(here we denote, the existential quantifier over heap assertions with $\exists^{h}$, and the existential quantifier over hyper-heap assertions with $\exists^{hh}$). In LGTM, the septegral corresponds to a finite iterated separating conjunction defined recursively, so this lemma is proven by induction over $|S|$: the empty case is trivial and the inductive step follows from the distributivity of $\exists^{h}$ over the separating conjunction. In IRL, induction is not possible. After unfolding the definitions, the left-hand side begins with $\forall i \in S,\ \exists x \in A, \ldots$ and the right-hand side with $\exists f : S \to A, \ldots$, which is exactly *skolemisation*: a principle equivalent to the axiom of choice. The use of induction over finite index sets in LGTM is not restricted to this single lemma: the same proof strategy underpins reasoning about the connection between septegrals and the separating conjunction of Separation Logic, index substitution in hyper-heap assertions, as well as composition and decomposition of IRL triples. In IRL, all such results (amounting to around 20 lemmas) had to be reproven using fundamentally different techniques suited to the infinite setting.

4.2 Programming Language and Deterministic Semantics

The programming constructs of the IRL language capture a necessary subset of common heap-manipulating languages, such as C and Python (Fig. 7). The two unusual aspects of this language are (a) a syntactic discrimination between locations (memory addresses) and all other values, and (b) region-based memory management discipline, in which the memory space allocated by commands such as `let x := malloc(n) in c` has a lexically-scoped lifetime bounded by the end of the command c. Both these aspects are crucial for the soundness of the Merge rule (Sec. 2.3) in the presence non-deterministic memory allocation, as we explain below.

The soundness of the Merge rule (Fig. 6) heavily relies on the fact that merged programs behave deterministically. As a negative example, assume we have a non-deterministic operator `nondet` in the language and consider the triple $\{\mathsf{emp}\}\ [1 : \texttt{nondet}; 2 : \texttt{nondet}]\ \{x,\ y \mid \lceil x = y \rceil\}$. As `nondet` is non-deterministic, the triple above is false: outputs x and y of `nondet` can be different. However, if we apply Merge rule with $\mu : \{1, 2\} \to \{1\} \triangleq \lambda x.1$, we can reduce it to $\{\mathsf{emp}\}\ [1 : \texttt{nondet}]\ \{x \mid \lceil x = x \rceil\}$, which is clearly true. As is the case with many Separation Logics for evolving heap-based data structures, the programming language of IRL comes with the command: `alloc()` for dynamic memory allocation. It *non-deterministically* allocates a fresh pointer in a heap, and returns its address. The value of such address can vary depending on the current memory state, so the result of calling `alloc()` might be different across a set of identical programs in an IRL triple. Making allocation deterministic by calculating a specific next heap location it should use is possible, and would require "baking in the frame" into the assertion model to prove the SL Frame rule sound [6]. This solution, however, is unsuitable for proving the soundness of the Merge rule, intuitively, due to the need to "reconcile" possibly different frames when justifying the merging of assertions.

As a solution, we adopt the approach to describe behaviour of individual programs in IRL based on *omnisemantics* [10]. Omnisemantics relates a pair $\langle c, s\rangle$ of a program and a state to a *set* Q of its possible outcomes $\langle v, s'\rangle$, *i.e.*, results values and final states. It is shown to be convenient for handling non-determinism and is equivalent to the standard operational semantics for deterministic computations. The rules of omnisemantics for IRL are standard and are defined inductively on the structure of a program. The most interesting rule for region-based memory allocation is:

$$\textsc{AllocExec}\ \frac{\forall p \notin s, \exists Q_p, \langle c[p/x], s \cup (p, v)\rangle \Downarrow Q_p \wedge \left(\forall v'\ s', Q(v', s') \Leftrightarrow \exists v_p, Q_p(v', s' \cup (p, v_p))\right)}{\langle x\ :=\ \texttt{alloc}(v)\ \texttt{in}\ c, s\rangle \Downarrow Q}$$

According to this rule, to construct a set of all possible outputs Q of the program $x\ :=\ \texttt{alloc}(v)\ \texttt{in}\ c$ evaluated on the state s, we first run the program $c[p/x]$ on an extended state $s \cup (p, v)$ for *each possible pointer* p outside the state s, each such run resulting in a set of possible outcomes Q_p. Next, as the pointer p is allocated non-deterministically, we take a union of all possible outcomes Q_p, removing the pointer p from the states in such Q_p.[4] This rule is the only source of possible non-determinism in IRL semantics according to the following definition of determinism:

$$\langle c, s\rangle \Downarrow Q \Longrightarrow \forall\, \langle v_1, s_1\rangle, \langle v_2, s_2\rangle \in Q,\ v_1 = v_2 \wedge s_1 = s_2 \tag{23}$$

Indeed, even if we know that each $c[p/x]$ has executed deterministically to $\langle v_p, s_p\rangle$, each v_p and s_p might be different, so the set of all possible outputs Q is not a singleton.

Perhaps surprisingly, with the syntactic restrictions on the treatment of locations (Fig. 7) and their lexically-scoped lifetimes, we can still show that IRL programs are nevertheless deterministic. Proving this requires establishing two auxiliary facts. First, we ensure that the set of pointers in the program's input and output heaps are *always the same*. This fact follows from region-based memory allocation, which ensures that all fresh allocated pointers get deallocated by the end of the lexically determined program region [23]. Second, we prove that for any two outcomes of a program the result values as well as values stored at each each location in the output heaps are equal. This fact can only be proven if the execution of c does not depend on the addresses of the pointers allocated inside c. As a negative example, consider the program `x := alloc(0) in x`, which allocates a pointer `x` storing the integer value `0` and returns its address. Although the memory containing the address `x` will be deallocated at the end of the program execution, thanks to our memory management discipline, the pointer address of `x` has been chosen non-deterministically, which means that such program might have multiple outcomes (and also return a dangling pointer!).

[4] As such, the AllocExec rule follows the *precise* variant of omnisemantics [10, §2.3], since it works directly with indexed families of postconditions instead of performing an over-approximation.

$$\textsc{While}\ \frac{S = \bigcup_{i=0}^{N-1} S_i \qquad \begin{array}{l} \forall \overline{x},\ \{\ I(N,\overline{x})\ \}\ [\ \iota : c\ ]\ \{\ b \mid [b = \mathit{false}] * I(N,\overline{x})\ \} \\ \forall \overline{x}, k < N,\ \{\ I(k,\overline{x})\}\ [\ \iota : \texttt{if}\ c\ \texttt{then}\ p,\ S_k : \mathcal{P}_k]\ \{\ z, \overline{y} \mid I(k+1, \overline{xy})\ \} \end{array}}{\{\ I(0,\overline{\emptyset})\}\ [\ \iota : \texttt{while}\ (c)\ \{p\},\ S : \mathcal{P}]\ \{\ z, \overline{y} \mid I(N,\overline{y})\ \}}$$

Fig. 8. A rule for `while`-iteration.

To rule out such programs, we treat locations differently from ordinary values (*i.e.*, integers, reals, and booleans), and syntactically only allow one to use location commands that explicitly read from and write to a pointer or an array element (*cf.* Fig. 7). Once we have proven the determinism of our semantics, we can assume that $\Downarrow$ relates a program and a state to a single output value and state.

To our surprise, the proof of semantic determinism of IRL was highly non-trivial in the presence of such language constructs as memory allocation, procedure calls, and loops. The main challenge was that it required developing two different semantics for the IRL language. The first is a standard substitution-based semantics, which is natural for stating and proving the soundness of the logic rules. However, reasoning about substitution in this setting is very tedious: one must carefully distinguish how substitutions act on values and locations, since IRL constrains computations not to depend on the identity of allocated locations. Furthermore, it would require measure-based inductions over the program structure. To simplify the determinism proof, we developed an alternative environment-based semantics, in which variable bindings are tracked in an environment rather than eagerly substituted into the program text. This formulation made the determinism argument considerably more manageable. Nevertheless, the proof effort required to establish the correspondence between the two semantics and use it to derive determinism is still substantial, amounting to around 1.5kLOC in our Lean development. Additional notes on the semantics of IRL assertions are provided in the accompanying artefact [21].

4.3 The While Rule

In Sec. 2.2.3, we explained the For rule, which aligns iterations of a `for`-loop with batches of auxiliary programs. Most of our mechanised case studies from Tab. 1, however, are implemented using `while`-loops. Reasoning about `while`-loops in IRL requires a dedicated rule.

The While rule (Fig. 8) embodies the following intuition: a `while` (c) $\{p\}$ loop, which terminates after *no more* than N steps, is simply equivalent to just N iterations of `if` c `then` p. Therefore, if we divide our index set S of auxiliary programs into N batches $S_0, \ldots, S_{N-1}$ (left assumption in the rule), we can align each batch with one execution of `if` c `then` p (bottom right assumption in the rule). The condition c might become false on any step $k < N$, allowing the loop to terminate before all the batches are exhausted. Finally, the top-right premise of the rule ensures termination: once all N batches have been processed, evaluating the loop condition c under the invariant $I(N,\overline{x})$ must still preserve the invariant and yield $b = \mathit{false}$, guaranteeing that the loop exits.

4.4 Soundness Theorem

The soundness definition of IRL is standard [20]. We say that IRL triple $\{P\}\ [S : \mathcal{P}]\ \{Q\}$ is *derivable* and write $\vdash_{\mathsf{IRL}} \{P\}\ [S : \mathcal{P}]\ \{Q\}$ if this triple can be obtained by using the rules of the logic. We say that the triple $\{P\}\ [S : \mathcal{P}]\ \{Q\}$ is *semantically valid* and write

$$\models \{P\}\ [S : \mathcal{P}]\ \{Q\} \triangleq \forall h, P(h) \Rightarrow \exists h'\ \overline{v}, \langle \mathcal{P}, h\rangle \Downarrow_S \langle h', \overline{v}\rangle \wedge Q(\overline{v})(h')$$

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

Theorem 4.1 (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,*

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

We mechanised the meta-theory of IRL in Lean. The size of the development is around 20kLOC.

5 Deductive Proofs of Weird Machine Realisability

In this section, we demonstrate a novel application of IRL in a computer security domain that has not previously been examined through the lens of deductive program logic-based reasoning: the characterisation of program vulnerabilities in the setting of so-called Weird Machines [8].

5.1 Weird Machines and Exploit Realisability

A *Weird Machine* (WM) is a formal abstraction that models exploits as a set of *unintended behaviours* permitted by a vulnerable program [8, 17, 39, 45]. Weird Machines provide a semantic description for the behaviours of security vulnerability exploits, including popular code reuse attacks, such as Return-Oriented Programming (ROP) [42], Jump-Oriented Programming (JOP) [7], and Data-Oriented Programming (DOP) [24]. Recent work has attempted to formalise Weird Machines in various ways: as a collection of "weird states" that exists in the actual execution but not in the informal program model [17], as a result of insecure compilation [39], or in a form of type-and-effect system that represents Weird Machines as regular expressions of exploits ascribed to programs [29].

The starting point for our idea to characterise a Weird Machine hidden in a vulnerable program is an observation made recently by Ling et al. that a space of unintended behaviours caused by a vulnerability in a program C can be captured by a domain-specific language L (described, *e.g.*, by a regular or context-free grammar) with a guarantee that *any* program $l \in L$ is *realisable*, *i.e.*, it can be simulated by a certain run (or a set of runs) of the original vulnerable program C [30]. Following this idea, we propose the notion of *Weird Machine Realisability* (WMR) to describe a WM enabled by a vulnerable program. A Weird Machine is considered realisable if and only if *all* the exploits it describes are realisable. Here, "realisable" means that there exists a *payload*—initial values of state components controlled by an attacker, *e.g.*, though a buffer overflow,—that forces the vulnerable program to achieve the execution intended by the exploit. A formal definition is below:

Definition 5.1 (Weird Machine Realisability). Let L be a Weird Machine of a vulnerable program C and $\mathcal{P}$ be the set of payloads for C. We say that L is realisable by C if the following holds:

$$\forall h, \forall l \in L, \exists p \in \mathcal{P}, \exists h', \langle l, h\rangle \Downarrow h' \wedge \langle C(p), h\rangle \Downarrow h'$$

Notice that Definition 5.1 relates executions of two (possibly infinite) families of programs: the Weird Machine described by the language L and the vulnerable program $C(\cdot)$ run with different payloads $p \in \mathcal{P}$. This makes IRL a perfect fit to describe WM realisability and provide a proof system for it, allowing one to establish that a program C does indeed harbour a Weird Machine L.

5.2 Specifying Weird Machine Realisability via Relational Hoare Triples

Fig. 9 shows a simple C program `vuln` with a buffer overflow vulnerability. The program takes two integer values passed by the pointers `x` and `y`, which are assigned to the stack variables `pa` and `pb`. However, at line 5, the `gets` function allows arbitrary data from an attackers to be written into the local buffer `buf`. Without a boundary checking of the input, `gets` can overwrite values of the recently assigned variables `pa` and `pb`, thus, giving the attacker control over the subsequent **`for`**- and **`if`**/**`else`**-statements, as well as the arithmetic operations involving the values `*x` and `*y`,

correspondingly. We model the WM induced over `vuln` by this memory corruption vulnerability as the following the context-free language L (24) for manipulating mutable variables x and y:[5]

$$\begin{array}{llll} L \to S_1; S_4 & S_1 \to S_2; S_1; S_3 \mid \epsilon & S_4 \to S_5 \mid S_6 & \\ S_2 \to \mathsf{x} := \mathsf{x}+1 & S_3 \to \mathsf{y} := \mathsf{y}+1 & S_5 \to \mathsf{y} := \mathsf{x}+\mathsf{y} & S_6 \to \mathsf{y} := \mathsf{x}-\mathsf{y} \end{array} \tag{24}$$

The language L includes all unintended executions controlled by the corrupted variables `pa` and `pb`, modifying the the variables `x` and `y`, which are modelled by passed-by-pointer parameters `x` and `y` of `vuln`, correspondingly. To state the realisability of L as a WM over `vuln`, let us make several convenient abbreviations and simplifications to the program state model. First, we will be using `vuln` to refer to the body of the procedure in Fig. 9. Second, instead of modelling the buffer overflow precisely, we will consider `vuln` to be parameterised by all possible payloads $p \in \mathcal{P}$—pairs of integer values $\langle pa, pb \rangle$ that can be given to variables `pa` and `pb` of `vuln`. We will, thus, index the family of vulnerable program runs with different payloads by the elements of $\mathcal{P}$ (spelled as $\mathtt{vuln}(p), p \in \mathcal{P}$), whereas programs in L will be indexed by themselves. Finally, we will refer to the pair values $\langle xv, yv \rangle$ associated with `x` and `y` in a postcondition's symbolic component indexed by some element i as $\sigma(i)$. With these conventions, realisability of L over `vuln` can be specified by the following IRL triple:

```
1  void vuln (int *x, int *y) {
2    int pa = *y;
3    int pb = *x;
4    char buf[10];
5    gets(buf, 18); //buffer overflow
6    for (int k = 0; k < pa; k++)
7      *x = *x + 1;
8    for (int k = 0; k < pa; k++)
9      *y = *y + 1;
10   if (pb > 10)
11     *y = *x + *y;
12   else
13     *y = *x - *y;
14 }
```

Fig. 9. The vulnerable program

$$\forall xv\ yv, \left\{ \nexists_{i \in L \cup \mathcal{P}} \mathsf{x}(i) \mapsto xv * \mathsf{y}(i) \mapsto yv \right\} \begin{bmatrix} p \in \mathcal{P} & : & \mathtt{vuln}(p) \\ l \in L & : & l \end{bmatrix} \left\{ \forall l \in L, \exists p \in \mathcal{P}, \sigma(l) = \sigma(p) \right\} \tag{25}$$

The precondition states that all programs in this triple, indexed by $L \cup \mathcal{P}$ (we abuse the grammar notation: L means the language $[\![L]\!]$), share the same initial state where `x` and `y` are set to arbitrary values xv and yv. The postcondition asserts a relation between the outcomes of the WM programs and those of `vuln`: for every program $l \in L$, there exists a payload $p \in \mathcal{P}$ such that the execution of $\mathtt{vuln}(p)$ reaches the same heap state upon termination. That is, each program in L is realisable by some exploit payload p passed to `vuln`, capturing precisely Definition 5.1 of WM realisability.

Non-determinism in IRL. The specifications of IRL do not fundamentally rely on built-in language-level non-determinism. Eq. 25 demonstrate that specifications over non-deterministic programs, such as `vuln()`, can be reduced to specifications over deterministic programs parameterised by choices in the form of ordinary program arguments. In this way, instead of reasoning about a program with language-level non-deterministic primitives, we reason about a family of deterministic programs indexed by the corresponding choices, and quantify over those indices at the specification level. Concretely, line 5 of Fig. 9 can be viewed as performing non-deterministic assignments to `pa` and `pb`, with the choices reified as payload components. The property "there exists an execution of the (non-deterministic) program such that ..." then becomes "there exists a payload such that the (deterministic) program instantiated with that payload ...". The intended existential reasoning is preserved, but expressed via quantification over explicit arguments.

5.3 Proving Weird Machine Realisability in IRL

The proof of the triple (25) will take advantage of its postcondition being in the form $\forall\exists$ and having no dependency on the specific return values of any program in the set. To support proving WM realisability, we will make use of four new proof rules tailored specifically for families of programs

[5]Here, we assume implicit coercion between string concatenation $\circ$ and the sequential composition constructor ; from Fig. 7.

$$\textsc{Weaken}\ \frac{\mathcal{P} \supseteq \mathcal{P}_1 \quad \mathtt{local}(H_1, \mathcal{P} \setminus \mathcal{P}_1) \quad \{\ H_1\ \}\,[C(\mathcal{P} \setminus \mathcal{P}_1)]\,\{\ \top\ \} \quad \{\ H_2\ \}\,[C(\mathcal{P}_1), L]\,\{\ Q(L, \mathcal{P}_1)\ \}}{\{\ H_1 * H_2\ \}\,[\ C(\mathcal{P}),\ L\ ]\,\{Q(L, \mathcal{P})\ \}}$$

$$\textsc{GrmDisj}\ \frac{L = L_1 \uplus L_2 \quad \mathtt{local}(H, \mathcal{P}) \quad \mathtt{local}(H_1, L_1) \quad \{\ H * H_1\ \}\,[C(\mathcal{P}), L_1]\,\{\ Q(L_1, \mathcal{P})\ \} \quad \mathtt{local}(H_2, L_2) \quad \{\ H * H_2\ \}\,[C(\mathcal{P}), L_2]\,\{\ Q(L_2, \mathcal{P})\ \}}{\{\ H * H_1 * H_2\ \}\,[C(\mathcal{P}),\ L\ ]\,\{Q(L, \mathcal{P})\ \}}$$

$$\textsc{Payload}\ \frac{p_0 \in \mathcal{P} \quad \mathtt{local}(H, \mathcal{P} \setminus \{p_0\}) \quad \{\ H\ \}\,[C(\mathcal{P} \setminus \{p_0\})]\,\{\ \top\ \} \quad \{\ H_0\ \}\,[C(p_0), L]\,\{\ \forall l \in L, \sigma(l) = \sigma(p_0)\ \}}{\{\ H * H_0\ \}\,[C(\mathcal{P}),\ L\ ]\,\{Q(L, \mathcal{P})\ \}}$$

$$\textsc{GrmSeq}\ \frac{\left\{\ \not\ast_{i \in L_1 \cup \mathcal{P}_1} H\ \right\}\,[C_1(\mathcal{P}_1), L_1]\,\left\{\ \not\ast_{i \in L_1 \cup \mathcal{P}_1} R_i\ \right\} \quad \tau : L_1 \to \mathcal{P}_1 \text{ is surjective} \quad \forall h, h \models \not\ast_{i \in L_1 \cup \mathcal{P}_1} R_i \Rightarrow h \models (\forall l \in L_1, \sigma(l) = \sigma(\tau(l))) \quad \forall j \in L_1, \left\{\ \not\ast_{i \in L_2 \cup \mathcal{P}_2} R_j\ \right\}\,[C_2(\mathcal{P}_2), L_2]\,\{\ Q(L_2, \mathcal{P}_2)\ \}}{\left\{\ \not\ast_{i \in L_1 \circ L_2 \cup \mathcal{P}_1 \times \mathcal{P}_2} H\ \right\}\,[\ C_1(\mathcal{P}_1); C_2(\mathcal{P}_2), L_1 \circ L_2\ ]\,\{Q(L_1 \circ L_2, \mathcal{P}_1 \times \mathcal{P}_2)\ \}}$$

Fig. 10. Derived structural IRL rules for verifying WMR. $Q(L, \mathcal{P})$ is a shorthand for $\forall l \in L, \exists p \in \mathcal{P}, \sigma(l) = \sigma(p)$.

indexed by their own syntactic encodings in a context-free grammar (Fig. 10). All these rules are derived from basic IRL rules. Due to space constraints, we do not show their derivations, but they can be found in our Lean code. We start the proof of (25) by abbreviating $\mathtt{x} \mapsto xv * \mathtt{y} \mapsto yv$ with $H_{\mathtt{xy}}$ and unfolding the definitions of $\mathcal{P}$ and L, obtaining the following goal:

$$\left\{\ \not\ast_{i \in S_1 \circ S_4 \cup \mathbb{Z}^2} H_{\mathtt{xy}}\ \right\}\left[\begin{array}{ll} p \in \mathbb{Z}^2 & : \mathtt{vuln}(p) \\ l \in S_1 \circ S_4 & : l \end{array}\right]\left\{\ \begin{array}{l} \forall l \in S_1 \circ S_4, \exists p \in \mathbb{Z}^2, \\ \sigma(l) = \sigma(p) \end{array}\ \right\} \tag{26}$$

5.3.1 Shrinking the Candidate Payload Set. Let us first simplify the proof goal (26). Observe that the language of S_1 is an infinite set with the structure $\{S_2^n; S_3^n \mid n \in \mathbb{N}\}$, and the payload value *pa* controls the iterations of the two `for`-loops, whose bodies realise statements from S_2 and S_3, respectively. Since the postcondition only requires the existence of *pa* to prove realisability for all elements of S_1, removing all negative values of *pa* will not affect the validity of the triple. We achieve this by applying the Weaken rule from Fig. 10 with $\mathcal{P}_1 = \mathbb{N} \times \mathbb{Z}$, which shrinks the payload set in the postcondition into $\mathbb{N} \times \mathbb{Z}$, and obtain the following subgoals:

$$\left\{\ \not\ast_{i \in \mathbb{Z}^- \times \mathbb{Z}} H_{\mathtt{xy}}\right\}\left[p \in \mathbb{Z}^- \times \mathbb{Z}\ :\ \mathtt{vuln}(p)\right]\{\top\} \tag{27}$$

$$\left\{\ \not\ast_{i \in S_1 \circ S_4 \cup \mathbb{N} \times \mathbb{Z}} H_{\mathtt{xy}}\right\}\left[\begin{array}{ll} \langle pa, pb\rangle \in \mathbb{N} \times \mathbb{Z} : & \texttt{for k in [0:pa] \{...\}} \\ & \texttt{for k in [0:pa] \{...\}} \\ & \texttt{if pb > 10 ... else ...} \\ l \in S_1 \circ S_4 \quad : l & \end{array}\right]\left\{\ \begin{array}{l} \forall l \in S_1 \circ S_4, \exists p \in \mathbb{N} \times \mathbb{Z}, \\ \sigma(l) = \sigma(p) \end{array}\ \right\} \tag{28}$$

The triple (27) can be reduced to a unary one via InfProd (Fig. 4) and proven without much trouble.

5.3.2 Simultaneously Splitting Infinite Sets. Our next observation based on the structure of the vulnerable program is that the payload *pa* controls the first two loops, while *pb* controls the `if`-condition. This dichotomy with the structure of the WM grammar (24), suggests that we can reduce the goal (28) to verifying two sequentially composed sub-programs of `vuln`, "aligned" with the respective sets of programs generated by S_1 and S_4.[6] Intuitively, this decomposition is made possible

[6]If the grammar of L were not structured as favourably, one could replace it by an equivalent L' after proving the equality of the respective languages $[\![L]\!] = [\![L']\!]$, thanks to the domain substitution/reindexing rule inherited by IRL from LGTM [20].

by the fact that the choice of the payload pa for the first component would not affect the choice of pb for the second one. This requirement is captured by the GrmSeq rule given in Fig. 10. GrmSeq reduces the WM realisability proof about a vulnarable program controlled by the product of two payload sets $\mathcal{P}_1, \mathcal{P}_2$, and the language concatenation $L_1 \circ L_2$, to a subgoal about $\mathcal{P}_1$ and L_1 and another about $\mathcal{P}_2$ and L_2. The subgoal about $\mathcal{P}_1$ and L_1 asserts a stronger postcondition than the usual WM realisability triple: a concrete heap predicate R_i has to be supplied, and the postcondition is a septegral of R_i. Moreover, such septegral needs to imply the realisability condition (*i.e.*, the side condition in the middle), where for each $l \in L_1$, the existence of the corresponding payload is indicated by a surjective *choice oracle* τ.[7] For the subgoal about $\mathcal{P}_2$ and L_2, its precondition is parameterised by elements in L_1, matching the intuition of independent choices between L_1 and L_2.

Let R^n_{xy} denote $\mathsf{x} \mapsto (xv + n) * \mathsf{y} \mapsto (yv + n)$, and let κ be the function from $S_1 \cup \mathbb{N}$ to $\mathbb{N}$ such that for any $l = S^n_2; S^n_3 \in S_1$ (recall the observation from Sec. 5.3.1), $\kappa(l) = n$ and for any $n \in \mathbb{N}$, $\kappa(n) = n$. After applying the GrmSeq rule with R_i being $R^{\kappa(i)}_{\mathsf{xy}}$ and τ being κ, we reduce the triple (28) to the following two to show (the septegral implication side condition is straightforward and thus elided):

$$\left\{ \circledast_{i \in S_1 \cup \mathbb{N}} H_{\mathsf{xy}} \right\} \left[\begin{array}{l} pa \in \mathbb{N} \,:\, \texttt{for k in [0:pa] \{...\}; for k in [0:pa] \{...\}} \\ l \in S_1 \;\; : \, l \end{array} \right] \left\{ \circledast_{i \in S_1 \cup \mathbb{N}} R^{\kappa(i)}_{\mathsf{xy}} \right\} \tag{29}$$

$$\forall j \in S_1, \left\{ \circledast_{i \in S_4 \cup \mathbb{Z}} R^{\kappa(j)}_{\mathsf{xy}} \right\} \left[\begin{array}{l} pb \in \mathbb{Z} \,:\, \texttt{if pb > 10 ... else ...} \\ l \in S_4 \;\; : \, l \end{array} \right] \left\{ \begin{array}{l} \forall l \in S_4, \exists p \in \mathbb{Z}, \\ \sigma(l) = \sigma(p) \end{array} \right\} \tag{30}$$

5.3.3 Index Sets as Infinite Disjointing Unions. To prove (29), we notice that the structure $\{S^n_2; S^n_3 \mid n \in \mathbb{N}\}$ of S_1 matches the `for`-loops in the program that control the repetition of statement `x = x + 1` and `y = y + 1`. We need to align loop iterations with the repetitions of S_2 and S_3, respectively. To achieve this, we use the InfInfProd rule (Fig. 4) to split the infinite index set $S_1 \cup \mathbb{N}$ into an infinite union of disjoint subsets over $\mathbb{N}$ with each j-index subset being $\left\{S^j_2; S^j_3, j\right\}$. By first applying the consequence rule to change the goal's postcondition into $\forall j \in \mathbb{N}, \sigma(S^j_2; S^j_3) = \sigma(j)$ and then applying InfInfProd in the way above, we obtain the following subgoal:

$$\forall j \in \mathbb{N}, \left\{ H_{\mathsf{xy}}(j) * H_{\mathsf{xy}}(S^j_2; S^j_3) \right\} \left[j : \texttt{for k in [0:j] ...}, \quad l \in S^j_2; S^j_3 : l \right] \{\sigma(l) = \sigma(j)\} \tag{31}$$

which can be proved by doing induction on j and using existing rules of IRL (details omitted).

5.3.4 Non-Disjoint Partitioning of Infinite Index Sets. Having proven the triple (29), we move on to verifying (30). To do so, let us notice that the interpretation of S_4 is a choice between S_5 and S_6, with the payload variable pb controlling the `if`-branch in the vulnerable program. Therefore, we aim to split the set of programs in S_4, while preserving the space of available payloads pb. This requires partitioning the infinite set $\mathbb{Z} \cup S_4$ into $\mathbb{Z} \cup S_5$ and $\mathbb{Z} \cup S_6$. This is because, intuitively, if the disjoint subsets of the language S_4 are each realisable under the same payload space, then S_4 itself is realisable under that payload space. To capture this intuition, we provide the derived rule, GrmDisj (Fig. 10), which partitions the language component of the index set into two pairwise distinct sets while keeping the payload space unchanged. Note that its derivation involves the Expand rule (*cf.* Sec. 2.3) for duplicating $\mathcal{P}$ into side conditions. Denoting $\kappa(j)$ as n, we apply it to (30), obtaining two identical triples for the languages of S_i, where $i \in \{5, 6\}$:

$$\left\{ \circledast_{i \in S_i \cup \mathbb{Z}} R^n_{\mathsf{xy}} \right\} \left[\begin{array}{l} pb \in \mathbb{Z} \,:\, \texttt{if pb > 10 ... else ...} \\ l \in S_i \;\; : \, l \end{array} \right] \left\{ \begin{array}{l} \forall l \in S_i, \exists p \in \mathbb{Z}, \\ \sigma(l) = \sigma(p) \end{array} \right\} \tag{32}$$

[7] Requiring a payload-choosing function and its surjectivity simplifies its presentation as well as the proofs of itself and those relying on it. For example, without surjectivity, GrmSeq needs an extra side condition like the first triple in Weaken.

Once the language of the Weird Machine set is reduced to a single instruction, *i.e.*, the terminal symbols in S_5 and S_6, the postcondition in each version of (32) can be further simplified from, *e.g.*, $\forall l \in S_5, \exists p \in \mathbb{Z}, \sigma(l) = \sigma(p)$ to $\exists p \in \mathbb{Z}, \sigma(\mathsf{y} := \mathsf{x} + \mathsf{y}) = \sigma(p)$. The final step is to determine the value of pb that proves the triple by instantiating the respective existential quantifier. At the moment, we do so manually in our proof, but this task can also be delegated to an SMT solver. Once the value of pb is fixed, the Payload rule (Fig. 10) aligns the language with this specific run of the vulnerable program and drops the other runs of the program out of the triple. For example, taking (32) for S_5, we apply Payload with $pb = 20$ obtaining the following two triples to prove:

$$\left\{ \not\Diamond_{i \in S_5 \cup (\mathbb{Z} \setminus \{20\})} R^n_{\mathsf{xy}} \right\} [pb \in \mathbb{Z} \setminus \{20\} : \texttt{if pb > 10 ... else ...}, \quad l \in S_5 : l] \, \{\top\} \tag{33}$$

$$\left\{ \not\Diamond_{i \in S_5 \cup \{20\}} R^n_{\mathsf{xy}} \right\} [20 : \texttt{if pb > 10 ... else ...}, \quad l \in S_5 : l] \, \{\sigma(\mathsf{y} := \mathsf{x} + \mathsf{y}) = \sigma(20)\} \tag{34}$$

The proof of (33) is similar to (27). The triple (34) is provable using Focus rule, as mentioned in Sec. 2.2, while the similar triple for S_6 can be proven by setting pb to any value that makes the **if** condition false. This completes the proof sketch for the triple (25) stating the realisability of the WM L over the vulnerable program `vuln`. The complete proof can be found in our Lean development.

Why IRL? One may wonder that Eq. 25 can also be expressed as an equivalent binary Hoare triple, like $\forall l. \exists p. \{\cdots\} \, [\mathtt{vuln}(p); l] \, \{\sigma(l) = \sigma(p)\}$. However, under this interpretation, the binary formulation does not yield a proof system for WM realisability, but still quantifies over all programs ($\forall l[\in L]$). Establishing it therefore requires a mechanism for reasoning about all l. Any reasonable approach (e.g., structural induction on the syntax of L) necessarily introduces structural rules analogous to GrmSeq, which, as Fig. 10 makes clear, constitute the technically most involved part of the system. The remaining rules (Weaken, GrmDisj, Payload) arise from internalising the $\forall$-$\exists$ quantifier structure from the IRL postcondition. Since the binary formulation also requires the core structural rules, it would need bespoke inductive reasoning outside of the logic, which would not provide a more streamlined proof. Therefore, the infinitary aspect of IRL is not an unnecessary detour, but a direct consequence of internalising the quantifier structure into the specification.

6 Related Work

Relational Program Logics. Our work extends a long line of research efforts on program logics for proving relational safety properties of many programs executed independently. Benton's pioneering work on Hoare-style relational program logics only considered properties of pairs of programs, providing rules to align their individual steps in order to prove the relations on their resulting states [5]. This approach has later been extended to support SL-style reasoning and used to verify correctness of a pragmatic algorithm for garbage collection [47] and to reason about the absence of information flow leaks in higher-order heap-manipulating programs [34].

An important step towards extending the utility of relational logics has been made by the work by Sousa and Dillig, who introduced Cartesian Hoare Logic (CHL)—a relational logic capable of expressing k-safety for $k > 2$ specifications and verifying them automatically by reducing the k-safety triples to unary Hoare triples [44]. This work has been further improved upon by D'Osualdo et al., who proposed the Logic for Hyper-triple Composition (LHC) [16] that addressed the inability of CHL to compose relational triples of *different* arity in the same proof. Curiously, LHC also features a version of Merge rule, which, however, works in the *opposite* direction *w.r.t.* to the Merge rule of IRL. Specifically, the rule wp-idx-merge rule of LHC [16, Fig. 9] *increases* the arity of a relational triple to verify, by removing an i-indexed program component from the conclusion, and replacing, in the generated proof obligation, with two copies, indexed respectively i and j, provided the postcondition is true under swapping i and j. The Merge rule of IRL (Fig. 6) has exactly the opposite effect: it replaces a (typically larger) set of indices S_2 in the conclusion by

a (typically smaller) set S_2' in the premise, so that the residual proof goal features a relational triple of a "smaller" cardinality. We are not aware of any other relational logic that offers a similar rule.

IRL is a generalisation of LGTM [20], and we have discussed the delta between the two formalisms extensively throughout the paper: in a nutshell, it boils down to (a) the new InfProd, InfInfProd, and Merge rules with the novel notion of septegral used by both, (b) generalisation of the state model from a finite partial map in LGTM to a function from an index set to finite partial maps in IRL, and (c) a restricted semantics of individual programs in IRL, controlling the spread of non-determinism caused by memory allocation and required for soundness of the Merge rule.

A concurrent line of work on Hyper Hoare Logic (HHL) [12, 13] addresses a related but different challenge: reasoning about hyper-properties of possibly infinitely many runs of the *same* program. Unlike IRL, HHL allows one to express not just over-approximation of the execution outcomes (*i.e.*, the traditional notion of hyper-safety), but also *under-approximation* in the style of Incorrectness Logic [35], as well as other hyperproperties. While HHL can in theory be used to encode the example (2) from Sec. 1 by rewriting the triple as one program with a variable n, it cannot be used to encode example (4) or to state Weird Machine Realisability (Sec. 5) as those triples contain different programs, while HHL (and its rules) only support reasoning over multiple executions of the same program. It is not clear whether rules for reasoning about families of syntactically different programs can be easily derived from the basic HHL rules. Hyper Separation Logic [22] extends HHL with a hyper separating conjunction and a generalised frame rule, thus enabling reasoning about mutable heaps with pointers, but it inherits HHL's single-program triples and, therefore, cannot relate executions of infinite families of syntactically different programs, as required by the specifications of the array-manipulating algorithms we have tackled in Sec. 2 via IRL.

The RHLE program logic allows one to prove ∀∃ properties between finite families of programs [15]. While one could attempt to use RHLE for stating WM realisability as a RHLE triple, which is also a ∀∃ property (Definition 5.1), it does not allow for relating arbitrary infinite sets of programs indexed by a grammar, and lacks rules following the structure of the index set (such as our derived GrmDisj and GrmSeq), making it impossible to conduct WMR proofs in the logic itself.

Unrealizability Logic (UL) [28] allows one to prove relational properties of infinitely many programs. In contrast with IRL, UL does not allow one to state properties of arbitrary families of programs (*e.g.*, indexed by elements of $\mathbb{R}$); instead it requires those programs to be generated by a context-free grammar. Furthermore, UL assertions do not allow for arbitrary combinations of different components of hyper-state: each program's pre/post-states are constrained in isolation from the others. UL proof rules are significantly different from those of other relational logics as they are tailored for proving unrealisability of program synthesis tasks [33]. While UL-based reasoning is close in spirit to our proofs in Sec. 5, we don't think it could be immediately applicable to establish Weird Machine Realisability, which inherently relates outcomes of two (rather differently represented) families of programs. We believe one can derive the rules of UL in IRL similarly to how we derived rules for WMR in Sec. 5, but we did not carry out this exercise in this work.

Determinism in Separation Logic. One of the key technical challenges we had to overcome is to choose the semantics for the IRL language that would allow all useful reasoning principles expected of Separation Logic, while making it possible to prove the soundness of our new rules, in particular, Merge. It has been long known in the SL community that, in order to allow for a useful and sound Frame rule, memory allocation should be modelled non-deterministically [48] or the semantics of the triple should quantify over all possible frames [6]—both these options being unsuitable for proving soundness of Merge. Our final design (Sec. 4.2) is close in spirit to the work described in Baktiev's MSc thesis [4] that establishes soundness of the Frame rule in a deterministic language

without pointer arithmetics and for heap assertions that are unaffected by address permutations [4]. Such restrictions are in effect in our language and are enforced in it syntactically.

A relatively new line of work on the Outcome Logic proposes a clean approach to separate semantic modelling and specification of effects (such as state manipulation, probability, and non-determinism) from state (un)reachability [50]. In particular, OSL, a recent generalisation of Outcome Logic to Separation Logic [51] shows how to elegantly express different semantics to Separation Logic in the presence of allocation, accounting for both a deterministic and a non-deterministic version of a memory allocator. That said, OSL does not provide a recipe to combine determinism and the Frame rule in a convenient way, which we could immediately adopt for our purposes.

Formal Reasoning about Weird Machines. At the time of this writing, relatively small body of work has been concerned with formal reasoning about Weird Machines [8] or quantifying exploitability of vulnerable programs. Most of the efforts focused predominantly on proposing approaches for efficient automated generation of individual exploits [3, 11, 24, 26, 40, 43], with an exception of the recent work by Ling et al. who proposed to synthesise a language of exploits for a vulnerable program [30], albeit, without providing a formal correctness argument for their approach.

Paykin et al. [39] suggested to consider the existence of a Weird Machine as an instance of insecure compilation [1, 37]—meaning that there are contexts in the target language whose interactions with the compiled vulnerable program cannot be represented by any source-level context. Unlike our WMR statements, Paykin et al.'s approach does not allow one to characterise the set of behaviours exhibited by a vulnerable program—only to show that such behaviours exist. Closer to our approach is the recent work by Lesani, who proposed a type-and-effect system that ascribes a an over-approximation of the sets of a Weird Machine behaviours to a vulnerable program, in a form of a regular expression [29]. In comparison with our approach, Lesani's type system does not allow for reasoning about attack families that are expressible as a context-free grammar. More conceptually, Lesani tackles the problem from the perspective of an *over-approximation* of possible Weird Machine behaviours (his type system is sound but not complete), whereas our approach from Sec. 5 considers an *under-approximation*, meaning that *all* exploits in a proven triple are realisable, but the triple is not guaranteed to capture the entire space of possibly realisable program exploits.

7 Conclusion

We presented IRL: the first Hoare-style separation logic for mechanically verifying hypersafety properties of uncountably infinite program families, and showcased its reasoning principles by specifying and verifying several intricate implementations of algorithms that manipulate discrete encodings of continuous data. We further demonstrated that IRL's infinitary reasoning enables a novel, logic-based methodology for proving Weird Machine Realisability (Sec. 5): by internalising the ∀-∃ quantifier structure of WMR into relational post-conditions, we derived domain-specific proof rules that decompose realisability proofs along the grammatical structure of the exploit language, providing the first deductive proof system for this class of security properties.

All results of our work: the logic, its meta-theory, and the case studies, have been mechanised in Lean, and we are going to make them available for experiments by the community. We believe our work opens exciting opportunities for (a) combining reasoning about specification-level continuous functions and discrete data manipulations in the same unified framework, (b) exploring synergies between relational and non-relational proofs to streamline reasoning about complex imperative programs, as we have shown in Sec. 3, and (c) applying deductive program logic to further security analyses that require reasoning about infinite families of program behaviours, as shown in Sec. 5.

A Consolidated Summary of IRL Proof Rules

For ease of reference, Fig. 11 collects the core proof rules of IRL in one place, reproduced verbatim from their respective figures in the main text: sequencing and focusing rules, adopted from LGTM [20] (*cf.* Fig. 3), iteration rules (*cf.* Fig. 5 and Fig. 8), and the rules for (in)finite products and merging that are specific to IRL (*cf.* Fig. 4 and Fig. 6). The Expand rule, used in Sec. 5, is the converse of Merge, obtained by swapping its two triples; the rules for Weird Machine Realisability, *derived* from the rules below, are collected in Fig. 10. The correspondence between the rule names and the Lean lemmas in the accompanying artefact [21] is documented in the artefact's README.

$$\textsc{SeqU1}\ \frac{\{\ P\ \}\ [\iota :\ p_1]\ \{\ x\ \mid\ H\ \} \qquad \{\ H\ \}\ [\iota :\ p_1',\ S :\ \mathcal{P}_2]\ \{\ x, \overline{z}\ \mid Q(\overline{z})\ \}}{\{\ P\ \}\ [\iota : p_1; p_1',\ S : \mathcal{P}_2]\ \{\ x, \overline{y}\ \mid Q(x\overline{y})\ \}} \qquad \textsc{SeqU2}\ \frac{\{P\}\ [\iota :\ p_1,\ S :\ \mathcal{P}_2]\ \{\ x, \overline{z}\ \mid H(\overline{z})\ \} \qquad \forall \overline{z}, \{\ H(\overline{z})\ \}\ [\iota :\ p_1']\ \{\ x\ \mid\ Q(x\overline{z})\ \}}{\{\ P\ \}\ [\iota : p_1; p_1',\ S :\ \mathcal{P}_2]\ \{\ x_1, \overline{x_2}\ \mid Q(x_1\overline{x_2})\ \}}$$

$$\textsc{LetU}\ \frac{\{P\}\ [\iota :\ p_1,\ S :\ \mathcal{P}_2]\ \{\ x, \overline{z}\ \mid H(x, \overline{z})\ \} \qquad \forall x, \overline{z}, \{\ H(x, \overline{z})\ \}\ [\iota :\ p_1'(x)]\ \{\ x\ \mid\ Q(x\overline{z})\ \}}{\{\ P\ \}\ [\iota : \texttt{let x} = p_1\ \texttt{in}\ p_1'(x),\ S : \mathcal{P}_2]\ \{\ x, \overline{y}\ \mid Q(x\overline{y})\ \}}$$

$$\textsc{Focus}\ \frac{S = S_1 \uplus S_2 \quad \{\ P\ \}\ [S_1 :\ \mathcal{P}_1]\ \{\ \overline{x}\ \mid\ H(\overline{x})\ \} \quad \forall \overline{x},\ \{\ H(\overline{x})\ \}\ [S_2 :\ \mathcal{P}_2]\ \{\ \overline{y}\ \mid\ Q(\overline{x} \cdot \overline{y})\ \}}{\{\ P\ \}\ [S :\ \mathcal{P}_1 \uplus \mathcal{P}_2]\ \{\ \overline{z}\ \mid\ Q(\overline{z})\ \}}$$

$$\textsc{For}\ \frac{S = \bigcup_{i=0}^{N-1} S_i \quad \forall \overline{x}, i < N, \{\ I(i, \overline{x})\}\ [\iota : p,\ S_i : \mathcal{P}_i]\ \{\ z, \overline{y}\ \mid I(i+1, \overline{x} \cup \overline{y})\ \}}{\{I(0, \overline{\emptyset})\}\ [\iota : \texttt{for}\ i\ \texttt{in}\ \texttt{range}(0, N)\ \{p\},\ S : \mathcal{P}]\ \{\ z, \overline{y}\ \mid I(N, \overline{y})\ \}}$$

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

$$\textsc{InfProd}\ \frac{\forall i, \{\ \lceil i \in S \rceil\ *\ P_i\ \}\ [\mathcal{P}(i)]\ \{\ x\ \mid\ Q_i(x)\ \}}{\left\{\ \circledast_{i \in S} P_i\ \right\}\ [S :\ \mathcal{P}]\ \left\{\ \overline{x}\ \mid\ \circledast_{i \in S} Q_i(\overline{x}(i))\ \right\}} \qquad \textsc{InfInfProd}\ \frac{\forall j \in \mathcal{I}, \left\{\ \circledast_{i \in S_j} P_i\ \right\}\ [S_j :\ \mathcal{P}]\ \{\ \overline{x}\ \mid\ Q_j(\overline{x})\ \}}{\left\{\ \circledast_{i \in \biguplus_{j \in \mathcal{I}} S_j} P_i\ \right\}\ \left[\biguplus_{j \in \mathcal{I}} S_j :\ \mathcal{P}\right]\ \{\ \overline{x}\ \mid\ \forall j \in \mathcal{I}, Q_j(\overline{x})\ \}}$$

$$\textsc{Merge}\ \frac{\begin{array}{c}\texttt{local}(\{P_1, Q\}, S_1) \quad \mu : S_2 \to S_2' \text{ is surjective} \\ \forall i \in S_2, \mathcal{P}_2(i) = \mathcal{P}_2'(\mu(i)) \end{array} \qquad \left\{P_1 * \circledast_{S_2'} P_2\right\}\left[\begin{array}{c} S_1 : \mathcal{P}_1 \\ S_2' : \mathcal{P}_2' \end{array}\right]\ \{\overline{x}, \overline{y} \mid Q(\overline{x}, \overline{y} \circ \mu)\ \}}{\left\{P_1 * \circledast_{S_2} P_2\right\}\left[\begin{array}{c} S_1 : \mathcal{P}_1 \\ S_2 : \mathcal{P}_2 \end{array}\right]\ \{\overline{x}, \overline{y}\ \mid Q(\overline{x}, \overline{y})\ \}}$$

Fig. 11. Consolidated summary of the core IRL proof rules: sequencing and focusing (SeqU1, SeqU2, LetU, Focus), iteration (For, While), and (in)finite products and merging (InfProd, InfInfProd, Merge). Side conditions in InfInfProd about Q are omitted for brevity.

Acknowledgments

We are grateful to Thibault Dardinier for his comments on a draft of this paper. We thank the reviewers of OOPSLA'26 for their feedback and, no less, for their verdict. Earlier versions of this work were reviewed, with considerable enthusiasm and ultimately fatal reservations, at PLDI'25, POPL'26, and PLDI'26. We are sincerely grateful to those anonymous reviewers as well: their requests for a comparison with unary program logics and for more diverse and realistic applications gave rise to Sec. 3 and Sec. 5, and thus to a substantially better paper, if a noticeably later one.

Data Availability

The software artefact accompanying this paper is available on Zenodo [21]. The artefact contains the source code and build scripts for the Lean-embedded formalisation of IRL as well as the collection of case studies that can be used to reproduce the results described in Sec. 2, Sec. 3, and Sec. 5. Our Lean implementation of the unary Separation Logic used in Sec. 3 is also available as a standalone open-source repository [19].

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *CSF*. IEEE, 256–271. doi:10.1109/CSF.2019.00025

[2] Andrew W. Appel. 2022. Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk). In *CPP*. ACM, 2–11. doi:10.1145/3497775.3503951

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84. doi:10.1145/2560217.2560219

[4] Murat Baktiev. 2006. *Permutation Semantics of Separation Logic.* Master's thesis. Saarland University. Available at: https://www.ps.uni-saarland.de/Publications/documents/baktiev2006.pdf.

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

[6] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. 2008. A Simple Model of Separation Logic for Higher-Order Store. In *ICALP (LNCS, Vol. 5126)*. Springer, 348–360. doi:10.1007/978-3-540-70583-3_29

[7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*. ACM, 30–40. doi:10.1145/1966913.1966919

[8] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. 2011. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation. *login Usenix Mag.* 36, 6 (2011). https://www.usenix.org/publications/login/december-2011-volume-36-number-6/exploit-programming-buffer-overflows-weird

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

[10] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1 (2023), 5:1–5:43. doi:10.1145/3579834

[11] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *ACM Trans. Priv. Secur.* 24, 4 (2021), 26:1–26:36. doi:10.1145/3462699

[12] Thibault Dardinier, Anqi Li, and Peter Müller. 2024. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 316 (2024), 30 pages. doi:10.1145/3689756

[13] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1485–1509. doi:10.1145/3656437

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE (LNCS, Vol. 9195)*. Springer, 378–388. doi:10.1007/978-3-319-21401-6_26

[15] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In *APLAS (LNCS, Vol. 13658)*. Springer, 67–87. doi:10.1007/978-3-031-21037-2_4

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

[17] Thomas Dullien. 2020. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Trans. Emerg. Top. Comput.* 8, 2 (2020), 391–403. doi:10.1109/TETC.2017.2785299

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247. doi:10.1145/155090.155113

[19] Vladimir Gladshtein, Sean Wang, Qiyuan Zhao, and Ilya Sergey. 2024. SPLean: Separation Logic Proofs in Lean. Available at https://github.com/verse-lab/splean.

[20] Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman P. Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI (2024), 647–670. doi:10.1145/3656403

[21] Vladimir Gladshtein, Qiyuan Zhao, Yuxi Ling, Sean Wang, and Ilya Sergey. 2026. Artifact for "Infinitary Relational Logic". doi:10.5281/zenodo.21787894 The code is available at https://github.com/verse-lab/irl.

[22] Trayan Gospodinov, Peter Müller, and Thibault Dardinier. 2026. Hyper Separation Logic. *Proc. ACM Program. Lang.* 10, PLDI (2026), 250:1–250:24. doi:10.1145/3808328

[23] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. ACM, 282–293. doi:10.1145/512529.512563

[24] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *USENIX Security Symposium*. USENIX Association, 177–192. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[25] Eugene Isaacson. 1989. Numerical Recipes in C: The Art of Scientific Computing. *SIAM Rev.* 31, 1 (1989), 142. doi:10.1137/1031025

[26] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *CCS*. ACM, 1868–1882. doi:10.1145/3243734.3243739

[27] 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, 36–43. doi:10.1109/ARITH58626.2023.00021

[28] Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. 2023. Unrealizability Logic. *Proc. ACM Program. Lang.* 7, POPL (2023), 659–688. doi:10.1145/3571216

[29] Mohsen Lesani. 2024. Vulnerability Flow Type Systems. In *IEEE Security and Privacy Workshops*. IEEE, 157–168. doi:10.1109/SPW63631.2024.00020

[30] Yuxi Ling, Gokul Rajiv, Kiran Gopinathan, and Ilya Sergey. 2025. Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis. In *USENIX Security Symposium*. USENIX Association, 413–429. https://www.usenix.org/conference/usenixsecurity25/presentation/ling

[31] The mathlib Community. 2020. The Lean mathematical library. In *CPP*. ACM, 367–381. doi:10.1145/3372885.3373824 https://github.com/leanprover-community/mathlib4.

[32] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2022. NeRF: representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2022), 99–106. doi:10.1145/3503250

[33] Shaan Nagy, Jinwoo Kim, Thomas Reps, and Loris D'Antoni. 2024. Automating Unrealizability Logic: Hoare-Style Proof Synthesis for Infinite Sets of Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 275 (Oct. 2024), 27 pages. doi:10.1145/3689715

[34] 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. doi:10.1109/SP.2011.12

[35] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. doi:10.1145/3371078

[36] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. doi:10.1007/3-540-44802-0_1

[37] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *CSF*. IEEE Computer Society, 392–404. doi:10.1109/CSF.2017.13

[38] Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Bövinga, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. 2024. Metaprogramming in Lean 4. Available at https://leanprover-community.github.io/lean4-metaprogramming-book/.

[39] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, Alexander Bakst, and Scott Moore. 2019. Weird Machines as Insecure Compilation. *CoRR* abs/1911.00157 (2019). http://arxiv.org/abs/1911.00157

[40] Jannik Pewny, Philipp Koppe, and Thorsten Holz. 2019. STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming. In *EuroS&P*. IEEE, 111–126. doi:10.1109/EUROSP.2019.00018

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

[42] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1 (2012), 2:1–2:34. doi:10.1145/2133375.2133377

[43] Edward J. Schwartz, Cory F. Cohen, Jeffrey Gennari, and Stephanie Schwartz. 2020. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *CCS*. ACM, 1789–1801. doi:10.1145/3372297.3417234

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

[45] Julien Vanegue. 2014. The Weird Machines in Proof-Carrying Code. In *IEEE Security and Privacy Workshops*. IEEE Computer Society, 209–213. doi:10.1109/SPW.2014.37

[46] Jaeyeon Won, Willow Ahrens, Teodoro Fields Collin, Joel S. Emer, and Saman Amarasinghe. 2025. The Continuous Tensor Abstraction: Where Indices Are Real. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025). doi:10.1145/3763146

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

[48] Hongseok Yang and Peter W. O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *FOSSACS (LNCS, Vol. 2303)*. Springer, 402–416. doi:10.1007/3-540-45931-6_28

[49] Qiyuan Zhao, George Pîrlea, Zhendong Ang, Umang Mathur, and Ilya Sergey. 2024. Rooting for Efficiency: Mechanised Reasoning about Array-Based Trees in Separation Logic. In *CPP*. ACM, 45–59. doi:10.1145/3636501.3636944

[50] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 522–550. doi:10.1145/3586045

[51] Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 276–304. doi:10.1145/3649821

Received 2026-03-15; accepted 2026-08-06