Proving Hypersafety Compositionally

EMANUELE D'OSUALDO, MPI-SWS, Germany
AZADEH FARZAN, University of Toronto, Canada
DEREK DREYER, MPI-SWS, Germany

Hypersafety properties of arity n are program properties that relate n traces of a program (or, more generally, traces of n programs). Classic examples include determinism, idempotence, and associativity. A number of *relational program logics* have been introduced to target this class of properties. Their aim is to construct simpler proofs by capitalizing on structural similarities between the n related programs. We propose unexplored, complementary proof principles that establish hyper-triples (i.e. hypersafety judgments) as a unifying compositional building block for proofs, and we use them to develop a *Logic for Hyper-triple Composition* (LHC), which supports forms of proof compositionality that were not achievable in previous relational logics. We prove LHC sound and apply it to a number of challenging examples.

CCS Concepts: • **General and reference** → Verification; • **Software and its engineering** → *Formal methods*; • **Theory of computation** → **Hoare logic**; **Logic and verification**; **Program reasoning**.

Additional Key Words and Phrases: Hyperproperties, Modularity, Compositionality, Weakest Precondition

ACM Reference Format:
Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving Hypersafety Compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 135 (October 2022), 26 pages. https://doi.org/10.1145/3563298

1 INTRODUCTION

Many properties of interest about programs are properties not of individual program traces but rather of multiple program traces. For example, stipulating a bound on mean response time over all executions of a program cannot be specified as a property of individual traces, because the acceptability of delays in a trace depends on the magnitude of delays in all other traces. Clarkson and Schneider [2008] formally studied this class of properties, coining the term *hyperproperties*. In this paper, we focus on the verification problem of a class of generalized hyperproperties, which we will refer to as *n-safety properties*: these are *safety* hyperproperties (i.e. only concerned with partial correctness) that govern n executions of *potentially different* programs. More formally, these n-safety properties have the form $\forall(s_1, s_1') \in [\![t_1]\!] \ldots \forall(s_n, s_n') \in [\![t_n]\!].\, \varphi(s_1, s_1', \ldots, s_n, s_n')$, where $[\![t]\!]$ denotes the set of input/output states of the traces of t. Examples of such properties are[1] commutativity ($n = 2$), associativity ($n = 4$), determinism ($n = 2$), noninterference ($n = 2$) and transitivity ($n = 3$). Input-output equivalences between two programs can also be proved as 2-safety properties (e.g. a compiler optimization preserves I/O behaviour).

One approach to proving n-safety properties is to obtain from t a precise mathematical characterization R of its functionality $[\![t]\!]$—i.e. t's strongest postcondition—and then prove using this

[1] Intuitively, for instance, associativity $\mathtt{f}(a, \mathtt{f}(b, c)) = \mathtt{f}(\mathtt{f}(a, b), c)$ compares 4 runs of $\mathtt{f}$.

Authors' addresses: Emanuele D'Osualdo, dosualdo@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Azadeh Farzan, azadeh@cs.toronto.edu, University of Toronto, Toronto, Canada; Derek Dreyer, dreyer@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany.

2475-1421/2022/10-ART135
https://doi.org/10.1145/3563298

mathematical characterization that the desired n-safety property φ is satisfied. This approach has a major drawback, however: functional correctness is in general a much stronger—thus more difficult to prove—property of t than what proving φ requires.

Thus, the trend in research on this problem has been instead towards reasoning *directly* about n-safety, through a number of so-called *relational program logics*, e.g. [Barthe et al. 2016; Benton 2004; Sousa and Dillig 2016; Yang 2007]. These logics move away from the traditional Hoare-triple judgment and introduce n-ary relational Hoare-style judgments, which we will refer to as *hyper-triples*. In our syntax, a hyper-triple (of arity n) is a judgment of the form

$$\vdash \{P\}\ [1\colon t_1, \ldots, n\colon t_n]\ \{Q\}$$

which means $\forall (s_1, s_1') \in [\![t_1]\!] \ldots \forall (s_n, s_n') \in [\![t_n]\!].\, P(s_1, \ldots, s_n) \Rightarrow Q(s_1', \ldots, s_n')$. For example, determinism of t can be expressed as $\vdash \{\mathtt{vars}_1 = \mathtt{vars}_2\}\ [1\colon t, 2\colon t]\ \{\mathtt{vars}_1 = \mathtt{vars}_2\}$, which asserts that if t is run from two states which agree on the values of some relevant variables `vars`, then, if the two runs terminate, the output states will agree on the values of `vars` as well.

The key idea underlying relational program logics is that, by *exploiting the similarity in the program structure* of $t_1, \ldots, t_n$, they avoid the need to specify and prove functional specifications for these programs, thus reducing the complexity of the work needed to prove the n-safety property. In particular, a recurring motif of these logics is the idea of "lockstep" proof rules, which decompose all the programs in the hyper-triple at the same time in the same way according to a common structure. For example, a binary lockstep rule for sequential composition might look as follows:

$$\textsc{LockStepSeq}\quad \frac{\vdash \{P\}\ [1\colon t_1, 2\colon t_2]\ \{P'\} \qquad \vdash \{P'\}\ [1\colon t_1', 2\colon t_2']\ \{Q\}}{\vdash \{P\}\ [1\colon (t_1\,;t_1'), 2\colon (t_2\,;t_2')]\ \{Q\}}$$

This lockstep rule is advantageous when the effects of t_1 and t_2 (and of t_1' and t_2' respectively) are correlated, so that the intermediate assertion P' does not need to reveal their individual effect, but only the relation between their effects. In this fortunate case, the rule logically *aligns* the two traces and propagates simple relational assertions, like $\mathtt{vars}_1 = \mathtt{vars}_2$, through the aligned pair of traces. For instance, it allows one to prove determinism of $(t\,;t')$ by proving determinism of t and of t'.

On their own, however, lockstep rules are too rigid: in general, such a perfect alignment might not be attainable. They are also too fragile: small syntactic differences between the two programs may make them inapplicable. Much of the effort in previous work has thus focused on overcoming this rigidity of lockstep rules. In particular, the state-of-the-art solution to the problem is to extend the logic with rules that replace terms with other, semantically equivalent, terms in a hyper-triple (see *e.g.*, [Barthe et al. 2011, 2017; Sousa and Dillig 2016]). The replacement can bring back the syntactic similarity needed to resume a lockstep proof. The resulting proof strategy, which we call *enhanced lockstep*, is to first rewrite the terms so that their structure reflects the desired alignment, and then proceed with a lockstep derivation.

Despite its simplicity, enhanced lockstep reasoning has produced relational program logics that have already been deployed successfully in a variety of applications. For example [Barthe et al. 2016, 2017] use them for translation validation tasks and even verification of cryptographic algorithms. Sousa and Dillig [2016] demonstrate how higher-arity hypersafety can encode correctness requirements of comparators for user-defined data structures.

Here, however, we would like to call attention to an orthogonal, unexplored, and very limiting dimension of rigidity in lockstep-based relational proofs. Consider for instance a MapReduce library. We ought to be able to prove that this library ensures the high-level guarantee of determinism for its operations, *provided that* the user-supplied parameters to the library—*i.e.*, the lower-level operations on which it depends—are deterministic, associative and commutative. Seeing as the latter

properties are all hyper-safety properties, it would be desirable to be able to take the heterogeneous collection of hyper-triples (of different arities!) representing the assumptions on the user-supplied parameters, and produce from them a determinism hyper-triple for the MapReduce implementation.

Unfortunately, existing relational logics do not support hyper-triple composition in this way. It is easy to observe this at a shallow level: these logics do not provide any rules for mixing and matching properties of different arities (i.e. using hyper-triples of one arity to prove hyper-triples of another arity). Even in CHL [Sousa and Dillig 2016], which is parametrized over the arity n of its hyper-triples, a fixed n must be used throughout an entire hyper-triple derivation. In order to compose hypersafety proofs as a library like MapReduce would require, we need more expressive ways of composing hyper-triples than what lockstep-style rules provide. To see the issue concretely, consider the following example.

Example 1.1. Imagine a library provides a binary operation $\mathsf{op}(a, b)$ that has been proven deterministic and commutative, in the form of two hyper-triples:[2]

$$\vdash \{\mathsf{True}\}\ [1\colon \mathsf{r}_1 \coloneqq \mathsf{op}(a,b), 2\colon \mathsf{r}_2 \coloneqq \mathsf{op}(a,b)]\ \{\mathsf{r}_1(1) = \mathsf{r}_2(2)\} \qquad (\textsc{Det}_{\mathsf{op}})$$

$$\vdash \{\mathsf{True}\}\ [1\colon \mathsf{r}_1 \coloneqq \mathsf{op}(a,b), 2\colon \mathsf{r}_2 \coloneqq \mathsf{op}(b,a)]\ \{\mathsf{r}_1(1) = \mathsf{r}_2(2)\} \qquad (\textsc{Comm}_{\mathsf{op}})$$

In assertions, we write $\mathsf{x}(i)$ to refer to the value of the program variable x in the store at index i.

Assuming op does not modify x and y, we ought to be able to use the hyper-triples above in a proof of the following:

$$\vdash \left\{\mathsf{True}\right\}\ \left[\begin{array}{l} 1\colon \mathsf{x} \coloneqq \mathsf{op}(a,b)\,;\mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{x}) \\ 2\colon \mathsf{x} \coloneqq \mathsf{op}(a,b)\,;\mathsf{y} \coloneqq \mathsf{op}(b,a)\,;\mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{y}) \end{array}\right]\ \left\{\mathsf{z}(1) = \mathsf{z}(2)\right\} \qquad (\textsc{Goal}_{\mathsf{op}})$$

The unfortunate surprise is that there is no alignment of the commands of the two components, that can transform the goal into a lockstep proof that reuses ($\textsc{Det}_{\mathsf{op}}$) and ($\textsc{Comm}_{\mathsf{op}}$). No matter which pairs of calls of op we align across the two components, there will always be one call in component 2 that remains unmatched; the abstract specification of op is relational and we have no hyper-triple that specifies the effect of a single call. Yet, the hyper-triple follows semantically from the assumptions.

We argue that enhanced lockstep reasoning is fundamentally incomplete, and does not allow compositions of proofs that are needed for modular reasoning. In this specific example, even though the arities of the hyper-triples match, we lack an appropriate way to compose them. Intuitively, one should be able to prove that the assignment to x in 1 is equivalent to the first assignment in 2, *and* to the second assignment in 2, and put together these two facts to conclude $\mathsf{x}(2) = \mathsf{y}(3)$.

In this paper, we ask more generally: what can and should composition of hypersafety proofs look like? We provide an answer to this question in the form of a new program logic, *Logic for Hyper-triple Composition (LHC)*. LHC enables the reasoning about a given hypersafety property, formally encoded as a hyper-triple, to be composed from subproofs of hyper-triples of potentially different arities, and with richer and more structurally complex means of composition than are afforded by traditional lockstep reasoning.

The key observation behind the design of LHC is that traditional relational rules are of two main varieties: they either hold generically w.r.t. the hyper-terms involved (like the classic Hoare-logic consequence rule), or they decompose the goal by matching on the structure of the related programs, (like the lockstep sequence rule). As we illustrate in Section 2, LHC identifies a *third* missing set of rules which decompose the goal by matching on *the structure of the hyper-term itself*. That is, by viewing hyper-terms as maps from indices to terms, LHC's new rules allow splitting, joining,

[2]Note that op can have non-deterministic side-effects on the state (determinism is restricted to the return value).

and re-indexing of these maps. In so doing, they provide a new dimension of compositionality in hypersafety reasoning, making it possible for the first time to verify a range of interesting examples (such as Example 1.1) that were beyond the reach of prior relational logics.

The remainder of the paper is structured as follows:

- In Section 2, we illustrate the main new reasoning principles of LHC, by means of a series of examples that cannot be handled with previous relational logics.
- In Sections 3 and 4, we formalize the model and formulate the rules in full generality.
- In Section 5, we include a discussion of various features of LHC and contrast it against the previous state of the art.
- In Section 6, we give a high-level survey of the literature in the problem space of hypersafety verification, and in Section 7, we discuss exciting future directions for this problem space.

Additional case studies, omitted details and the soundness proofs can be found in the extended version of this paper [D'Osualdo et al. 2022].

2 OVERVIEW OF LHC

In this section, we introduce the core new reasoning principles of LHC, illustrating them through a series of simple examples. None of these examples can be handled with existing relational logics. More extensive case studies can be found in [D'Osualdo et al. 2022].

2.1 A Weakest-Precondition-Based Calculus

Using LHC, we can overcome the limitations of pure lockstep reasoning—without relying on functional specifications for subprograms, nor meta-level reasoning—by embracing hyper-triples as building blocks for proving other hyper-triples. The first step to achieve this is to move from a calculus based on triples precondition/hyper-term/postcondition, to a calculus based on a weakest-precondition (WP) predicate over hyper-terms. This allows for a minimal and flexible logic.

More formally,[3] our judgments are of the form $R \vdash P$ where both R and P are assertions over hyper-stores, *i.e.*, maps from indices to variable stores. The judgment asserts that $R \Rightarrow P$ holds on every hyper-store. To state properties of hyper-terms, we introduce the WP assertion $\mathbf{wp}\, \boldsymbol{t}\, \{Q\}$ that holds on a hyper-store $\boldsymbol{s}$ if running $\boldsymbol{t}$ from $\boldsymbol{s}$ results in a hyper-store $\boldsymbol{s}'$ which satisfies the assertion Q.[4] A hyper-term is "run" from a hyper-store by running each of its components on the store at the corresponding index. On the indices where the hyper-term is undefined, the store is simply preserved. Hyper-triples $\{P\}\ \boldsymbol{t}\ \{Q\}$ are notation for $P \Rightarrow \mathbf{wp}\, \boldsymbol{t}\, \{Q\}$.

Standard unary laws for weakest preconditions hold for our WP as well; for example, the usual (unary) rule for sequence $\mathbf{wp}\, [1\colon t]\, \{\mathbf{wp}\, [1\colon t']\, \{Q\}\} \dashv\vdash \mathbf{wp}\, [1\colon (t\,;t')]\, \{Q\}$ is valid in LHC. It is easy to provide WP-based versions of lockstep rules. For example, LockStepSeq can be captured by:

$$\frac{}{\mathbf{wp}\, [1\colon t_1, 2\colon t_2]\, \{\mathbf{wp}\, [1\colon t_1', 2\colon t_2']\, \{Q\}\} \dashv\vdash \mathbf{wp}\, [1\colon (t_1\,;t_1'), 2\colon (t_2\,;t_2')]\, \{Q\}}\ \text{WP-SEQ}_2$$

The WP-SEQ$_2$ rule helps us start a proof[5] of the goal (Goal$_{\text{op}}$) of Example 1.1, by aligning the components at the assignments to z, as in Fig. 1. The right-hand premise is an instance of (Det$_{\text{op}}$). To derive the left-hand premise, however, we need genuinely new reasoning principles.

[3] The notation used in this section will be made fully precise in Sections 3 and 4.
[4] In general Q may also predicate over the return values of $\boldsymbol{t}$, but we ignore this for simplicity in this section.
[5] Here and in following derivations, we omit applications of the rule of consequence.

$$\frac{\vdash \mathbf{wp}\left[\begin{array}{l}1\colon \mathsf{x} \coloneqq \mathsf{op}(a,b)\\ 2\colon \mathsf{x} \coloneqq \mathsf{op}(a,b);\mathsf{y} \coloneqq \mathsf{op}(b,a)\end{array}\right]\left\{\begin{array}{l}\mathsf{x}(1)=\mathsf{x}(2)\\ \mathsf{y}(2)=\mathsf{x}(1)\end{array}\right\} \quad \begin{array}{l}\mathsf{x}(1)=\mathsf{x}(2)\\ \mathsf{y}(2)=\mathsf{x}(1)\end{array} \vdash \mathbf{wp}\left[\begin{array}{l}1\colon \mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{x})\\ 2\colon \mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{y})\end{array}\right]\left\{\mathsf{z}(1)=\mathsf{z}(2)\right\}}{\vdash \mathbf{wp}\left[\begin{array}{l}1\colon \mathsf{x} \coloneqq \mathsf{op}(a,b);\mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{x})\\ 2\colon \mathsf{x} \coloneqq \mathsf{op}(a,b);\mathsf{y} \coloneqq \mathsf{op}(b,a);\mathsf{z} \coloneqq \mathsf{op}(\mathsf{x},\mathsf{y})\end{array}\right]\left\{\mathsf{z}(1)=\mathsf{z}(2)\right\}} \quad (1)$$

Fig. 1. A first step in the proof of Example 1.1, aligning the programs at the assignment to z. The right-hand premise is an instance of the determinism assumption ($\textsc{Det}_{\mathsf{op}}$).

2.2 Conjunction and Nesting

What characterises lockstep rules is that they match on the structure of the *program terms* in multiple components simultaneously. The new rules introduced by LHC focus instead on the structure of the *hyper-term* as a map from indices to terms. The question we ask is: which combinations of WPs are induced by operations on hyper-terms as maps? We start with the ones we need to finish the proof of Example 1.1: two rules induced by the union operation. The union $\boldsymbol{t}_1 + \boldsymbol{t}_2$ of two hyper-terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is a hyper-term if $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ coincide, on the indices they have in common. For example $[1\colon t_1, 2\colon t_2] + [2\colon t_2', 3\colon t_3]$ is well-defined if $t_2 = t_2'$. When the two hyper-terms do not have indices in common, the result is disjoint union, which we write $\boldsymbol{t}_1 \cdot \boldsymbol{t}_2$.

Now we can ask, how can we combine two WPs to obtain a WP on the union of their hyper-terms? LHC's answer is the $\textsc{wp-conj}_0$ rule:

$$\textsc{wp-conj}_0 \quad \frac{\mathrm{idx}(Q_1) \subseteq \mathrm{supp}(\boldsymbol{t}_1) \qquad \mathrm{idx}(Q_2) \subseteq \mathrm{supp}(\boldsymbol{t}_2)}{\mathbf{wp}\,\boldsymbol{t}_1\,\{Q_1\} \wedge \mathbf{wp}\,\boldsymbol{t}_2\,\{Q_2\} \vdash \mathbf{wp}\,(\boldsymbol{t}_1 + \boldsymbol{t}_2)\,\{Q_1 \wedge Q_2\}}$$

Read from right to left, the rule states that, to prove that we are in a hyper-state from which $\boldsymbol{t}_1 + \boldsymbol{t}_2$ takes us to $Q_1 \wedge Q_2$, it is sufficient to prove separately that $\boldsymbol{t}_1$ takes us to Q_1 and $\boldsymbol{t}_2$ to Q_2. This is sound as long as the postconditions only predicate over indices pertaining to their corresponding hyper-term, as mandated by the two premises.

LHC proposes a second way of decomposing a WP by seeing its hyper-term as a *disjoint* union, with the $\textsc{wp-nest}_0$ rule:

$$\textsc{wp-nest}_0 \quad \mathbf{wp}\,\boldsymbol{t}_1\,\{\mathbf{wp}\,\boldsymbol{t}_2\,\{Q\}\} \dashv\vdash \mathbf{wp}\,(\boldsymbol{t}_1 \cdot \boldsymbol{t}_2)\,\{Q\}$$

The rule establishes an equivalence between two nested WP on disjoint hyper-terms, and a single one on their union. The idea is that since the semantics of WP preserves the stores at indices not belonging to the WP's hyper-term, the inner WP on the left receives as input on the indices of $\boldsymbol{t}_2$ the initial stores, just like on the right; and it preserves the outputs of $\boldsymbol{t}_1$. The postcondition Q is free to predicate on the indices of both hyper-terms, and applies to the same hyper-stores on both sides of the equivalence.

This rule unlocks a very useful proof pattern. It is very common to have a goal with many components, but wanting to temporarily focus the proof on a subset of the components. The $\textsc{wp-nest}_0$ rule, applied from right to left, would correspond to "shelving" the components in $\boldsymbol{t}_2$, allowing the proof to operate on $\boldsymbol{t}_1$, *e.g.*, in lockstep. When $\boldsymbol{t}_2$ becomes relevant again, applying the rule from left to right would "unshelve" the components for the rest of the proof.

The seemingly simple $\textsc{wp-nest}_0$ and $\textsc{wp-conj}_0$, are powerful enough to close the proof of Example 1.1, if combined with the standard principles of consequence and frame. Figure 2 shows the LHC proof of the left-hand premise of step (1) in Fig. 1. The derivation starts with an application of $\textsc{wp-nest}_0$ which allows us to apply the (unary) sequence rule $\textsc{wp-seq}_1$ only on 2. Another application of $\textsc{wp-nest}_0$ obtains the overall effect of shelving the assignment to y in 2 for later,

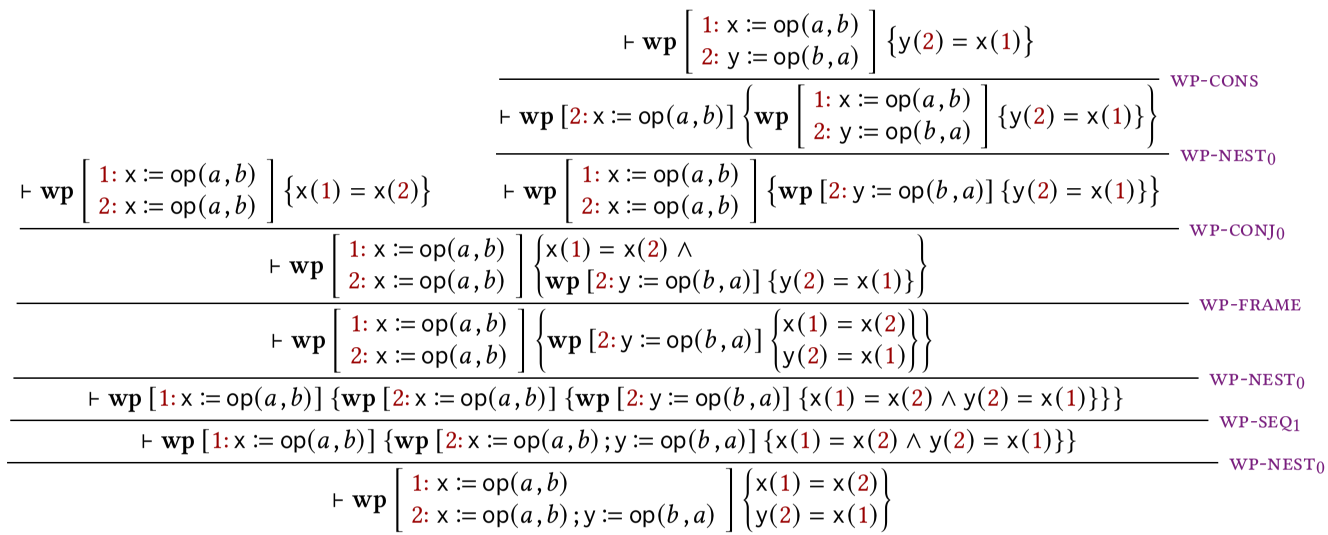

Fig. 2. LHC proof of Example 1.1: derivation of left-hand premise of step (1). The leaf on the left is discharged by the assumption of determinism of op ($\textsc{Det}_{\mathsf{op}}$). The one on the right is discharged by the assumption of commutativity of op ($\textsc{Comm}_{\mathsf{op}}$).

aligning the two assignments to x. Then, the application of the rule of frame (see WP-FRAME in Fig. 6), borrowed from Hoare logic, is justified by the assumption that op does not modify x. This means that if we prove $\mathsf{x}(1) = \mathsf{x}(2)$ in the postcondition of the outer WP, it would hold after running $[2{:}\, \mathsf{y} := \mathsf{op}(b, a)]$ too.

Since the postcondition is now a conjunction, we can apply the WP-CONJ$_0$. Note that the two hyper-terms in the premises overlap in *all* the components. The first premise coincides with our determinism assumption (DET$_{\mathsf{op}}$). For the second premise, we use WP-NEST$_0$ twice: once to push component 1 in the postcondition, and another time to fuse it with the nested 2. We obtain a top-level WP with a postcondition that coincides with our commutativity assumption (COMM$_{\mathsf{op}}$). The final application of consequence (WP-CONS) amounts to say that if the WP in the premise holds on every state, then it should hold on the state resulting from running $[2{:}\, \mathsf{x} := \mathsf{op}(a, b)]$.

The combination of WP-CONJ$_0$ and WP-NEST$_0$ allowed us to carry out the intuitive proof strategy, which relates both assignments of x and y in 2 with the single assignment to x in 1.

2.3 Projection

The rules in the previous section were induced by union of hyper-terms. We next present a rule that is induced by the operation of removing components from a hyper-term. To motivate the rule, we present another simple example that lockstep reasoning cannot handle.

Imagine we are given some deterministic terminating[6] t_1 and t_2 satisfying the specification

$$\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}\ [1{:}\, t_1\,;t_2, 2{:}\, t_2\,;t_1]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\} \qquad (\text{SWAP}_{t_1,t_2})$$

Given a vector of pairwise distinct program variables $\vec{\mathsf{x}} = \mathsf{x}_1 \ldots \mathsf{x}_n$ we write $\vec{\mathsf{x}}(i)$ for the vector $\mathsf{x}_1(i) \ldots \mathsf{x}_n(i)$; the assertion $\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)$ states pointwise equality between the two vectors. The specification above then states that, relative to some relevant variables $\vec{\mathsf{x}}$, sequencing the two commands t_1 and t_2 in one order or the other generates the same result. It should be possible to derive, within the logic, that

$$\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}\ \left[\begin{array}{l} 1{:}\, t_1\,;t_2\,;t_2 \\ 2{:}\, t_2\,;t_2\,;t_1 \end{array}\right]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}$$

It is however not possible to prove this by aligning the two programs. The only alignment that generates a match with the assumption (SWAP$_{t_1,t_2}$) would leave an unmatched t_2 at the start of 2 and one at the end of 1.

The natural proof strategy here is to introduce an auxiliary term $t_2\,;t_1\,;t_2$, and show that both components 1 and 2 above are equivalent to it. In derivation form, the desired proof looks as follows:

$$\dfrac{\dfrac{\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(3)\right\}\ \left[\begin{array}{l} 1{:}\, t_1\,;t_2\,;t_2 \\ 3{:}\, t_2\,;t_1\,;t_2 \end{array}\right]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(3)\right\} \qquad \vdash \left\{\vec{\mathsf{x}}(2) = \vec{\mathsf{x}}(3)\right\}\ \left[\begin{array}{l} 3{:}\, t_2\,;t_1\,;t_2 \\ 2{:}\, t_2\,;t_2\,;t_1 \end{array}\right]\ \left\{\vec{\mathsf{x}}(2) = \vec{\mathsf{x}}(3)\right\}}{\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(3) = \vec{\mathsf{x}}(2)\right\}\ \left[\begin{array}{l} 1{:}\, t_1\,;t_2\,;t_2 \\ 3{:}\, t_2\,;t_1\,;t_2 \\ 2{:}\, t_2\,;t_2\,;t_1 \end{array}\right]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(3) = \vec{\mathsf{x}}(2)\right\}}\ \text{WP-CONJ}_0}{\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}\ \left[\begin{array}{l} 1{:}\, t_1\,;t_2\,;t_2 \\ 2{:}\, t_2\,;t_2\,;t_1 \end{array}\right]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}}\ \text{WP-PROJ}_i$$

Read from top to bottom, the two premises express the equivalence between the original components and the auxiliary component 3. They are easily proved by the obvious lockstep proof, using (SWAP$_{t_1,t_2}$) and the assumptions of determinism (as hyper-triples).

The crucial last step removes component 3 from the hyper-triple, an operation that we call "projecting out". Note that this step is not just a by-product of the principle of consequence:

[6] We will elaborate on the assumption of termination later.

the precondition of the goal does not imply the precondition of the premise, which additionally constrains the store at 3. To justify such a step, LHC provides the following *projection* rule:

$$\frac{P \vdash \mathbf{wp}\,(\boldsymbol{t}' \cdot [i{:}\,t])\,\{Q\}}{\Pi_i.\,P \vdash \mathbf{wp}\,\boldsymbol{t}'\,\{\Pi_i.\,Q\}}\ \text{WP-PROJ}_i\ \text{proj}(t)$$

The rule introduces a novelty in the assertion language, the projection modality $\Pi_i.\,P$. A hyper-store $\boldsymbol{s}$ satisfies $\Pi_i.\,P$ if there is some store s' that can be placed at i so that the resulting hyper-store $\boldsymbol{s}[i{:}\,s']$ satisfies P. Intuitively, $\Pi_i.\,P$ removes the constraints imposed by P on the store at index i, while keeping the explicit and implied constraints imposed on the other components. For example, $\Pi_2.\,(\mathsf{x}(1) < \mathsf{x}(2) < \mathsf{x}(3))$ is logically equivalent to $\mathsf{x}(1) + 1 < \mathsf{x}(3)$: if an hyper-store satisfies $\mathsf{x}(1) + 1 < \mathsf{x}(3)$ we can replace the store at 2 with one satisfying $\mathsf{x}(2) = \mathsf{x}(1) + 1$ and obtain a hyper-store satisfying $\mathsf{x}(1) < \mathsf{x}(2) < \mathsf{x}(3)$.

Since the hyper-term in the conclusion of the rule has had its i-th component removed, both the precondition and postcondition of the conclusion are subject to the Π_i projection. The rule has an important side condition, however: $\text{proj}(t)$, which requires t to have terminating traces from any input store (in Section 4 we relax this condition). In our example this is discharged by the assumption that t_1 and t_2 are terminating. Perhaps surprisingly, the rule without side conditions is unsound: since the semantics of WP only constrains the terminating runs, if t does not terminate, Q can be False in the premise, which would result in an invalid triple in the conclusion.

The combination of projection and conjunction that we used in the example corresponds roughly to using transitivity in a refinement-based proof. It is striking that previous relational logics for hypersafety do not admit this principle. A notable exception is RHL [Benton 2004] which has a "transitivity" rule for 2-properties that could handle our example. The soundness of the rule relies however on a semantics of triples that goes beyond hypersafety and constrains the non-terminating behaviour too, in a way that is incompatible with some of LHC's rules (*e.g.*, WP-NEST_0). We discuss the trade-off involved in Section 5.5.

2.4 Reindexing and Indirect-Style Hyper-triples

A third way in which lockstep reasoning is unable to reuse hyper-triples, is the case of what we call "indirect-style" specifications. Consider the case of idempotence, that is, that t and $t\,;t$ achieve the same effect. A naive encoding of idempotence of t with respect to some relevant variables $\vec{\mathsf{x}}$, is:

$$\vdash \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\}\ [1{:}\,t, 2{:}\,(t\,;t)]\ \left\{\vec{\mathsf{x}}(1) = \vec{\mathsf{x}}(2)\right\} \qquad (\text{IDEMSEQ}_t)$$

In [Sousa and Dillig 2016], however, idempotence is specified instead as follows:

$$\vdash \left\{\vec{\mathsf{x}}(2) = \vec{v}\right\}\ [1{:}\,t, 2{:}\,t]\ \left\{\vec{\mathsf{x}}(1) = \vec{v} \Rightarrow \vec{\mathsf{x}}(2) = \vec{v}\right\} \qquad (\text{IDEM}_t)$$

The input of 1 is unconstrained; the input of 2 is assumed to be $\vec{v}$; after a run of t in each component, the postcondition asserts that, if the input of 2 happened to coincide with the output of 1, then the output of 2 is the same as the output of 1. The specification uses a slightly unintuitive pattern, which we call "indirect-style": the output of 1 is fed as input to 2 indirectly, through an implication in the postcondition.

Let us compare the two specifications. The hyper-triple (IDEMSEQ_t) is certainly easier to interpret; it also is in a form that seems readily applicable: one would expect to use idempotence precisely to relate the two components of the hyper-triple. In contrast, the hyper-triple (IDEM_t) is not directly applicable in lockstep proofs that need to relate t and $t\,;t$. It has the advantage, however, of making the two components syntactically coincide, which can be exploited by a tool to produce a simple lockstep idempotency proof, if one exists. On the other hand, attacking directly a proof of (IDEMSEQ_t)

might require one to characterize very precisely the effect of the first run of t so that the second one can be shown equivalent to a **skip**.

A difference which might not be immediate is that ($\textsc{IdemSeq}_t$) holds in fact for strictly fewer programs than ($\textsc{Idem}_t$). For an example, consider **if** $\mathtt{x} = 0$ **then** (**if** $*$ **then** $\mathtt{x} := 1$ **else** $\mathtt{x} := 2$): the result of a run from a store with $\mathtt{x} = 0$ would output a store with a random value $\mathtt{x} \in \{1, 2\}$; executing the same program again would preserve the store, therefore the program satisfies ($\textsc{Idem}_t$). It does not however satisfy ($\textsc{IdemSeq}_t$), as the programs at 1 and 2 may store different values in $\mathtt{x}$.

The two encodings of idempotence are, however, equivalent for *deterministic* programs. Given the tension between the two—one is easier to prove but harder to use and vice versa—we would like to be able to derive one from the other, *within* the logic. In CHL it is impossible to use ($\textsc{Idem}_t$) and the hyper-triple encoding determinism of t to prove ($\textsc{IdemSeq}_t$). This is, again, because the logical alignment proof strategy is fundamentally limited.

In order to support these derivations within the logic, LHC provides one final way of manipulating the hyper-term structure: *reindexing*, *i.e.*, the ability to substitute indices for indices in hyper-triples.

To illustrate the idea, let us derive ($\textsc{IdemSeq}_t$) from ($\textsc{Idem}_t$) plus determinism in LHC. There are two main obstacles to overcome: the first is that ($\textsc{IdemSeq}_t$) has three occurrences of t but ($\textsc{Idem}_t$) has only two; the second, and more relevant, is that the two runs of t in ($\textsc{IdemSeq}_t$) that correspond to the two runs in ($\textsc{Idem}_t$) are *sequenced* in the same component 2.

The first step of the proof can be dealt with using the rules we have already introduced. We want to appeal to determinism to establish that, in $[1\colon t, 2\colon (t\,;t)]$, whatever we can say about the output of the first run of t in 2, will hold for the output at 1 too, and reduce the goal to proving a triple only involving component 2. This can be achieved with the help of $\textsc{wp-conj}_0$:

$$\dfrac{\dfrac{\dfrac{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \vdash \mathbf{wp}\,[1\colon t, 2\colon t]\,\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2)\} \qquad \vdash \mathbf{wp}\,[2\colon t]\,\{\exists \vec{v}.\, \vec{\mathtt{x}}(2) = \vec{v} \wedge \mathbf{wp}\,[2\colon t]\,\{\vec{\mathtt{x}}(2) = \vec{v}\}\}}{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \vdash \mathbf{wp}\,[1\colon t, 2\colon t]\,\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \wedge \exists \vec{v}.\, (\vec{\mathtt{x}}(2) = \vec{v} \wedge \mathbf{wp}\,[2\colon t]\,\{\vec{\mathtt{x}}(2) = \vec{v}\})\}}\ \textsc{wp-conj}_0}{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \vdash \mathbf{wp}\,[1\colon t, 2\colon t]\,\{\mathbf{wp}\,[2\colon t]\,\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2)\}\}}\ (3)}{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \vdash \mathbf{wp}\,[1\colon t, 2\colon (t\,;t)]\,\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2)\}}\ (2)$$

The derivation starts just like Fig. 2: component 2 is decomposed at the sequential composition, with the help of $\textsc{wp-nest}_0$ and $\textsc{wp-seq}_1$. Then, again similarly to Fig. 2, consequence and frame are used to decouple the assertions on 1 and 2 in the postcondition: we can prove (by determinism) that $\mathtt{x}(1) = \mathtt{x}(2)$ holds already after the first runs of t in 1 and 2; and separately we can prove the second run in 2 will preserve whatever value is stored in $\mathtt{x}(2)$. The $\textsc{wp-conj}_0$ rule allows us to discharge the first conjunct of the postcondition by appealing to the assumption of determinism of t.

We are now left with our goal being the right-hand premise, which has only two occurrences of t, although they are both at 2. This identifies the crux of the problem: our new goal involves two *sequenced* runs of the same term t, but the indirect-style hyper-triple we take as our assumption (*i.e.*, ($\textsc{Idem}_t$)) relates two runs in separate indices of a hyper-triple.

To close the proof we need two new rules of LHC that deal precisely with re-assigning indices to components in a WP. LHC uses the *reindexing* notation $P(\!|i/j|\!)$ to denote the assertion that is true on a hyper-store $\boldsymbol{s}$ if P is true on the hyper-store $\boldsymbol{s}[j\colon \boldsymbol{s}(i)]$. For a more syntactic intuition, $P(\!|i/j|\!)$ is the assertion P where the occurrences of the index j are replaced with i. LHC proposes two main rules (simplified here slightly for clarity) explaining the interaction between reindexing and WP:

$$\dfrac{\vdash \mathbf{wp}\,\boldsymbol{t}\,\{Q\}}{\vdash \mathbf{wp}\,\boldsymbol{t}\,\{Q(\!|i/j|\!)\}}\ j \notin \mathrm{supp}(\boldsymbol{t})\ \textsc{wp-idx-post}_0 \qquad \dfrac{}{\left(\mathbf{wp}\,[j\colon t]\,\{Q\}\right)(\!|i/j|\!) \vdash \mathbf{wp}\,[i\colon t]\,\{Q(\!|i/j|\!)\}}\ i \notin \mathrm{idx}(Q)\ \textsc{wp-idx-swap}_0$$

Rule WP-IDX-POST$_0$ states that it is possible to substitute j for i in the postcondition of a WP if $\boldsymbol{t}$ does not have a component at index j. Rule WP-IDX-SWAP$_0$ shows the effect of applying the substitution to a WP that does contain a component at j. (We explain the need for the side conditions on these rules in Section 4.)

With these two rules we can now complete the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vec{x}(3)=\vec{v}\vdash \mathbf{wp}\,[2{:}\,t,3{:}\,t]\,\{\vec{x}(2)=\vec{v}\Rightarrow\vec{x}(3)=\vec{v}\}}{\vdash \mathbf{wp}\,[2{:}\,t]\,\{\forall\vec{v}.\,(\vec{x}(2)=\vec{v}\wedge\vec{x}(3)=\vec{v})\Rightarrow \mathbf{wp}\,[3{:}\,t]\,\{\vec{x}(3)=\vec{v}\}\}}\,(7)}{\vdash \mathbf{wp}\,[2{:}\,t]\,\{(\forall\vec{v}.\,(\vec{x}(2)=\vec{v}\wedge\vec{x}(3)=\vec{v})\Rightarrow \mathbf{wp}\,[3{:}\,t]\,\{\vec{x}(3)=\vec{v}\})(\!|2/3|\!)\}}\,\text{WP-IDX-POST}_0}\,(6)}{\vdash \mathbf{wp}\,[2{:}\,t]\,\{\forall\vec{v}.\,\vec{x}(2)=\vec{v}\Rightarrow(\mathbf{wp}\,[3{:}\,t]\,\{\vec{x}(3)=\vec{v}\})(\!|2/3|\!)\}}\,(5)}{\vdash \mathbf{wp}\,[2{:}\,t]\,\{\exists\vec{v}.\,\vec{x}(2)=\vec{v}\wedge(\mathbf{wp}\,[3{:}\,t]\,\{\vec{x}(3)=\vec{v}\})(\!|2/3|\!)\}}\,\text{WP-IDX-SWAP}_0}{\vdash \mathbf{wp}\,[2{:}\,t]\,\{\exists\vec{v}.\,\vec{x}(2)=\vec{v}\wedge \mathbf{wp}\,[2{:}\,t]\,\{(\vec{x}(3)=\vec{v})(\!|2/3|\!)\}\}}\,(4)$$

$$\vdash \mathbf{wp}\,[2{:}\,t]\,\{\exists\vec{v}.\,\vec{x}(2)=\vec{v}\wedge \mathbf{wp}\,[2{:}\,t]\,\{\vec{x}(2)=\vec{v}\}\}$$

We start at the top with (IDEM$_t$); we renamed 1 to 2 and 2 to 3, which is possible as LHC's specifications are closed under unambiguous renaming of indices.[7] Step (7) uses WP-NEST$_0$ and some simple logical manipulations (mainly commutation laws between implication and WP) to put the postcondition in a suitable form. An application of WP-IDX-POST$_0$ applies the $(\!|2/3|\!)$ substitution to the postcondition, which is propagated using consequence in step (6), until it reaches the inner WP. Step (5) uses consequence and the tautology $\exists\vec{v}.\,\vec{x}(2)=\vec{v}$ to apply the universal quantification in the postcondition. We then use WP-IDX-SWAP$_0$ to propagate the reindexing to the postcondition of the inner WP. Step (4) finally applies the reindexing to the postcondition of the inner WP obtaining the desired judgment.

2.5 Loop Invariants

The treatment of loops is a thorny issue in relational logics. Loops are where all the shortcomings of the alignment proof strategy get amplified. Consider the following simple consequence of idempotence and determinism of t (picking `i` to be a fresh variable):

$$\vdash \{\vec{x}(1)=\vec{x}(2)\}\ [1{:}\,t,2{:}\,(t;\texttt{while i>0 do }(t;\texttt{i--}))]\ \{\vec{x}(1)=\vec{x}(2)\} \qquad (\text{IDEMLOOP}_t)$$

The idea of lockstep proofs underlies all the relational rules for loops in the literature: they work well when all the components in a hyper-triple are loops which can be aligned at the boundaries of their bodies. Since the relational specifications of idempotence and determinism of t do not give us direct information about single runs of t (only of *pairs* of runs) this strategy immediately falls apart when attempting a proof of (IDEMLOOP$_t$). We only have one component with a loop, containing runs of t which we fail to align to any other runs of t in the other component.

Let us inspect the issue more closely, by sketching a derivation:

$$\dfrac{\vdash \{\vec{x}(1)=\vec{x}(2)\}\ [1{:}\,t,2{:}\,t]\ \{P\} \qquad \dfrac{\vdash \{P\}\ [2{:}\,(t;\texttt{i--})]\ \{P\}}{\vdash \{P\}\ [2{:}\,(\texttt{while i>0 do }(t;\texttt{i--}))]\ \{P\}}}{\vdash \{\vec{x}(1)=\vec{x}(2)\}\ [1{:}\,t,2{:}\,(t;\texttt{while i>0 do }(t;\texttt{i--}))]\ \{\vec{x}(1)=\vec{x}(2)\}}$$

We start by considering the first runs of t in 1 and 2 in lockstep, and separately the loop, from the resulting state. As in a standard proof, proving the loop boils down to finding a suitable loop invariant. In all the previous relational logics the most precise information we can obtain for the

[7] Formally this is handled by rules IDX and WP-IDX, presented in Section 4.

left-hand premise, reusing the specifications of t, is $P = \left(\vec{x}(1) = \vec{x}(2)\right)$. Unfortunately, the right-hand premise cannot be proven with this P, since t does not in general preserve it: the assertion is too weak in that it does not record the fact that a run of t already happened before the loop starts. Short of this information, the verification of the loop needs to consider the case where the body executes t for the first time ever, which in general would not preserve the equivalence (as there is no corresponding run of t in 1).

This is where the extended expressivity of LHC is crucial to obtain a modular and relational proof. In LHC we can set $P = \exists \vec{v}.\ \left(\vec{x}(1) = \vec{x}(2) = \vec{v} \land \mathbf{wp}\ [2\colon t]\ \{\vec{x}(2) = \vec{v}\}\right)$ which correctly records the fact that t has run already, by asserting that a further run of t at 2 would not modify $\vec{x}$. The left-hand premise can be established with this P by reusing the derivation of the previous section—it is a direct consequence of the premise of step (3). The verification of the body of the loop can now use the WP in P to justify why t has no effect on $\vec{x}$. The full derivation, shown for a variant of this example in [D'Osualdo et al. 2022], crucially relies on LHC's ability to handle nested WPs.

Through this series of examples we introduced the main novel reasoning principles of LHC, and showed how they enable compositional, modular proofs, embracing hyper-triples as the fundamental building block of hypersafety proofs.

3 PRELIMINARIES

In this section we fix notation, and define assertions over hyper-stores and their basic laws.

Definition 3.1 (Finite maps). Given a type A, we define the type $A_\bot \triangleq A \uplus \{\bot\}$. Given a function $f\colon A \to B_\bot$ we define $\mathrm{supp}(f) \triangleq \{a : A \mid f(a) \neq \bot\}$. We say f is a *finite map from A to B*, written $f\colon A \rightharpoonup B$, if $f\colon A \to B_\bot$ and $\mathrm{supp}(f)$ is finite.

We write $[a_1\colon b_1, \ldots, a_n\colon b_n]$ for the finite map associating each a_i to b_i (and everything else to $\bot$). Similarly to set comprehensions, we use the notation $[a\colon b \mid \varphi(a,b)]$, e.g. $[i\colon j \mid i \in \mathbb{N}, i = 2j \leq 4] = [0\colon 0, 2\colon 1, 4\colon 2]$. Given $f, g\colon A \rightharpoonup B$ such that $\forall x \in \mathrm{supp}(f) \cap \mathrm{supp}(g).\ f(x) = g(x)$, the union of f and g, written $f + g\colon A \rightharpoonup B$ is defined as:

$$(f+g)(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{supp}(f) \setminus \mathrm{supp}(g) \\ g(x) & \text{if } x \in \mathrm{supp}(g) \\ \bot & \text{otherwise} \end{cases}$$

We leave $f + g$ undefined if $f(x) \neq g(x)$ for some $x \in \mathrm{supp}(f) \cap \mathrm{supp}(g)$. The *disjoint union* of f and g, written $f \cdot g\colon A \rightharpoonup B$, is defined as $f \cdot g \triangleq f + g$ if $\mathrm{supp}(f) \cap \mathrm{supp}(g) = \emptyset$, undefined otherwise. For any function $f\colon A \to B$, we define $f[a\colon b] \triangleq \lambda x.\ \mathbf{if}\ x = a\ \mathbf{then}\ b\ \mathbf{else}\ f(x)$.

Both operations $(+)$ and $(\cdot)$ on maps are commutative and associative (where defined) and have the empty map $[]$ as neutral element. Indices are natural numbers $i \in \mathbb{I} \triangleq \mathbb{N}$. We make extensive use of finite maps from $\mathbb{I}$: if A is the type of *things* then $\mathbb{I} \to A_\bot$ is the type of *hyper-things*. As a notational convention, if a meta-variable a ranges over A we use $\boldsymbol{a}$ to range over $\mathbb{I} \to A_\bot$. For a set of indices $I \subseteq \mathbb{I}$ and some $x \in A$, we write x^I for $[i\colon x \mid i \in I]$. Given $I \subseteq \mathbb{I}$, we lift a relation $\sim$ on A to the relation $\sim_I$ on $\mathbb{I} \rightharpoonup A$ as follows; for $\boldsymbol{a}_1, \boldsymbol{a}_2\colon \mathbb{I} \rightharpoonup A$, $\boldsymbol{a}_1 \sim_I \boldsymbol{a}_2$ holds when $I \subseteq \mathrm{supp}(\boldsymbol{a}_1), \mathrm{supp}(\boldsymbol{a}_2)$ and $\forall i \in \mathbb{I}.\ \boldsymbol{a}_1(i) \sim \boldsymbol{a}_2(i)$. When $a_0 \in A$, we write $\boldsymbol{a} \sim_I a_0$ to mean $\forall i \in \mathbb{I}.\ \boldsymbol{a}(i) \sim a_0$.

Definition 3.2 (Reindexing). A function $\pi\colon \mathbb{I} \to \mathbb{I}$ is called a *reindexing*. Given $\boldsymbol{a}\colon \mathbb{I} \to A$, we write $\boldsymbol{a}(\!|\pi|\!)$ to apply the reindexing π to the map: $\boldsymbol{a}(\!|\pi|\!) \triangleq [i\colon \boldsymbol{a}(\pi(i)) \mid i \in \mathbb{I}]$. We write $\boldsymbol{a}(\!|j_1/i_1,\ \ldots,\ j_n/i_n|\!)$ to denote $\boldsymbol{a}(\!|\pi|\!)$ where $\pi(i_k) = j_k$ and $\pi(i) = i$ if $i \in \mathbb{I} \setminus \{i_1, \ldots, i_n\}$.

3.1 Hyper-programs

Syntax. We use a minimal untyped imperative language to formalize our ideas. We will assume an enumerable set of *program variables* $\mathsf{x} \in \mathbb{X}$. The set of *values* $v \in \mathbb{V} \triangleq \mathbb{Z}$ is the set of integers, for simplicity. Booleans are represented using 0 for false, and any other integer for true.

$$\begin{aligned}\mathbb{T} \ni t, g, e ::= {} & v \mid \mathsf{x} \mid * \mid e + e \mid e - e \mid e \leq e \mid \dots \\ & \mid \textbf{skip} \mid \mathsf{x} := e \mid t ; t \mid \textbf{if } g \textbf{ then } t \textbf{ else } t \mid \textbf{while } g \textbf{ do } t\end{aligned}$$

Every term evaluates to some value and mutates a first-order store. We use the meta-variables g for terms that are meant to evaluate to a boolean (i.e. guards), and e for terms that are meant to evaluate to an integer (i.e. expressions). The $*$ expression chooses some integer non-deterministically, and returns it. Commands like **skip** and **while** have an irrelevant (and thus arbitrary) return value. A simplifying (but inessential) assumption is that evaluation never faults. If needed, one can model expressions like $n/0$ as returning a special NaN value.

To keep the development as simple as possible, we also don't model scoping and function calls. Including them can be handled using standard techniques. Function calls in our examples should be understood as simply naming code blocks.

The functions $\mathrm{pvar}(t)$ and $\mathrm{mods}(t)$ return the set of program variables occurring in t and modified by t respectively. Their definition is standard. We extend them to hyper-terms by setting $\mathrm{pvar}(\boldsymbol{t}) = \{(\mathsf{x}, i) \mid i \in \mathrm{supp}(\boldsymbol{t}), \mathsf{x} \in \mathrm{pvar}(\boldsymbol{t}(i))\}$ (and similarly for mods).

Semantics. A *store* is an element of $\mathbb{S} \triangleq \mathbb{X} \to \mathbb{V}$. For simplicity, we adopt a big-step semantics for our language. The judgment $\langle t, s\rangle \Downarrow \langle v, s'\rangle$ indicates that the term t starting from input store s may terminate with the return value $v \in \mathbb{V}$ and output store s'. The definition of $\langle t, s\rangle \Downarrow \langle v, s'\rangle$ is in [D'Osualdo et al. 2022]. We define $\langle t, s\rangle\!\Downarrow \triangleq \exists v, s'.\, \langle t, s\rangle \Downarrow \langle v, s'\rangle$. Note that $\langle t, s\rangle\!\Downarrow$ is equivalent to termination only if t does not have non-deterministic steps. For example, $\langle \textbf{while } * \textbf{ do skip}, s\rangle\!\Downarrow$ holds even though, in a standard small-step semantics, the program has a diverging execution.

Definition 3.3. A *hyper-term* is a finite partial function $\boldsymbol{t} : \mathbb{I} \rightharpoonup \mathbb{T}$. A *hyper-store* is a function $\boldsymbol{s} : \mathbb{I} \to \mathbb{S}$. A *hyper-return-value* is a finite partial function $\boldsymbol{r} : \mathbb{I} \rightharpoonup \mathbb{V}$. The big-step semantics judgment is lifted to hyper-terms as follows:

$$\langle \boldsymbol{t}, \boldsymbol{s}\rangle \Downarrow \langle \boldsymbol{r}, \boldsymbol{s}'\rangle \triangleq \forall i \in \mathbb{I}. \begin{cases} \langle \boldsymbol{t}(i), \boldsymbol{s}(i)\rangle \Downarrow \langle \boldsymbol{r}(i), \boldsymbol{s}'(i)\rangle & \text{if } i \in \mathrm{supp}(\boldsymbol{t}) \\ \boldsymbol{s}(i) = \boldsymbol{s}'(i) \wedge \boldsymbol{r}(i) = \bot & \text{otherwise} \end{cases}$$

Note that on all indices where $\boldsymbol{t}$ is undefined, the store is preserved untouched.

3.2 Hyper-assertions

Definition 3.4 (Hyper-assertion). *Assertions* are of type $A \in \mathrm{Assrt} \triangleq \mathbb{S} \to \mathrm{Prop}$. A *hyper-assertion* is a function of type $P \in \mathrm{HAssrt} \triangleq (\mathbb{I} \to \mathbb{S}) \to \mathrm{Prop}$. A *post hyper-assertion* has the type $Q : (\mathbb{I} \rightharpoonup \mathbb{V}) \to \mathrm{HAssrt}$. They are used in postconditions, where the hyper-value argument is bound to the return value of the hyper-term in the triple. We require post hyper-assertions to be *upward-closed* on the hyper-return-value: if $Q(\boldsymbol{v})(\boldsymbol{s})$ and $\forall i \in \mathrm{supp}(\boldsymbol{v}).\, \boldsymbol{v}(i) = \boldsymbol{v}'(i)$ then $Q(\boldsymbol{v}')(\boldsymbol{s})$.

The set of indices that are relevant for P is the set $\mathrm{idx}(P)$ defined below. For example, we have $\mathrm{idx}\big((\mathsf{x}(1) = \mathsf{y}(2)) \vee \mathsf{z}(3) = 0\big) = \{1, 2, 3\}$.

*Definition 3.5 (*idx*).* The *indices of* a (post) hyper-assertion P (resp. Q) are the set:

$$\begin{aligned}\mathrm{idx}(P) &\triangleq \mathbb{I} \setminus \{i \in \mathbb{I} \mid \forall \boldsymbol{s}, s'.\, P(\boldsymbol{s}) \Leftrightarrow P(\boldsymbol{s}[i : s'])\} \\ \mathrm{idx}(Q) &\triangleq \mathbb{I} \setminus \{i \in \mathbb{I} \mid \forall \boldsymbol{r}, \boldsymbol{s}, s'.\, Q(\boldsymbol{r})(\boldsymbol{s}) \Leftrightarrow Q(\boldsymbol{r}[i : \bot])(\boldsymbol{s}[i : s'])\}\end{aligned}$$

$$
\begin{array}{rlrl}
(\mathsf{x}(i) = v) & \triangleq \lambda \boldsymbol{s}.\, \boldsymbol{s}(i)(\mathsf{x}) = v & P_1 \Rightarrow P_2 & \triangleq \lambda \boldsymbol{s}.\, P_1(\boldsymbol{s}) \Rightarrow P_2(\boldsymbol{s}) \\
P_1 \wedge P_2 & \triangleq \lambda \boldsymbol{s}.\, P_1(\boldsymbol{s}) \wedge P_2(\boldsymbol{s}) & P_1 \vee P_2 & \triangleq \lambda \boldsymbol{s}.\, P_1(\boldsymbol{s}) \vee P_2(\boldsymbol{s}) \\
\exists x.\, P(x) & \triangleq \lambda \boldsymbol{s}.\, \exists x.\, P(x)(\boldsymbol{s}) & \forall x.\, P(x) & \triangleq \lambda \boldsymbol{s}.\, \forall x.\, P(x)(\boldsymbol{s}) \\
P(\!|\pi|\!) & \triangleq \lambda \boldsymbol{s}.\, P(\boldsymbol{s}(\!|\pi|\!)) & Q(\!|\pi|\!) & \triangleq \lambda \boldsymbol{r}.\, Q(\boldsymbol{r}(\!|\pi|\!))(\!|\pi|\!) \\
\Pi_I.\, P & \triangleq \lambda \boldsymbol{s}.\, \exists \boldsymbol{s}'.\, P(\boldsymbol{s}[i{:}\,\boldsymbol{s}'(i) \mid i \in I]) & \hat{\Pi}_I.\, Q & \triangleq \lambda \boldsymbol{r}.\, \exists \boldsymbol{v}.\, \Pi_I.\, Q(\boldsymbol{r}[i{:}\,\boldsymbol{v}(i) \mid i \in I]) \\
A@I & \triangleq \lambda \boldsymbol{s}.\ \textstyle\bigwedge_{i \in I} A(\boldsymbol{s}(i)) & Q_1 \wedge Q_2 & \triangleq \lambda \boldsymbol{r}.\, Q_1(\boldsymbol{r}) \wedge Q_2(\boldsymbol{r})
\end{array}
$$

Fig. 3. Hyper-assertions

Similarly, we define $\mathrm{pvar}(P)$ as the set of (indexed) program variables of an hyper-assertion such that $(\mathsf{x}, i) \in \mathrm{pvar}(P)$ if arbitrarily changing the value of x at i may affect whether P holds.

*Definition 3.6 (*pvar*).* The *program variables of* a (post) hyper-assertion P (resp. Q) are the set:

$$
\begin{aligned}
\mathrm{pvar}(P) &\triangleq (\mathbb{X} \times \mathbb{I}) \setminus \left\{ (\mathsf{x}, i) \;\middle|\; \forall \boldsymbol{s}, v.\, P(\boldsymbol{s}) \Leftrightarrow P\big(\boldsymbol{s}[i{:}\,\boldsymbol{s}(i)[\mathsf{x}{:}\,v]]\big) \right\} \\
\mathrm{pvar}(Q) &\triangleq \textstyle\bigcup_{\boldsymbol{r}:\, \mathbb{I} \rightharpoonup \mathbb{V}} \mathrm{pvar}(Q(\boldsymbol{r}))
\end{aligned}
$$

Even though we defined idx and pvar semantically, any over-approximation would suffice to preserve the soundness of the side conditions of our rules.

We will be using a number of hyper-assertions, summarized in Fig. 3. Pure meta-level propositions φ lift to *pure* hyper-assertions $\lambda\_.\, \varphi$, which hold independently of the state. An assertion $A : \mathrm{Assrt}$, which predicates over single stores, can be lifted to a hyper-assertion, which predicates over hyper-stores, by $A@I$ which specifies on which indices I, A is required to hold. In addition to the usual logical connectives, we introduce some notation to deal with hyper-stores. To refer to the value of a program variable x at index i in a hyper-store, we write $\mathsf{x}(i)$.

The hyper-assertion $P(\!|\pi|\!)$ uses the reindexing π to change the indices of the hyper-store before checking P holds on it. The notation is extended to post hyper-assertions by reindexing the hyper-return-value too. Note that while reindexing an assertion is intuitively akin to applying a substitution to indices, we are giving it here a *semantic* definition. On simple assertions, reindexing does propagate just like a substitution of indices. For example $\big(5 \le \mathsf{x}(1) \wedge \mathsf{x}(2) \le \mathsf{x}(3)\big)(\!|2/1|\!)$ is logically equivalent to $\big(5 \le \mathsf{x}(2) \le \mathsf{x}(3)\big)$. We will however see in Section 4.4 that its interaction with WPs is not trivial.

The *projection modality* Π_I projects out the information on the components in I of the hyper-store under consideration. $\Pi_I.\, P$ holds on a hyper-store $\boldsymbol{s}$ if P holds on some hyper-store that coincides with $\boldsymbol{s}$ on every index except for the ones in I. Notice that $\mathrm{idx}(\Pi_I.\, P) \cap I = \emptyset$. For brevity, when $I = \{i\}$ is a singleton, we omit the braces and write $\Pi_i.\, P$. Projection of a post hyper-assertion $\hat{\Pi}_I.\, Q$, projects out the components at I of the hyper-return-value too.

Definition 3.7. An assertion A is *valid*, written $\vdash A$, just if $\forall s.\, A(s)$. Similarly, we define *validity*, *entailment*, and *logical equivalence* of hyper-assertions:

$$
\vdash P \triangleq \forall \boldsymbol{s}.\, P(\boldsymbol{s}) \qquad \Gamma \vdash P \triangleq (\vdash \textstyle\bigwedge \Gamma \Rightarrow P) \qquad P \dashv\vdash P' \triangleq (P \vdash P') \wedge (P' \vdash P)
$$

where Γ is a list of hyper-assertions. When Γ is empty, $\bigwedge \Gamma = \mathsf{True}$. Since Γ is always interpreted as the conjunction of its items we will not make a distinction between Γ and $\bigwedge \Gamma$.

In Fig. 4 we show a selection of laws on entailments of basic hyper-assertions. These laws are standard and mirror the laws for standard connectives. In Fig. 5 we show the core laws that apply to the two constructs that are special to hyper-assertions: reindexing and projection.

$$\Gamma, P \vdash P \qquad \frac{\Gamma \vdash P' \qquad \Gamma, P' \vdash P}{\Gamma \vdash P} \qquad \frac{\Gamma, P \vdash R}{\Gamma \vdash P \Rightarrow R} \qquad \frac{\forall x.\, (\Gamma, P(x) \vdash R)}{\Gamma, \exists x.\, P(x) \vdash R} \qquad \frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$$

Fig. 4. Basic hyper-assertion laws (selection).

IDX
$$\frac{\Gamma \vdash P}{\Gamma(\!|\pi|\!) \vdash P(\!|\pi|\!)}$$

IDX-PVAR
$$(\mathsf{x}(i) = v)(\!|\pi|\!) \dashv\vdash \mathsf{x}(\pi(i)) = v$$

IDX-EX
$$(\exists x.\, P(x))(\!|\pi|\!) \dashv\vdash \exists x.\, (P(x)(\!|\pi|\!))$$

IDX-CONJ
$$(P_1 \wedge P_2)(\!|\pi|\!) \dashv\vdash (P_1(\!|\pi|\!) \wedge P_2(\!|\pi|\!))$$

IDX-IMPL
$$(P_1 \Rightarrow P_2)(\!|\pi|\!) \dashv\vdash (P_1(\!|\pi|\!) \Rightarrow P_2(\!|\pi|\!))$$

IDX-IRREL
$$\frac{\forall i \in \mathrm{idx}(P).\, \pi(i) = i}{P(\!|\pi|\!) \dashv\vdash P}$$

PROJ
$$\frac{\Gamma \vdash P}{\Pi_I.\, \Gamma \vdash \Pi_I.\, P}$$

PROJ-INTRO
$$P \vdash \Pi_I.\, P$$

PROJ-MERGE
$$\Pi_{I_1}.\, \Pi_{I_2}.\, P \dashv\vdash \Pi_{I_1 \cup I_2}.\, P$$

PROJ-IRREL
$$\frac{\mathrm{idx}(P) \cap I = \emptyset}{\Pi_I.\, P \vdash P}$$

PROJ-STORE
$$\frac{i \notin \mathrm{idx}(P) \qquad |\vec{v}| = |\vec{\mathsf{x}}|}{\exists \vec{v}.\, P(\vec{v}) \vdash \Pi_i.\, \exists \vec{v}.\, (P(\vec{v}) \wedge \vec{\mathsf{x}}(i) = \vec{v})}$$

Fig. 5. Hyper-assertion laws for reindexing and projection.

Rule IDX allows the application of arbitrary reindexing on both sides of the turnstile. Rules IDX-PVAR to IDX-IMPL show how reindexing distributes over the other connectives. To eliminate a reindexing one can use these rules to push the reindexing down the structure of the assertion until either IDX-PVAR applies or the reindexing does not affect the indices of the assertion and IDX-IRREL allows to remove it.

Rule PROJ allows projection to be introduced on both sides of the turnstile. Notice that the rule is not an instance of consequence: the assumption $\Pi_I.\, \Gamma$ in the conclusion does not imply the assumption Γ in the premise.

Rule PROJ-INTRO is the most obvious way to introduce a projection, but it is not the most useful; typically when introducing an assertion $\Pi_i.\, P$ we want to assert in P facts that are not true for the current store at index i, but would be true if we reassigned the store at i appropriately. Rule PROJ-STORE supports this common scenario: for instance, we can use it to prove $(\mathsf{x}(1) = \mathsf{x}(2)) \vdash \exists v.\, \mathsf{x}(1) = \mathsf{x}(2) = v \vdash \Pi_3.\, \big(\exists v.\, \mathsf{x}(1) = \mathsf{x}(2) = v \wedge \mathsf{x}(3) = v\big) \vdash \Pi_3.\, (\mathsf{x}(1) = \mathsf{x}(2) = \mathsf{x}(3))$.

Rule PROJ-IRREL is the main mean to eliminate projection: starting from an assertion P_0 with $\mathrm{idx}(P_0) \cap I \neq \emptyset$ one first proves $P_0 \vdash P$ for some suitably strong P with $\mathrm{idx}(P_0) \cap I = \emptyset$; then an application of PROJ and PROJ-IRREL give us $\Pi_I.\, P_0 \vdash P$.

4 THE PROGRAM LOGIC

To make our logic capable of asserting properties of programs, we introduce a *weakest precondition* (WP) modality $\mathbf{wp}\, \boldsymbol{t}\, \{Q\}$, where $\boldsymbol{t} : \mathbb{I} \rightharpoonup \mathbb{T}$ is a hyper-term, and $Q : (\mathbb{I} \rightharpoonup \mathbb{V}) \to \mathrm{HAssrt}$ is the postcondition.[8] It intuitively holds on the hyper-states from which $\boldsymbol{t}$ runs yielding an hyper-return-value $\boldsymbol{v}$ and an hyper-state $\boldsymbol{s}'$ satisfying the postcondition Q. The *arity* of $\mathbf{wp}\, \boldsymbol{t}\, \{Q\}$ is $|\mathrm{supp}(\boldsymbol{t})|$.

[8] We omit the binder in the postcondition when the return values are simply ignored, i.e. $\mathbf{wp}\, \boldsymbol{t}\, \{P\} \triangleq \mathbf{wp}\, \boldsymbol{t}\, \{\lambda\_.\, P\}$.

$$\textsc{wp-triv}\quad \vdash \mathbf{wp}\, t\, \{\mathrm{True}\} \qquad \textsc{wp-cons}\quad \frac{\forall v.\ Q(v) \vdash Q'(v)}{\mathbf{wp}\, t\, \{Q\} \vdash \mathbf{wp}\, t\, \{Q'\}} \qquad \textsc{wp-all}\quad \forall x.\, \mathbf{wp}\, t\, \{Q(x)\} \dashv\vdash \mathbf{wp}\, t\, \{\forall x.\, Q(x)\}$$

$$\textsc{wp-frame}\quad \frac{\mathrm{pvar}(P) \cap \mathrm{mods}(t) = \emptyset}{P \wedge \mathbf{wp}\, t\, \{Q\} \vdash \mathbf{wp}\, t\, \{P \wedge Q\}} \qquad \textsc{wp-impl-r}\quad \frac{\mathrm{pvar}(P) \cap \mathrm{mods}(t) = \emptyset}{P \Rightarrow \mathbf{wp}\, t\, \{Q\} \dashv\vdash \mathbf{wp}\, t\, \{\lambda r.\, P \Rightarrow Q(r)\}}$$

$$\textsc{wp-subst}\quad \frac{\mathrm{x} \notin \mathrm{mods}(t)}{\mathrm{x}(i) = v \wedge \mathbf{wp}\left(\left[i{:}\, t[v/\mathrm{x}]\right] \cdot t'\right)\{Q\} \vdash \mathbf{wp}\left([i{:}\, t] \cdot t'\right)\{Q\}} \qquad \textsc{wp-idx}\quad \frac{\pi \text{ bijective}}{(\mathbf{wp}\, t\, \{Q\})(\!|\pi|\!) \vdash \mathbf{wp}\, t(\!|\pi|\!)\, \{Q(\!|\pi|\!)\}}$$

Fig. 6. Weakest precondition laws: structural rules.

Definition 4.1 (Weakest precondition). $\mathbf{wp}\, t\, \{Q\} \triangleq \lambda s.\, \big(\forall s', v.\, \langle t, s\rangle \Downarrow \langle v, s'\rangle \Rightarrow Q(v)(s')\big)$.

Hyper-triples are defined in terms of weakest-preconditions: $\{P\}\ t\ \{Q\} \triangleq P \Rightarrow \mathbf{wp}\, t\, \{Q\}$. This definition is consistent with the one adopted by [Barthe et al. 2011; Sousa and Dillig 2016]. Other relational program logics, notably [Benton 2004; Yang 2007], insist that the hyper-terms either all diverge or all terminate. We elaborate on the trade-offs implied by this choice in Section 5.5.

One possible variation is to require *safety* of the terms, *i.e.*, that the terms do not fault (as in e.g. [Yang 2007]) which is a common choice in Hoare/Separation logic. For simplicity we do not model faults in our language. The most flexible extension would model faults as a special return value $\lightning$. This way, the postcondition can decide what should happen when each component faults. For some applications, requiring safety ($\lambda r.\, \forall i.\, r(i) \neq \lightning$) may be appropriate, for others it may be sufficient to show that if one component faults then the others do too ($\lambda r.\, \forall i, j.\, r(i) = \lightning \Leftrightarrow r(j) = \lightning$).

Another important point of departure from the literature is that we consider a non-deterministic programming language (in contrast with e.g. [Barthe et al. 2011; Benton 2004; Yang 2007]). An hyper-triple of shape $\vdash \{\vec{\mathrm{x}}(1) = \vec{\mathrm{x}}(2)\}\ [1{:}\, t_1, 2{:}\, t_2]\ \{\vec{\mathrm{x}}(1) = \vec{\mathrm{x}}(2)\}$, for example, encodes semantic equivalence of t_1 and t_2 in a deterministic language. But in general it represents a stronger property, which implies determinism of t_1 and t_2 (assuming they terminate). The flexibility of our logic allows us to consider non-deterministic programs, and obtain the stronger proofs that the hypothesis of determinism enables by simply involving in the derivations the *hyper-triple encoding* of determinism.

Overview of the proof rules. We take a semantic approach in formalizing WP, therefore proof rules are just lemmas involving WP. We divide LHC's rules in four groups. The *structural* rules (in Fig. 6) apply regardless of the hyper-terms in the WP, and are mostly adaptations of standard Hoare Logic rules. Then we have the *lockstep* rules (in Fig. 7), which match on the structure of the program terms of all components at the same time. These are minor adaptations of rules that are virtually present (most often in the special case of arity 2) in all the relational program logics in the literature. The *hyper-structure* rules (in Fig. 8) provide the basic reasoning principles needed to compose hyper-triples of varying arity. Finally, the *reindexing* rules (in Fig. 9) allow the sound merging of components, which underpins the manipulations needed to handle indirect-style triples.

All rules apply only if all the components are defined. This implies some implicit side conditions. For example, whenever a rule involves an expression $t_1 \cdot t_2$, it only applies if $\mathrm{supp}(t_1) \cap \mathrm{supp}(t_2) = \emptyset$. Similarly, an expression $t_1 + t_2$ implies the constraint that $\forall i \in \mathrm{supp}(t_1) \cap \mathrm{supp}(t_2).\, t_1(i) = t_2(i)$.

$$\textsc{wp-seq}_I$$
$$\mathbf{wp}\,[i\colon t_i \mid i \in I]\,\{\mathbf{wp}\,[i\colon t'_i \mid i \in I]\,\{Q\}\} \dashv\vdash \mathbf{wp}\,[i\colon (t_i\,;\,t'_i) \mid i \in I]\,\{Q\}$$

$$\textsc{wp-assign}_I$$
$$\frac{\forall i \in I.\,(\mathsf{x}_i, i) \notin \mathrm{pvar}(Q)}{\mathbf{wp}\,[i\colon e_i \mid i \in I]\,\{Q\} \vdash \mathbf{wp}\,[i\colon \mathsf{x}_i := e_i \mid i \in I]\,\{\lambda \boldsymbol{r}.\,Q(\boldsymbol{r}) \wedge \textstyle\bigwedge_{i\in I} \boldsymbol{r}(i) = \mathsf{x}_i(i)\}}$$

$$\textsc{wp-if}_I$$
$$\mathbf{wp}\,[i\colon g_i \mid i \in I]\left\{\lambda \boldsymbol{b}.\,\mathbf{wp}\left(\begin{matrix}[i\colon t_i \mid i \in I, \boldsymbol{b}(i) \neq 0]\cdot \\ [i\colon t'_i \mid i \in I, \boldsymbol{b}(i) = 0]\end{matrix}\right)\{Q\}\right\} \dashv\vdash \mathbf{wp}\,[i\colon \mathtt{if}\ g_i\ \mathtt{then}\ t_i\ \mathtt{else}\ t'_i \mid i \in I]\,\{Q\}$$

$$\textsc{wp-while}_I$$
$$\frac{P \vdash \mathbf{wp}\,[i\colon g_i \mid i \in I]\,\{\lambda \boldsymbol{b}.\,(\boldsymbol{b} =_I 0 \wedge R) \vee (\boldsymbol{b} \neq_I 0 \wedge \mathbf{wp}\,[i\colon t_i \mid i \in I]\,\{P\})\}}{P \vdash \mathbf{wp}\,[i\colon \mathtt{while}\ g_i\ \mathtt{do}\ t_i \mid i \in I]\,\{R\}}$$

$$\textsc{wp-refine}$$
$$(t_1 \preceq t_2)@i, \mathbf{wp}\,([i\colon t_2]\cdot \boldsymbol{t})\,\{Q\} \vdash \mathbf{wp}\,([i\colon t_1]\cdot \boldsymbol{t})\,\{Q\}$$

Fig. 7. Weakest precondition laws: lockstep rules.

4.1 Structural Rules

The rules in Fig. 6 represent basic inferences that are typically available in Hoare-style program logics. They hold generically on the hyper-term of the relevant WP: they apply independently of which components and which terms the hyper-term contains. The rules can be understood as axiomatic accounts of how the basic connectives commute with WP. Rule WP-TRIV states that a WP "commutes" with True: the WP with trivial postcondition is trivially true. This rule is sound for our model of WP that does not insist on safety of $\boldsymbol{t}$. Rule WP-CONS states that WP preserves entailment. This encodes the usual rule of consequence: if one can prove the WP with postcondition Q, the same WP with a weaker Q' is also provable. Rule WP-ALL states that WP commutes with $\forall$.

Rule WP-FRAME generalises Hoare's *frame* rule, also known as *constancy*. It shows that WP commutes with $(P\wedge)$ when P does not depend on the variables modified by $\boldsymbol{t}$. Note that P does not depend on the return values. Rule WP-IMPL-R is the analog of WP-FRAME for $(P \Rightarrow)$. Rule WP-SUBST allows the sound subsitution of values for program variables.

Rule WP-IDX explains how a bijective reindexing propagates through a WP, by applying the reindexing both to the term and to the postcondition. Combined with IDX, this effectively closes the proofs under all renamings, revealing an underlying symmetry of the entailment judgments (*i.e.*, no index is treated specially). We typically use the rule in concert with IDX and WP-CONS to uniformly rename the components across a judgment.

4.2 The Lockstep Rules

The rules in Fig. 7 are all straightforward extensions of the corresponding Hoare logic rules, to k-ary hyper-triples. Variations of these rules appear in virtually all other relational logics. In fact, one can recover Hoare logic by instantiating our rules to the 1-ary hyper-triple case. This embedding is more awkward to obtain in other relational logics that insist on a fixed arity greater than 1 for relational triples, e.g. [Benton 2004].

The rule WP-WHILE$_I$ rule follows the same pattern. The rule applies if all the components are while loops, and verifies their guards and their bodies as if they executed in lockstep. Exactly like

$$\textsc{wp-nest}\quad \mathbf{wp}\, t_1\, \{\lambda v.\, \mathbf{wp}\, t_2\, \{\lambda w.\, Q(v \cdot w)\}\} \dashv\vdash \mathbf{wp}\, (t_1 \cdot t_2)\, \{Q\}$$

$$\textsc{wp-conj}\quad \frac{\mathrm{idx}(Q_1) \cap \mathrm{supp}(t_2) \subseteq \mathrm{supp}(t_1) \qquad \mathrm{idx}(Q_2) \cap \mathrm{supp}(t_1) \subseteq \mathrm{supp}(t_2)}{\mathbf{wp}\, t_1\, \{Q_1\} \wedge \mathbf{wp}\, t_2\, \{Q_2\} \vdash \mathbf{wp}\, (t_1 + t_2)\, \{Q_1 \wedge Q_2\}}$$

$$\textsc{wp-proj}\quad \frac{}{\Pi_I.\, (\mathrm{proj}(t_2) \Rrightarrow \mathrm{proj}(t_1) \wedge \mathbf{wp}\, (t_1 \cdot t_2)\, \{Q\}) \vdash \mathbf{wp}\, t_2\, \{\hat{\Pi}_I.\, Q\}}\ I = \mathrm{supp}(t_1)$$

Fig. 8. Weakest precondition laws: hyper-structure rules.

its Hoare logic counterpart, the rule is based on loop invariants: here P is the (relational) loop invariant. The premise asks to prove, assuming the loop invariant holds initially, that after the evaluation of all the guards, we only have two cases. The first case is where all the guards evaluated to false ($\boldsymbol{b} =_I 0$) and the overall postcondition R holds. The second case is where all the guards evaluated to true ($\boldsymbol{b} \neq_I 0$). In that case we also have to prove that running all the loop bodies once results in re-establishing the loop invariant P. Note that the disjunction does not allow for the guards to go "out of sync": the loops execute exactly the same number of times.

The lockstep principle is very advantageous when it applies, but its applicability is very restricted: most often the control paths taken in two components will differ to a point where this strategy cannot be used or becomes counterproductive. Overcoming the rigidity of the basic lockstep proof strategy has been a goal of many proposals in the literature. For example, [Barthe et al. 2011, 2017] include a number of semantic-preserving transformations, like loop unrolling and loop splitting, that can be applied to terms so that they are brought to a shape amenable to application of lockstep rules. Similarly, [Sousa and Dillig 2016] provide a set of rules, dubbed "Cartesian Loop Logic", that perform a limited set of such transformations to terms. In [Barthe et al. 2017] a generalized while rule allows stuttering in the alignment of the (two) loops considered, without having to syntactically rewrite the terms. To express these non-trivial alignments, we include in our logic the WP-REFINE rule, which allows to replace a term t_1 in a WP, with another term t_2 if every behaviour of t_1 is also a behaviour of t_2.

Definition 4.2 (Refinement). The refinement $t_1 \preceq t_2$ holds when t_2 has all the behaviours of t_1:

$$t_1 \preceq t_2 \triangleq \lambda s.\, \left(\forall s', v.\, \langle t_1, s\rangle \Downarrow \langle v, s'\rangle \Rightarrow \langle t_2, s\rangle \Downarrow \langle v, s'\rangle\right)$$

Semantic equivalence is defined as $(t_1 \simeq t_2) \triangleq (t_1 \preceq t_2 \wedge t_2 \preceq t_1)$.

Proving $t_1 \preceq t_2$ generally requires meta-level reasoning about the semantics of the terms. The rule should therefore be considered as a last resort, and be used by instantiating the side condition with known generic refinements. Thanks to the flexibility of LHC, many challenging examples that in other logics would require ad-hoc refinements can still be handled within the logic, by only including the special case of the WP-REFINE rule instantiated with loop unfolding: **while** g **do** $t \simeq$ **if** g **then** $(t$;**while** g **do** $t)$. We show an example of this pattern in Section 5.2.

4.3 The Hyper-structure Rules

The rules in Fig. 8 are the key new rules of LHC. We call them *hyper-structure* rules because they decompose the goal by breaking up the components of a hyper-term. They represent three key reasoning principles available for WP on hyper-terms: how nested WPs can be merged into one

(WP-NEST), how a conjunction of WPs can be merged into one (WP-CONJ), and how to soundly remove components from a WP (WP-PROJ). They naturally arise from studying how WP commutes with other constructs, specifically, with another WP, with conjunction and with projection.

The WP-NEST rule states that a WP on a hyper-term that can be split into two disjoint hyper-terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, can be equivalently expressed as the nesting of a WP for $\boldsymbol{t}_2$ in the postcondition of the WP for $\boldsymbol{t}_1$. As we saw in Section 2.2, this rule increases the flexibility of the logic. As seen in Section 2.5, the nested WP pattern becomes essential when using the WP-WHILE$_I$ rule with a loop invariant that needs "side-computation" to be stated.

The WP-CONJ rule states that the conjunction of two WPs entails a single WP where the two postconditions are conjoined, and the hyper-terms are unioned. On an index $i \notin \mathrm{supp}(\boldsymbol{t}_1) \cap \mathrm{supp}(\boldsymbol{t}_2)$, the postcondition of the WP on $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ would observe different stores. The side conditions make sure Q_1 and Q_2 ignore the problematic indices.

In practice, WP-CONJ allows us to break down the current goal as a conjunction of two smaller hyper-triples. It is not mandatory for the hyper-terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ to have components in common, but the application of the rule is more powerful when they do. As a simple example, when $\boldsymbol{t}_1 = \boldsymbol{t}_2$, the rule allows the combination of multiple WPs on the same hyper-term. We have seen instances of this pattern in Section 2.

The embedding of Hoare logic together with WP-CONJ immediately entails a relative completeness result for our logic: given an oracle for derivations of Hoare triples, we can prove any hyper-triple by using WP-CONJ to compose the derivations for the strongest unary Hoare triple for each component, and then WP-CONS to imply the original goal (see [D'Osualdo et al. 2022] for details). This proof strategy, however, is the one that removes all opportunities for relational proofs, negating the benefits of working in a relational logic.

The last hyper-structure rule is WP-PROJ, that can be seen as a commutation rule between WP and projection. By using rules WP-PROJ and PROJ one can derive WP-PROJ$_i$ (shown in Section 2.3), which explains how to soundly remove some components from a WP. As illustrated in Section 2.3, and further discussed in Sections 5.2 and 5.3, WP-PROJ$_i$ is typically used in combination with WP-CONJ to introduce an auxiliary hyper-term $\boldsymbol{t}$, so that a goal that relates some $\boldsymbol{t}_1 \cdot \boldsymbol{t}_2$ can be broken into a goal that relates $\boldsymbol{t}_1 \cdot \boldsymbol{t}$, and one that relates $\boldsymbol{t}_2 \cdot \boldsymbol{t}$.

If we ignored the conjunct regarding the proj($\cdot$) assertion, the WP-PROJ rule would be unsound. The reason is that when even a single component of a WP diverges, the WP definition does not require the output stores of the other (terminating) components to satisfy the postcondition. For example, $\vdash$ **wp** [1: **skip**, 2: **while** 1 **do skip**] {False} holds. Projecting out component 2 blindly, however, would produce the invalid triple $\vdash$ **wp** [1: **skip**] {False}. To ensure soundness, WP-PROJ uses the *projectability* assertion.

Definition 4.3 (Projectable). The assertion $\mathrm{proj}(t)$ holds on states where t is *projectable*, *i.e.*, can yield a result: $\mathrm{proj}(t) \triangleq \lambda s.\, \langle t, s\rangle\!\Downarrow$. We also define an analogous hyper-assertion parameterized over hyper-terms: $\mathrm{proj}(\boldsymbol{t}) \triangleq \lambda \boldsymbol{s}.\, \langle \boldsymbol{t}, \boldsymbol{s}\rangle\!\Downarrow$.

To be able to apply WP-PROJ, one needs to establish that, if all the terms in $\boldsymbol{t}_2$ can terminate so can all the terms in $\boldsymbol{t}_1$ which we are projecting out. This would rule out our counterexample, and in fact is sufficient to prove soundness.

The projectability condition is strictly weaker than requiring termination of t', in two ways. First, it corresponds to "may" termination: it would hold if t' has both a diverging and a terminating trace from some initial store. For example, **while** $*$ **do skip** is projectable: from every initial store it has both diverging and terminating traces. Second, it is conditional on termination of the terms in $\boldsymbol{t}_2$. This can be useful when $\boldsymbol{t}_1(i)$ is a sub-term of some other component $\boldsymbol{t}_2(j)$, causing $\boldsymbol{t}_1(i)$ and $\boldsymbol{t}_2(j)$ to terminate under the same conditions.

WP-IDX-POST

$$\frac{\Gamma \vdash \mathbf{wp}\, \boldsymbol{t}\, \{Q\} \qquad j \notin \mathrm{supp}(\boldsymbol{t}) \cup \mathrm{idx}(\Gamma)}{\Gamma \vdash \mathbf{wp}\, \boldsymbol{t}\, \{Q(\!|i/j|\!)\}}$$

WP-IDX-SWAP

$$\frac{i \notin \mathrm{idx}(Q)}{\left(\mathbf{wp}\, ([j\colon t] \cdot \boldsymbol{t}')\, \{Q\}\right)(\!|i/j|\!) \vdash \mathbf{wp}\, ([i\colon t] \cdot \boldsymbol{t}')\, \{Q(\!|i/j|\!)\}}$$

WP-IDX-PASS

$$\frac{i, j \notin \mathrm{supp}(\boldsymbol{t})}{(\mathbf{wp}\, \boldsymbol{t}\, \{Q\})(\!|i/j|\!) \vdash \mathbf{wp}\, \boldsymbol{t}\, \{Q(\!|i/j|\!)\}}$$

WP-IDX-MERGE

$$\left(\mathbf{wp}\, ([i\colon t, j\colon t] \cdot \boldsymbol{t}')\, \{Q\}\right)(\!|i/j|\!) \vdash \mathbf{wp}\, ([i\colon t] \cdot \boldsymbol{t}')\, \{Q(\!|i/j|\!)\}$$

Fig. 9. Weakest precondition laws: reindexing rules.

4.4 The Reindexing Rules

Reindexing is a useful tool in relational proofs. One way of introducing a reindexing is by using rule IDX. While propagating the effects of a reindexing is straightforward for assertions with basic connectives (through the rules of Fig. 5), its interaction with WPs is much more interesting.

Rule WP-IDX already handles bijective reindexing. The rules in Fig. 9 represent the sound interactions between *non-bijective* reindexing and WPs. These reindexings boil down to compositions of reindexings of the form $(\!|i/j|\!)$ where $i \neq j$. The objective is to understand how a WP and a reindexing "commute": given $\mathbf{wp}\, \boldsymbol{t}\, \{Q\}(\!|i/j|\!)$, how does the reindexing propagate to $\boldsymbol{t}$ and Q? There are four cases to consider, depending on whether i and j are indices of $\boldsymbol{t}$ or not.

Rule WP-IDX-PASS deals with the simple case where $i, j \notin \mathrm{supp}(\boldsymbol{t})$, in which the reindexing has no effect on the hyper-term, and can be simply propagated to the postcondition.

Rule WP-IDX-SWAP handles the case where $j \in \mathrm{supp}(\boldsymbol{t})$ but $i \notin \mathrm{supp}(\boldsymbol{t})$. In this case $\boldsymbol{t} = [j\colon t] \cdot \boldsymbol{t}'$ for some t and $\boldsymbol{t}'$, with $i \notin \boldsymbol{t}'$ (the latter constraint is implied by well-formedness of the judgment in the rule). The rule then states that the reindexing is applied to the hyper-term by exchanging index j for index i and the reindexing is propagated to the postcondition. To be sound, the rule requires Q not to predicate on the index i: indeed in the starting WP, references to index i would refer to the initial store at i (since the hyper-term does not affect it), while in the resulting WP the store at i is modified by the hyper-term.

Rule WP-IDX-MERGE deals with the case where both $i, j \in \mathrm{supp}(\boldsymbol{t})$. In this case the effect on the hyper-term should be of "merging" the two components, which is only meaningful if they are mapped to the same term. Therefore the rule matches on $\boldsymbol{t} = [i\colon t, j\colon t] \cdot \boldsymbol{t}'$. The reindexing is then propagated to the postcondition. The intuition is that we have proved that Q holds on the results of any two runs of t, so it will hold when both runs are the same run (at a single index i).

The only missing case is when $i \in \mathrm{supp}(\boldsymbol{t})$ and $j \notin \mathrm{supp}(\boldsymbol{t})$. Since the reindexing does not affect the indices of $\boldsymbol{t}$, one might be tempted to write a rule like WP-IDX-PASS: $(\mathbf{wp}\, \boldsymbol{t}\, \{Q\})(\!|i/j|\!) \vdash \mathbf{wp}\, \boldsymbol{t}\, \{Q(\!|i/j|\!)\}$. Such rule would however be unsound. For example, we could start with the valid $\mathsf{x}(2) = 0 \vdash \mathbf{wp}\, [1\colon \mathsf{x} := 1]\, \{\mathsf{x}(1) = 1 \wedge \mathsf{x}(2) = 0\}$, apply IDX with $(\!|\pi|\!) = (\!|1/2|\!)$ to obtain the valid $\mathsf{x}(1) = 0 \vdash \left(\mathbf{wp}\, [1\colon \mathsf{x} := 1]\, \{\mathsf{x}(1) = 1 \wedge \mathsf{x}(2) = 0\}\right)(\!|1/2|\!)$. An application of our tentative rule would yield $\mathsf{x}(1) = 0 \vdash \mathbf{wp}\, [1\colon \mathsf{x} := 1]\, \{\mathsf{x}(1) = 1 \wedge \mathsf{x}(1) = 0\}$ which has an unsatisfiable postcondition (and is thus invalid). One simple side condition that would make the rule sound is $j \notin \mathrm{idx}(Q)$; this situation is however already handled by the IDX-IRREL rule.

Rule WP-IDX-POST represents a more useful way of dealing with the reindexing $(\!|i/j|\!)$ when $i \in \mathrm{supp}(\boldsymbol{t})$ and $j \notin \mathrm{supp}(\boldsymbol{t})$. The rule states that in such case it is possible to *introduce* the reindexing in the postcondition, provided the assumptions Γ do not constrain the store at j.

The soundness argument of WP-IDX-POST goes as follows. Let $\boldsymbol{s}$ be an input hyper-store satisfying Γ, and let $\boldsymbol{s}'[i\colon s]$ be the hyper-store resulting from running $\boldsymbol{t}$ on $\boldsymbol{s}$. To establish $Q(\!|i/j|\!)$ on $\boldsymbol{s}'[i\colon s]$, we need to prove Q holds on $\boldsymbol{s}'[i\colon s, j\colon s]$. Since Γ does not constrain the store at j, we

know it holds on $\boldsymbol{s}[j{:}s]$. Moreover, since $j \notin \mathrm{supp}(\boldsymbol{t})$, the hyper-term run from $\boldsymbol{s}[j{:}s]$ will output $\boldsymbol{s'}[i{:}s, j{:}s]$. The premise therefore implies Q holds on the output hyper-store. The crucial step in the proof is the act of feeding the output store at i in the conclusion, as the input store at j in the premise. This precisely captures the conversion from indirect-style triples to direct triples.

5 DISCUSSION

Here, we highlight some key features of LHC through some examples and a discussion of how certain design choices make this logic stand out. In the interest of space, the examples are presented in broad strokes. The extended version [D'Osualdo et al. 2022] contains their full proofs and additional case studies.

5.1 Modularity

```
f(x, y) ≜
  while (i < x) do
    r := op(r, y);
    i := i + 1
```

Fig. 10. A program that is parametric on op.

One of the main goals of LHC is allowing the construction of truly modular proofs. Consider for example the code in Fig. 10: the code uses a library-provided function op; f accumulates its output in r (assumed initially 0). Suppose we want to prove that f distributes over op in the second argument: $\mathsf{f}(a, \mathsf{op}(b,c)) = \mathsf{op}(\mathsf{f}(a,b), \mathsf{f}(a,c))$. This property holds if op is: (i) not modifying variables of f, (ii) a total function (*i.e.*, projectable and deterministic) with $\mathsf{op}(0,0) = 0$ (1-safety), (iii) associative (4-safety), and (iv) commutative (2-safety). Seeing as both the goal and these assumptions are hyper-safety properties, we would want to build a modular proof of distributivity of f: one that does not rely on the specific implementation of op, but only on the properties listed above expressed as hyper-triples.

In this example, the intuitive proof strategy is a vanilla lockstep alignment: if we consider the hyper-term $[1{:}\,\mathsf{f}(a,\mathsf{op}(b,c)), 2{:}\,\mathsf{f}(a,b), 3{:}\,\mathsf{f}(a,c)]$ all the components execute a iterations of their loops; intuitively, we should be able to prove the relational loop invariant $\mathsf{r}(1) = \mathsf{op}(\mathsf{r}(2), \mathsf{r}(3))$ with a lockstep proof. However, even though the high-level proof strategy of this example is a lockstep alignment, the lockstep-based logics are unable to provide a modular proof. There are two main obstacles. First, the loop invariant sketched above applies op and thus cannot be readily represented as an assertion. Second, the properties of op we want to rely on have mixed arities, and we would need to apply them to runs of op in the bodies of the loops, the initial call to $\mathsf{op}(b,c)$ and the one in the loop invariant, simultaneously.

By contrast, the ability of manipulating nested and mixed arity WPs of LHC provides a modular proof of this example. The goal can be expressed as:

$$\begin{pmatrix}\mathbf{wp}\,[4{:}\,\mathsf{op}(b,c)]\,\{\lambda \boldsymbol{r}.\,\boldsymbol{r}(4) = d\}\\ \mathsf{r}(1) = \mathsf{r}(2) = \mathsf{r}(3) = 0\\ \mathsf{i}(1) = \mathsf{i}(2) = \mathsf{i}(3)\end{pmatrix} \vdash \mathbf{wp}\begin{bmatrix}1{:}\,\mathsf{f}(a,d)\\ 2{:}\,\mathsf{f}(a,b)\\ 3{:}\,\mathsf{f}(a,c)\end{bmatrix}\left\{\begin{array}{l}\exists v_1, v_2, v_3.\\ \quad\begin{pmatrix}\mathsf{r}(1) = v_1 \wedge \mathsf{r}(2) = v_2 \wedge \mathsf{r}(3) = v_3\\ \mathbf{wp}\,[5{:}\,\mathsf{op}(v_2,v_3)]\,\{\lambda \boldsymbol{r}.\,\boldsymbol{r}(5) = v_1\}\end{pmatrix}\end{array}\right\}$$

Note the use of nested WPs to invoke op on b and c, and on the results of components 2 and 3.

Using the same style, we can represent (and prove!) the desired loop invariant:

$$\begin{pmatrix}\mathsf{i}(1) = \mathsf{i}(2) = \mathsf{i}(3)\\ \mathbf{wp}\,[4{:}\,\mathsf{op}(b,c)]\,\{\lambda \boldsymbol{r}.\,\boldsymbol{r}(4) = d\}\end{pmatrix} \wedge \exists v_1, v_2, v_3.\begin{pmatrix}\mathsf{r}(1) = v_1 \wedge \mathsf{r}(2) = v_2 \wedge \mathsf{r}(3) = v_3\\ \mathbf{wp}\,[5{:}\,\mathsf{op}(v_2,v_3)]\,\{\lambda \boldsymbol{r}.\,\boldsymbol{r}(5) = v_1\}\end{pmatrix}$$

5.2 Parsimonious Use of Refinement

Relational logics in the literature try to overcome some of the limitations of lockstep reasoning by using refinement to allow richer alignments between programs. Our observation is that the required refinements can be recast in many cases as hypersafety proofs. LHC is flexible enough to prove these refinements *within* the logic, and to apply them using the hyper-structure rules.

Consider again the program of Fig. 10 specialized with $\mathtt{op}(x, y) \triangleq x + y$. We can prove that the program enjoys distributivity on the *first* argument as well. In the form of a hyper-triple:

$$\vdash \left\{ \bigwedge_{j \in \{1,2,3\}} \mathtt{r}(j) = \mathtt{i}(j) = 0 \right\} \ [1{:}\, \mathtt{f}(a+b,c), 2{:}\, \mathtt{f}(a,c), 3{:}\, \mathtt{f}(b,c)] \ \left\{ \mathtt{r}(1) = \mathtt{r}(2) + \mathtt{r}(3) \right\} \tag{8}$$

All three components in the judgment are while loops, but they do not iterate the same number of times. We therefore cannot apply the lockstep while rule. Intuitively, we want to argue that the first a iterations of component 1 are matched exactly by component 2, and the remaining b iterations by component 3. To formally implement the above strategy, we replace component 1 with an equivalent program (here at index 4) that splits the while loop into two loops:

$$\vdash \left\{ \bigwedge_{j \in \{1,2,4\}} \mathtt{r}(j) = \mathtt{i}(j) = 0 \right\} \ [4{:}\, \mathtt{f}(a,c)\,;\mathtt{f}(a{+}b,c), 2{:}\, \mathtt{f}(a,c), 3{:}\, \mathtt{f}(b,c)] \ \left\{ \mathtt{r}(4) = \mathtt{r}(2) + \mathtt{r}(3) \right\} \tag{9}$$

The term at 4 may look like a typo, but it is not: f does not initialize its variables, so the second call will find in i the value left by the previous call. It is easy to discharge this version of the hyper-triple with a lockstep derivation.

To close the gap between (8) and (9) we need to prove the equivalence between component 1 and 4, which can be encoded by the auxiliary hyper-triple:

$$\vdash \begin{Bmatrix} \mathtt{r}(1) = \mathtt{r}(4) \\ \mathtt{i}(1) = \mathtt{i}(4) \end{Bmatrix} \begin{bmatrix} 1{:}\, \mathtt{f}(a+b\,,c) \\ 4{:}\, \mathtt{f}(a\,,c)\,;\mathtt{f}(a+b\,,c) \end{bmatrix} \begin{Bmatrix} \mathtt{r}(1) = \mathtt{r}(4) \\ \mathtt{i}(1) = \mathtt{i}(4) \end{Bmatrix} \tag{10}$$

Then we would be able to apply WP-PROJ and WP-CONS to replace component 4 for component 1 in our original goal:

$$\dfrac{\dfrac{(9)\ \vdash \{\ldots\}\ [4{:}\, \mathtt{f}(a,c)\,;\mathtt{f}(a{+}b,c), 2{:}\, \mathtt{f}(a,c), 3{:}\, \mathtt{f}(b,c)]\ \{\ldots\} \quad (10)\ \vdash \{\ldots\}\ [1{:}\, \mathtt{f}(a{+}b,c), 4{:}\, \mathtt{f}(a,c)\,;\mathtt{f}(a{+}b,c)]\ \{\ldots\}}{\vdash \{\ldots\}\ [1{:}\, \mathtt{f}(a{+}b,c), 2{:}\, \mathtt{f}(a,c), 3{:}\, \mathtt{f}(b,c), 4{:}\, \mathtt{f}(a,c)\,;\mathtt{f}(a{+}b,c)]\ \{\ldots\}}\ \textsc{wp-conj}}{(8)\ \vdash \{\ldots\}\ [1{:}\, \mathtt{f}(a{+}b,c), 2{:}\, \mathtt{f}(a,c), 3{:}\, \mathtt{f}(b,c)]\ \{\ldots\}}\ \textsc{wp-proj}$$

We have thus reduced the original goal to proving (10). The derivation, shown in [D'Osualdo et al. 2022], uses the WP-REFINE rule only once using the most basic refinement for while loops, loop unfolding: $\mathbf{while}\ g\ \mathbf{do}\ t \simeq \mathbf{if}\ g\ \mathbf{then}\ (t\,;\mathbf{while}\ g\ \mathbf{do}\ t)$.

As we noted in Section 4, the hyper-triple encoding of equivalence for non-deterministic programs is stronger than semantic equivalence. This means that for some programs the above proof pattern might not allow replacing some t_1 for some t_2, even when $t_1 \preceq t_2$ holds. A precise characterization of the limits of this proof pattern is an interesting direction of future research. For instance, one could ask which proofs are possible in our logic if the WP-REFINE rule is *replaced* by the special case of unfolding one iteration of a loop.

5.3 Divide and Conquer

The hyper-structure rules of LHC are a useful tool to decompose proofs into more tractable subgoals. The advantages of this are two-fold. First, in some cases, they permit the proof engineer to perform the formal decomposition following the same line of a natural informal argument. Second, smaller/simpler subgoals are more likely provable using off-the-shelf automatic provers.

Take again the example in Fig. 10, fixing $\mathtt{op}(x, y) \triangleq x + y$. We have shown how to prove the two distributivity properties, which we can informally summarize as $\mathtt{f}(a, b + c) = \mathtt{f}(a, b) + \mathtt{f}(a, c)$ and $\mathtt{f}(a + b, c) = \mathtt{f}(a, c) + \mathtt{f}(b, c)$. These two properties imply distributivity on *both* arguments: $\mathtt{f}(a + b, c + d) = \mathtt{f}(a, c) + \mathtt{f}(b, c) + \mathtt{f}(a, d) + \mathtt{f}(b, d)$. The intuitive argument breaks the property as the conjunction of $\mathtt{f}(a + b, c + d) = \mathtt{f}(a + b, c) + \mathtt{f}(a + b, d)$, $\mathtt{f}(a + b, c) = \mathtt{f}(a, c) + \mathtt{f}(b, c)$, and $\mathtt{f}(a + b, d) = \mathtt{f}(a, d) + \mathtt{f}(b, d)$, which are instances of the distributivity properties we already established. LHC can replicate the same simple argument, on the hyper-triple encodings of these

informal equations. The conjunction of the three equations above can be handled using WP-CONJ. Rule WP-PROJ can remove the auxiliary components running the terms $\mathtt{f}(a+b,c)$ and $\mathtt{f}(a+b,d)$ (see [D'Osualdo et al. 2022]). This illustrates how our logic allows the top-level goal to be decomposed into simpler goals within the logic, before the programs in the hyper-triples are even considered.

5.4 Hyper-triples as Goals and as Assumptions

Existing work on relational logics in the literature does not discuss the general problem of how to encode an arbitrary verification goal as a hyper-triple. Conceptually, the high-level goal can often be formulated as some semi-formal assertion involving "calls" to code (e.g. $\mathtt{f}(x) \le \mathtt{g}(x)$); then a formalization in terms of hyper-triples can be obtained by introducing a component for each "call" (e.g. $\vdash \mathbf{wp}\ [1{:}\,\mathtt{f}(x), 1{:}\,\mathtt{g}(x)]\ \{\lambda \boldsymbol{r}.\, \boldsymbol{r}(1) \le \boldsymbol{r}(2)\}$). While this procedure seems obvious enough in simple instances, it can be surprisingly ambiguous in general. Consider the case of idempotence, which we already partially addressed in Section 2.4. Informally, the goal looks like $t \sim (t\,;t)$, which can plausibly be formalized in (at least) three different ways:

$$\vdash \left\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2)\right\}\ [1{:}\,t, 2{:}\,(t\,;t)]\ \left\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2)\right\} \qquad (\textsc{IdemSeq}_t)$$

$$\vdash \left\{\vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(2) \wedge \vec{\mathtt{x}}(3) = \vec{v}\right\}\ [1{:}\,t, 2{:}\,t, 3{:}\,t]\ \left\{\vec{\mathtt{x}}(2) = \vec{v} \Rightarrow \vec{\mathtt{x}}(1) = \vec{\mathtt{x}}(3)\right\} \qquad (\textsc{Idem}^3_t)$$

$$\vdash \left\{\vec{\mathtt{x}}(2) = \vec{v}\right\}\ [1{:}\,t, 2{:}\,t]\ \left\{\vec{\mathtt{x}}(1) = \vec{v} \Rightarrow \vec{\mathtt{x}}(2) = \vec{v}\right\} \qquad (\textsc{Idem}_t)$$

The most verbatim translation of the informal goal is ($\textsc{IdemSeq}_t$). The hyper-triple ($\textsc{Idem}^3_t$) uses indirect-style to feed the output of 2 as the input of 3, and considers each occurrence of t in the informal equation $t \sim (t\,;t)$ as a separate component. Sousa and Dillig [2016] propose ($\textsc{Idem}_t$) which also uses indirect-style, but conflates components 1 and 2 of ($\textsc{Idem}^3_t$) into the single component 1.

A natural question arises about the differences between these formulations. Thanks to the generality of LHC rules, this question can be investigated within the logic. For example, we can prove: (i) ($\textsc{Idem}^3_t$) $\Rightarrow$ ($\textsc{Idem}_t$) (ii) ($\textsc{Idem}^3_t$) $\Rightarrow$ ($\textsc{IdemSeq}_t$) (iii) $\big(($\textsc{Idem}$_t) \wedge ($\textsc{Det}$_t)\big) \Rightarrow$ ($\textsc{Idem}^3_t$) (iv) $\big(($\textsc{Idem}$^3_t) \wedge \mathrm{proj}(t)\big) \Rightarrow$ ($\textsc{Det}_t$) where ($\textsc{Det}_t$) is the hyper-triple asserting determinism of t on $\vec{\mathtt{x}}$. All the implications in the other directions do not hold. All the details are in [D'Osualdo et al. 2022].

As we explained in Section 2.4, even in the cases where the specifications are equivalent, there is a tension between which hyper-triples are easier to *prove*, *i.e.*, they are better as goals, and the ones which are easier to *use*, *i.e.*, they are better as assumptions. There is no silver bullet: different proofs may need idempotence to be expressed as relating occurrences of t in the same component, or two or three components. If the logic cannot perform the inter-derivations we just examined, choosing one specification over the other may make it impossible to prove some valid goals. LHC is unique among relational logics in supporting the derivation of one specification from the other.

5.5 Hyper-triple Semantics and Termination

Some relational logics, notably [Benton 2004; Yang 2007], propose an "equi-termination" semantics: given a hyper-store satisfying the precondition, a component can diverge only if all the others do. In a language with non-determinism, the more appropriate semantics would be "*may* equi-termination": if a component *may* terminate, then all the others *may* terminate as well. Notice that the (may) equi-termination semantics encodes a property that goes beyond k-safety: it quantifies over traces existentially.

The equi-termination semantics has some useful consequences for the rules. For example, the WP-PROJ rule would not need the projectability side condition. The WP-CONJ rule, on the other hand, would only be sound if $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ overlap on at least one component. In fact the [R-Tr] rule of [Benton 2004] can be seen as a binary special case of a combination of WP-PROJ and overlapping WP-CONJ which is only sound in the equi-termination semantics.

The equi-termination semantics nicely supports the soundness of the lockstep WP-WHILE$_I$ rule: if the guards are always either all true or all false, the while loops are equi-terminating; note that this is weaker than proving termination of the loops (no variant-based reasoning is needed).

The equi-termination semantics, however, does not validate some important rules like the left to right direction of WP-NEST, and the WP-IDX-POST rule.

A more powerful extension of our logic with support for equi-termination WPs would annotate hyper-triples with an equivalence relation $\approx$ on its indexes. An assertion $\mathbf{wp}_{\approx}\, \boldsymbol{t}\, \{Q\}$ would—in addition to the constraints of the current definition—require that, for every hyper-store $\boldsymbol{s}$ satisfying P, if $\langle \boldsymbol{t}(i), \boldsymbol{s}(i)\rangle\Downarrow$ and $i \approx j$, then $\langle \boldsymbol{t}(j), \boldsymbol{s}(j)\rangle\Downarrow$. This would make it feasible to derive the most precise equi-termination guarantees each rule can provide, given the equivalence relations that hold in the premises. In WP-PROJ, the projectability side condition could be then replaced by $\forall i \in I.\, \exists j \notin I. i \approx j$. More ambitiously, as we speculate in Section 7, termination-related properties could become first-class citizens in a logic for *hyperliveness*.

6 RELATED WORK

There is a large body of work on relational verification. Here, we survey and compare against the most closely related work.

Relational Hoare Logic and product programs. The seminal Relational Hoare Logic (RHL) of [Benton 2004] is a program logic with judgments on pairs of programs supporting variations of lockstep proofs. RHL and its extension to Separation Logic [Yang 2007] have been used to prove a wide range of relational properties of first-order imperative programs, from correctness of optimisations to information-flow. Barthe et al. [2004] introduced self-composition as a relatively complete method to reason about properties like non-interference and information flow, an approach further developed in Beringer [2011]; Beringer and Hofmann [2007]; Darvas et al. [2005]; Terauchi and Aiken [2005]. Building on this idea, Barthe et al. [2011] proposed product programs as a way to reduce relational verification of 2-properties to standard Hoare logic reasoning on a single program, without losing the relational flavour of the proofs. The technique has been incorporated in program logics and extended to handle probabilistic properties [Barthe et al. 2017, 2013] like differential privacy or security of cryptographic code. Kovács et al. [2013] incorporates this approach in abstract interpretation. Cartesian Hoare Logic (CHL) [Sousa and Dillig 2016] is an extension of RHL to handle k-safety properties, for fixed k. The utility of going beyond 2-properties is shown by providing hyper-triple-like specifications for common correctness checks that arise in application domains like comparators of user-defined types, *e.g.*, transitivity, associativity, and commutativity. The problem of alignment of while loops is addressed through a limited set of rules that implicitly construct product programs through derivation sequences. Godlin and Strichman [2013] proposed *regression verification* for proving equivalence between two *similar* programs using automatically inferred alignments.

We build on these logics by introducing patterns of reasoning that involve hyper-triples of mixed arities—i.e. transforming a hyper-triple goal of arity k into hyper-triple subgoals with arities potentially different from k. No prior relational logics support such reasoning. See Section 4 and Section 5.5 for a more technical distinction between LHC's design decisions and other logics.

We briefly commented on relative completeness of LHC in Section 4.3. Nagasamudram and Naumann [2021] studies completeness of various combinations of enhanced lockstep binary rules in terms of the classes of product programs (called alignments) they can represent. Characterising, in the same spirit, the new classes of alignments expressible through LHC, is interesting future work.

Logics for proving refinement. Another set of program logics commonly labelled as "relational" are the ones which aim to prove refinement between two programs, often with a focus on concurrency, e.g. [Frumin et al. 2018; Liang et al. 2012]. Refinement is a property that quantifies both universally and existentially over traces, and is thus not a hypersafety property (since the latter quantifies universally over all traces). Moreover, all refinement judgments are between exactly two programs. However, since proving refinement often retains the "relational" flavour of proofs that we seek in proving hypersafety, exploring a single logic encompassing both kinds of relational reasoning is an interesting path for future research—see Section 7.

Higher-order logics. Relational proofs can also be conducted in a higher-order logic framework. Approaches like Relational Higher-Order Logic (RHOL) [Aguirre et al. 2019] propose rules that can derive relational proofs of 2-safety properties of pure higher-order functional programs. Notably, RHOL can embed various quantitative/probabilistic extensions as special cases of the general framework. This approach tends to work well for programs that can be expressed directly as pure expressions in the host meta-logic (e.g. Coq). LHC obtains compositional relational reasoning in a *first-order* program logic that supports direct treatment of non-deterministic impure code. We establish WPs on hyper-terms as the right building block to embed such programs in assertions.

Automation. While we do not consider automation in this paper, part of the motivation for considering relational proofs is that they are more feasible for automation, whereas functional-specification-based proofs of the same properties would be completely out of reach of current algorithms. Consider, as an example, the relational proof of distributivity of multiplication (e.g. our running example `f` when `op` is addition). The proof can be done by exclusively using conjunctions of equalities and additions, which is within the power of Presburger arithmetic, a decidable theory. A functional-specification-based proof of the same property would require the use of mathematical multiplication in assertions, which requires the use of undecidable Peano arithmetic.

This observation has been exploited in various works that use product programs [Barthe et al. 2011; Eilers et al. 2020]. In [Farzan and Vandikas 2019] this idea is taken even further by showing that it is possible to automate the *search of appropriate product programs* itself, even in the more general case of k-safety. Shemer et al. [2019] and Unno et al. [2021] push the boundaries of automatically inferrable enhanced lockstep proofs, with the latter supporting hyper-liveness too.

All these works are based on whole-program enhanced lockstep proofs, leading to searches that are global and do not scale well beyond small programs. Our logic outlines how a monolithic task like this can be effectively decomposed. It would be possible to integrate these tools in our logic with the goal of discharging as many sub-hypertriples involved in a proof as possible. The compositionality achieved by our proof system can make these tools applicable to smaller, more tractable instances, increasing the scalability of the overall method.

Model checking. HyperLTL and HyperCTL* [Clarkson et al. 2014] are temporal logics which have been proposed as specification formalisms for hyperproperties (with k-safety as a special case). MultiLTL [Goudsmid et al. 2021] extends the idea to the case where a hyperproperty property involves k different finite-state models. They target finite-state systems, and therefore substantially differ in scope compared to this work. Temporal logics specify hyperproperties as global properties of sets of traces, which does not lead naturally to the kind of decomposition of proofs that we seek to obtain with LHC. The hope is that compositional program logics like ours can inspire interesting decomposition heuristics for these monolithic verification tasks in the same style that compositional concurrent program logics inspired algorithms for model checking of concurrent systems [Flanagan et al. 2002; Gupta et al. 2011; McMillan 1999].

7 CONCLUSION AND FUTURE WORK

We introduced LHC, proved it sound, and showed how a handful of new rules can unlock important compositional proof principles for hypersafety.

LHC is just a first step in exploring the change of perspective of embracing hyper-triples as proof building blocks. A first interesting line for future work is to investigate how the compositionality and locality dimension added to proofs by Separation Logic [Yang 2007] interacts with the hyper-triple compositions of LHC. In addition to supporting proofs of heap-manipulating programs, such an extension could pave the way to support for parallel composition.

Hypersafety properties such as determinism, non-interference, and correctness of parallelizing optimizations are examples of important applications of an extension of LHC supporting concurrency. The key challenge would be to blend rely/guarantee-style reasoning together with the relational flavour of hypersafety verification. In particular, obtaining rely/guarantee proofs that avoid the use of strong functional specifications looks like a very intriguing research problem.

An important next step is exploring automation. The rules of LHC generate a search space that is different from those of previous automated work (*e.g.*, [Farzan and Vandikas 2019; Unno et al. 2021]) in two key senses: (1) As a consequence of expressivity of LHC, the search space is larger; and (2) as a counter effect to (1), its structure offers enticing opportunities for proof reuse and compositional arity-changing proof patterns. None of the heuristics used in previous relational logics handle this dimension of the search.

An even more ambitious direction for future research is the extension of LHC to reason about hyperproperties beyond hypersafety, such as *hyperliveness* [Clarkson and Schneider 2008]. For example, it would be interesting to extend hyper-triples from hypersafety statements of the form $\forall^n$ to hyperliveness statements of the form $\forall^n\exists^m$. As an enticing by-product, such a logic would embed both n-safety triples *and refinement* (which is a $\forall\exists$ hyperproperty) in a single generalized judgment. Understanding the proof principles available for such a judgment would be a way to unify hypersafety-based and refinement-based relational reasoning.

ACKNOWLEDGMENTS

The authors would like to thank Deepak Garg and Viktor Vafeiadis for their useful comments on an early version of the paper.

This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2019. A relational logic for higher-order programs. *J. Funct. Program.* 29 (2019), e16. https://doi.org/10.1017/S0956796819000145

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

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *J. Log. Algebraic Methods Program.* 85, 5 (2016), 847–859. https://doi.org/10.1016/J.JLAMP.2016.05.004

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *CSFW*. IEEE Computer Society, 100–114. https://doi.org/10.1109/CSFW.2004.17

Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *POPL*. ACM, 161–174. https://doi.org/10.1145/3009837.3009896

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013), 9:1–9:49. https://doi.org/10.1145/2492061

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL*. ACM, 14–25. https://doi.org/10.1145/964001.964003

Lennart Beringer. 2011. Relational Decomposition. In *ITP*. 39–54. https://doi.org/10.1007/978-3-642-22863-6_6

Lennart Beringer and Martin Hofmann. 2007. Secure information flow and program logics. In *CSF*. IEEE Computer Society, 233–248. https://doi.org/10.1109/CSF.2007.30

Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *POST (Lecture Notes in Computer Science, Vol. 8414)*. Springer, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *CSF*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2008.7

Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3450)*. Springer, 193–209. https://doi.org/10.1007/978-3-540-32004-3_20

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving Hypersafety Compositionally (Extended Version). *CoRR* arxiv:2209.07448 (2022). https://doi.org/10.48550/arxiv.2209.07448

Marco Eilers, Peter Müller, and Samuel Hitz. 2020. Modular Product Programs. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 3:1–3:37. https://doi.org/10.1145/3324783

Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *CAV (1) (Lecture Notes in Computer Science, Vol. 11561)*. Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

Cormac Flanagan, Shaz Qadeer, and Sanjit A. Seshia. 2002. A Modular Checker for Multithreaded Programs. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*. Springer, 180–194. https://doi.org/10.1007/3-540-45657-0_14

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. ACM, 442–451. https://doi.org/10.1145/3209108.3209174

Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab.* 23, 3 (2013), 241–258. https://doi.org/10.1002/stvr.1472

Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. 2021. Compositional Model Checking for Multi-properties. In *VMCAI (Lecture Notes in Computer Science, Vol. 12597)*. Springer, 55–80. https://doi.org/10.1007/978-3-030-67067-2_4

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*. Springer, 412–417. https://doi.org/10.1007/978-3-642-22110-1_32

Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. 2013. Relational abstract interpretation for the verification of 2-hypersafety properties. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 211–222. https://doi.org/10.1145/2508859.2516721

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*. ACM, 455–468. https://doi.org/10.1145/2103656.2103711

Kenneth L. McMillan. 1999. Circular Compositional Reasoning about Liveness. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1703)*. Springer, 342–345. https://doi.org/10.1007/3-540-48153-2_30

Ramana Nagasamudram and David A. Naumann. 2021. Alignment Completeness for Relational Hoare Logics. In *LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470690

Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *CAV (1) (Lecture Notes in Computer Science, Vol. 11561)*. Springer, 161–179. https://doi.org/10.1007/978-3-030-25540-4_9

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare logic for verifying k-safety properties. In *PLDI*. ACM, 57–69. https://doi.org/10.1145/2908080.2908092

Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *SAS*. 352–367. https://doi.org/10.1007/11547662_24

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 742–766. https://doi.org/10.1007/978-3-030-81685-8_35

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