# Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties

ANONYMOUS AUTHOR(S)

Hoare logics are proof systems that allow one to formally establish properties of computer programs. Traditional Hoare logics prove properties of individual program executions (so-called trace properties, such as functional correctness). Hoare logic has been generalized to prove also properties of multiple executions of a program (so-called hyperproperties, such as determinism or non-interference). These program logics prove the *absence* of (bad combinations of) executions. On the other hand, program logics similar to Hoare logic have been proposed to *disprove* program properties (e.g., Incorrectness Logic), by proving the *existence* of (bad combinations of) executions. All of these logics have in common that they specify program properties using assertions over a fixed number of states, for instance, a single pre- and post-state for functional properties or pairs of pre- and post-states for non-interference.

In this paper, we present Hyper Hoare Logic, a generalization of Hoare logic that lifts assertions to properties of arbitrary *sets* of states. The resulting logic is simple yet expressive: its judgments can express arbitrary trace- and hyperproperties over the terminating executions of a program. By allowing assertions to reason about sets of states, Hyper Hoare Logic can reason about both the *absence* and the *existence* of (combinations of) executions, and, thereby, supports both proving and disproving program (hyper-)properties within the same logic, including (hyper-)properties that no existing Hoare logic can express. We prove that Hyper Hoare Logic is sound and complete, and demonstrate that it captures important proof principles naturally. All our technical results have been proved in Isabelle/HOL.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**.

Additional Key Words and Phrases: Hyperproperties, Program Logic, Incorrectness Logic

## 1 INTRODUCTION

Hoare Logic [Floyd 1967; Hoare 1969] is a logic designed to formally prove functional correctness of computer programs. It enables proving judgments (so-called *Hoare triples*) of the form $\{P\}\ C\ \{Q\}$, where $C$ is a program command, and $P$ (the *precondition*) and $Q$ (the *postcondition*) are assertions over execution states. The Hoare triple $\{P\}\ C\ \{Q\}$ is valid if and only if executing $C$ in an initial state that satisfies $P$ can only lead to final states that satisfy $Q$.

Hoare Logic is widely used to prove the absence of runtime errors, functional correctness, resource bounds, etc. All of these properties have in common that they are properties of *individual* program executions (so-called *trace properties*). However, classical Hoare Logic cannot reason about properties of *multiple* program executions (so-called *hyperproperties* [Clarkson and Schneider 2008]), such as determinism (executing the program twice in the same initial state results in the same final state) or information flow security, which is often phrased as non-interference [Volpano et al. 1996] (executing the program twice with the same low-sensitivity inputs results in the same low-sensitivity outputs). To overcome such limitations and to reason about more types of properties, Hoare Logic has been extended and adapted in various ways. We refer to those extensions and adaptations collectively as *Hoare logics*.

Among them are several logics that can establish properties of two [Aguirre et al. 2017; Amtoft et al. 2006; Benton 2004; Costanzo and Shao 2014; Eilers et al. 2023; Ernst and Murray 2019; Francez 1983; Maillard et al. 2019; Naumann 2020; Yang 2007] or even $k$ [D'Osualdo et al. 2022; Sousa and Dillig 2016] executions of the same program, where $k > 2$ is useful for properties such as transitivity and associativity. *Relational Hoare logics* are able to prove *relational properties*, i.e., properties relating executions of two (potentially different) programs, for instance, to prove program equivalence.

2024. 2475-1421/2024/6-ART1 $15.00
https://doi.org/

| | Number of executions | | | |
|---|---|---|---|---|
| **Type** | 1 | 2 | $k$ | $\infty$ |
| Overapproximate (hypersafety) | ✓ HL, OL, RHL, CHL, RHLE, MHRM | ✓ RHL, CHL, RHLE, MHRM | ✓ CHL, RHLE | ✓ ∅ |
| Backward underapproximate | ✓ IL, InSec | ✓ InSec | ✓ ∅ | ✓ ∅ |
| Forward underapproximate | ✓ OL, RHLE, MHRM | ✓ RHLE, MHRM | ✓ RHLE | ✓ ∅ |
| $\forall^*\exists^*$ | not applicable | ✓ RHLE, MHRM | ✓ RHLE | ✓ ∅ |
| $\exists^*\forall^*$ | not applicable | ✓ ∅ | ✓ ∅ | ✓ ∅ |
| Set properties | not applicable | not applicable | not applicable | ✓ ∅ |

Fig. 1. (Non-exhaustive) overview of Hoare logics, classified in two dimensions: The type of properties a logic can establish, and the number of program executions these properties can relate (column "$\infty$" subsumes an unbounded and an infinite number of executions). We explain the distinction between backward and forward underapproximate properties in App. C.2. $\forall^*\exists^*$- and $\exists^*\forall^*$-hyperproperties are discussed in Sect. 2. App. B gives examples of (hypersafety and set) properties for an unbounded number of executions. A green checkmark indicates that a property is handled by our Hyper Hoare Logic for the programming language defined in Sect. 3.1, and ∅ indicates that no other Hoare logic supports it. The acronyms refer to the following. CHL: Cartesian Hoare Logic [Sousa and Dillig 2016], HL: Hoare Logic [Floyd 1967; Hoare 1969], IL: Incorrectness Logic [O'Hearn 2019] or Reverse Hoare Logic [de Vries and Koutavas 2011], InSec: Insecurity Logic [Murray 2020], OL: Outcome Logic [Zilberstein et al. 2023], RHL: Relational Hoare Logic [Benton 2004], RHLE [Dickerson et al. 2022], MHRM [Maillard et al. 2019].

All of these logics have in common that they can prove only properties that hold *for all* (combinations of) executions, that is, they prove the *absence* of bad (combinations of) executions; to achieve that, their judgments *overapproximate* the possible executions of a program. Overapproximate logics cannot prove the *existence* of (combinations of) executions, and thus cannot establish certain interesting program properties, such as the presence of a bug or non-determinism.

To overcome this limitation, recent work [de Vries and Koutavas 2011; Murray 2020; O'Hearn 2019; Raad et al. 2020, 2022] proposed Hoare logics that can prove the *existence* of (individual) executions, for instance, to *disprove* functional correctness. We call such Hoare logics *underapproximate*. Tools based on underapproximate Hoare logics have proven useful for finding bugs on an industrial scale [Blackshear et al. 2018; Distefano et al. 2019; Gorogiannis et al. 2019; Le et al. 2022]. More recent work [Dickerson et al. 2022; Maksimović et al. 2023; Zilberstein et al. 2023] has proposed Hoare logics that combine underapproximate and overapproximate reasoning.

*The problem.* Fig. 1 presents a (non-exhaustive) overview of the landscape of Hoare logics, where logics are classified in two dimensions: the type of properties they can establish, and the number of program executions those properties can relate. The table reveals two open problems. First, some types of hyperproperties cannot be expressed by any existing Hoare logic (represented by ∅). For example, to prove that a program implements a function that has a minimum, one needs to show that there *exists* an execution whose result is smaller than or equal to the result of *all* other executions. Such $\exists\forall$-hyperproperties cannot be proved by any existing Hoare logic. Second, the existing logics cover different, often disjoint program properties, which may hinder practical applications: reasoning about a wide spectrum of properties of a given program requires the application of several logics, each with its own judgments; properties expressed in different, incompatible logics cannot be composed within the same proof system.

*This work.* We present *Hyper Hoare Logic*, a novel Hoare logic that enables *proving or disproving* any (trace or) hyperproperty over the set of terminating executions of a program. As indicated by the green checkmarks in Fig. 1, these include many different types of properties, relating *any* (potentially unbounded or even infinite) number of program executions, and many hyperproperties that no existing Hoare logic can handle. Among them are $\exists^*\forall^*$ hyperproperties such as violations of generalized non-interference (Sect. 4.3) and the existence of a minimum (Sect. 5.3), and hyperproperties relating an unbounded or infinite number of executions such as quantifying information flow with min-capacity [Assaf et al. 2017; Smith 2009; Yasuoka and Terauchi 2010] (App. B).

Hyper Hoare Logic is based on a simple yet powerful idea: We lift pre- and postconditions from assertions over a *fixed* number of execution states to *hyper-assertions* over *sets* of execution states. Hyper Hoare Logic then establishes *hyper-triples* of the form $\{P\}\ C\ \{Q\}$, where $P$ and $Q$ are hyper-assertions. Such a hyper-triple is valid iff for any set of initial states $S$ that satisfies $P$, the set of all final states that can be reached by executing $C$ in some state from $S$ satisfies $Q$. By allowing assertions to quantify *universally* over states, Hyper Hoare Logic can express overapproximate properties, whereas *existential* quantification expresses underapproximate properties. Combinations of universal and existential quantification in the same assertion, as well as assertions over infinite sets of states, allow Hyper Hoare Logic to prove or disprove properties beyond existing logics.

*Contributions.* Our main contributions are:

- We present Hyper Hoare Logic, a novel Hoare logic that can prove or disprove arbitrary hyperproperties over terminating executions.
- We formalize our logic and prove soundness and completeness in Isabelle/HOL [Nipkow et al. 2002].
- We derive easy-to-use syntactic rules for a restricted class of *syntactic* hyper-assertions, as well as additional loop rules that capture different reasoning principles.
- We prove compositionality rules for hyper-triples, which enable the flexible composition of hyper-triples of different forms and, thus, facilitate modular proofs.
- We demonstrate the expressiveness of Hyper Hoare Logic, both on judgments of existing Hoare logics and on hyperproperties that no existing Hoare logic supports.

*Outline.* Sect. 2 informally presents hyper-triples, and shows how they can be used to specify hyperproperties. Sect. 3 introduces the rules of Hyper Hoare Logic, and proves that these rules are sound and complete for establishing valid hyper-triples. Secs. 4 and 5 derive additional rules that enable concise proofs in common cases. We discuss related work in Sect. 6 and conclude in Sect. 7. The appendix contains further details and extensions. In particular, App. C shows how to express judgments of existing logics in Hyper Hoare Logic, and App. D presents compositionality rules. **All our technical results (Secs. 3, 4, 5, and the appendix) have been proved in Isabelle/HOL [Nipkow et al. 2002]; the mechanization has been submitted as supplementary material.**

## 2 HYPER-TRIPLES, INFORMALLY

In this section, we illustrate how hyper-triples can be used to express different types of hyperproperties, including over- and underapproximate hyperproperties for single (Sect. 2.1) and multiple (Sect. 2.2 and Sect. 2.3) executions.

### 2.1 Overapproximation and Underapproximation

Consider the command $C_0 \triangleq (x := \mathit{randIntBounded}(0, 9))$, which generates a random integer between 0 and 9 (both included), and assigns it to the variable $x$. Its functional correctness properties include: (P1) The final value of $x$ is in the interval $[0, 9]$, and (P2) every value in $[0, 9]$ can occur for every initial state (i.e., the output is not determined by the initial state).

Property P1 expresses the *absence* of bad executions, in which the output $x$ is outside the interval $[0, 9]$. This property can be expressed in classical Hoare logic, with the triple $\{\top\}\ C_0\ \{0 \leq x \leq 9\}$. In Hyper Hoare Logic, where assertions are properties of sets of states, we express it using a postcondition that *universally* quantifies over all possible final states: In all final states, the value of

$x$ should be in $[0, 9]$. The hyper-triple $\{\top\}\ C_0\ \{\forall\langle\varphi'\rangle.\, 0 \leq \varphi'(x) \leq 9\}$ expresses this property. The postcondition, written in the syntax that will be introduced in Sect. 4, is semantically equivalent to $\{\lambda S'.\, \forall\varphi' \in S'.\, 0 \leq \varphi'(x) \leq 9\}$. This hyper-triple means that, for any set $S$ of initial states $\varphi$ (satisfying the trivial precondition $\top$), the set $S'$ of all final states $\varphi'$ that can be reached by executing $C_0$ in some initial state $\varphi \in S$ satisfies the postcondition, i.e., all final states $\varphi' \in S'$ have a value for $x$ between 0 and 9. This hyper-triple illustrates a systematic way of expressing classical Hoare triples as hyper-triples (see App. C.1).

Property P2 expresses the *existence* of desirable executions and can be expressed using an underapproximate Hoare logic. In Hyper Hoare Logic, we use a postcondition that *existentially* quantifies over all possible final states: For each $n \in [0, 9]$, there exists a final state where $x = n$. The hyper-triple $\{\exists\langle\varphi\rangle.\, \top\}\ C_0\ \{\forall n.\, 0 \leq n \leq 9 \Rightarrow \exists\langle\varphi'\rangle.\, \varphi'(x) = n\}$ expresses P2. The precondition is semantically equivalent to $(\lambda S.\, \exists\varphi \in S)$. It requires the initial set of states $S$ to be non-empty (otherwise the set of states reachable from states in $S$ by executing $C_0$ would also be empty, and the postcondition would not hold). The postcondition ensures that, for any $n \in [0, 9]$, it is possible to reach at least one state $\varphi'$ with $\varphi'(x) = n$.

This example shows that hyper-triples can express both under- and overapproximate properties, which allows Hyper Hoare Logic to reason about both the *absence* of bad executions and the *existence* of good executions. Moreover, hyper-triples can also be used to prove the existence of *incorrect* executions, which has proven useful in practice to find bugs without false positives [Le et al. 2022; O'Hearn 2019]. To the best of our knowledge, the only other Hoare logics that can express both properties P1 and P2 are Outcome Logic [Zilberstein et al. 2023] and Exact Separation Logic [Maksimović et al. 2023].[1] However, these logics are limited to properties of single executions and, thus, cannot handle hyperproperties such as the examples we discuss next.

## 2.2 (Dis-)Proving $k$-Safety Hyperproperties

A $k$-safety hyperproperty [Clarkson and Schneider 2008] is a property that characterizes *all combinations of* $k$ executions of the same program.

An important example is information flow security, which requires that programs that manipulate secret data (such as passwords) do not expose secret information to their users. In other words, the content of high-sensitivity (secret) variables must not leak into low-sensitivity (public) variables. For deterministic programs, information flow security is often formalized as *non-interference* (NI) [Volpano et al. 1996], a 2-safety hyperproperty: Any two executions of the program with the same low-sensitivity (*low* for short) inputs (but potentially different high-sensitivity inputs) must have the same low outputs. That is, for all pairs of executions $\tau_1, \tau_2$, if $\tau_1$ and $\tau_2$ agree on the initial values of all low variables, they must also agree on the final values of all low variables. This ensures that the final values of low variables are not influenced by the values of high variables. Assuming for simplicity that we have only one low variable $l$, the hyper-triple $\{low(l)\}\ C_1\ \{low(l)\}$, where $low(l) \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(l) = \varphi_2(l))$, expresses that $C_1$ satisfies NI: If all states in $S$ have the same value for $l$, then all final states reachable by executing $C_1$ in any initial state $\varphi \in S$ will have the same value for $l$. Note that this set-based definition is equivalent to the standard definition based on pairs of executions. In particular, instantiating $S$ with a set of two states directly yields the standard definition.

Non-interference requires that all final states have the same value for $l$, irrespective of the initial state that leads to any given final state. Other $k$-safety hyperproperties need to relate initial and final states. For example, the program $y := f(x)$ is *monotonic* iff for any two executions with

[1]While RHLE [Dickerson et al. 2022] can in principle reason about the existence of executions, it is unclear how to express the existence *for all* numbers $n$.

$\varphi_1(x) \geq \varphi_2(x)$, we have $\varphi'_1(y) \geq \varphi'_2(y)$, where $\varphi_1$ and $\varphi_2$ are the initial states $\varphi'_1$ and $\varphi'_2$ are the *corresponding* final states.

To relate initial and final states, Hyper Hoare Logic uses *logical variables* (also called *auxiliary variables* [Kleymann 1999]). These variables cannot appear in a program, and thus are guaranteed to have the same values in the initial and final states of an execution. We use this property to tag corresponding states, as illustrated by the hyper-triple for monotonicity: $\{mono^t_x\}$ $y := f(x)$ $\{mono^t_y\}$, where $mono^t_x \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(t) = 1 \land \varphi_2(t) = 2 \Rightarrow \varphi_1(x) \geq \varphi_2(x))$. Here, $t$ is a logical variable used to distinguish the two executions of the program.

*Disproving k-safety hyperproperties.* As explained in the introduction, being able to prove that a property does *not* hold is valuable in practice, because it allows building tools that can find bugs without false positives. Hyper Hoare Logic is able to *disprove* hyperproperties by proving a hyperproperty that is essentially its negation. For example, we can prove that the insecure program $C_2 \triangleq (\textbf{if } (h > 0)\ \{l := 1\} \textbf{ else } \{l := 0\})$, where $h$ is a high variable, *violates* non-interference (NI), using the following hyper-triple: $\{low(l) \land (\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(h) > 0 \land \varphi_2(h) \leq 0)\}$ $C_2$ $\{\exists\langle\varphi'_1\rangle, \langle\varphi'_2\rangle.\, \varphi'_1(l) \neq \varphi'_2(l)\}$. The postcondition is the negation of the postcondition for $C_1$ above, hence expressing that $C_2$ *violates* NI. Note that the precondition needs to be stronger than for $C_1$. Since the postcondition has to hold for *all* sets that satisfy the precondition, we have to require that the set of initial states includes two states that will definitely lead to different final values of $l$.

The only other Hoare logic that can be used to both prove and disprove $k$-safety hyperproperties is RHLE, since it supports $\forall^*\exists^*$-hyperproperties, which includes both hypersafety (that is, $\forall^*$) properties and their negation (that is, $\exists^*$-hyperproperties). However, RHLE does not support $\exists^*\forall^*$-hyperproperties, and thus cannot disprove $\forall^*\exists^*$-hyperproperties such as generalized non-interference, as we discuss next.

## 2.3 Beyond $k$-Safety

NI is widely used to express information flow security for deterministic programs, but is overly restrictive for non-deterministic programs. For example, the command $C_3 \triangleq (y := nonDet();\ l := h + y)$ is information flow secure. Since the secret $h$ is added to an unbounded non-deterministically chosen integer $y$, any secret $h$ can result in any[2] value for the public variable $l$ and, thus, we cannot learn anything certain about $h$ from observing the value of $l$. However, because of non-determinism, $C_3$ does not satisfy NI: Two executions with the same initial values for $l$ could get different values for $y$, and thus have different final values for $l$.

Information flow security for non-deterministic programs (such as $C_3$) is often formalized as *generalized non-interference* (GNI) [McCullough 1987; McLean 1996], a security notion weaker than NI. GNI allows two executions $\tau_1$ and $\tau_2$ with the same low inputs to have *different* low outputs, provided that there is a third execution $\tau$ with the same low inputs that has the same high inputs as $\tau_1$ and the same low outputs as $\tau_2$. That is, the difference in the low outputs between $\tau_1$ and $\tau_2$ cannot be attributed to their secret inputs.[3] The non-deterministic program $C_3$ satisfies GNI, which can be expressed via the hyper-triple[4] $\{low(l)\}$ $C_3$ $\{\forall\langle\varphi'_1\rangle, \langle\varphi'_2\rangle.\, \exists\langle\varphi'\rangle.\, \varphi'(h) = \varphi'_1(h) \land \varphi'(l) = \varphi'_2(l)\}$. The final states $\varphi'_1$ and $\varphi'_2$ correspond to the executions $\tau_1$ and $\tau_2$, respectively, and $\varphi'$ corresponds to execution $\tau$.

[2]This property holds for both unbounded and bounded arithmetic.

[3]GNI is often formulated without the requirement that $\tau_1$ and $\tau_2$ have the same low inputs, e.g., in Clarkson and Schneider [2008]. This alternative formulation can also be expressed in Hyper Hoare Logic, with the hyper-triple $\{\forall\langle\varphi\rangle.\, \varphi(l_{in}) = \varphi(l)\}$ $C_3$ $\{\forall\langle\varphi'_1\rangle, \langle\varphi'_2\rangle.\, \exists\langle\varphi'\rangle.\, \varphi'(h) = \varphi'_1(h) \land \varphi'(l_{in}) = \varphi'_2(l_{in}) \land \varphi'(l) = \varphi'_2(l)\}$. The precondition binds, in each state, the initial value of $l$ to the logical variable $l_{in}$, which enables the postcondition to refer to the initial value of $l$.

[4]We assume here for simplicity that $h$ is not modified by $C_3$.

As before, the expressivity of hyper-triples enables us not only to express that a program *satisfies* complex hyperproperties such as GNI, but also that a program *violates* them. For example, the program $C_4 \triangleq (y := \mathit{nonDet}();\ \mathbf{assume}\ y \leq 9;\ l := h + y)$, where the first two statements model a non-deterministic choice of $y$ smaller or equal to 9, leaks information: Observing for example $l = 20$ at the end of an execution, we can deduce that $h \geq 11$ (because $y \leq 9$). We can formally express that $C_4$ violates GNI using the following hyper-triple:[5]

$\{low(l) \land (\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(h) \neq \varphi_2(h))\}\ C_4\ \{\exists\langle\varphi_1'\rangle, \langle\varphi_2'\rangle.\ \forall\langle\varphi'\rangle.\ \varphi'(h) = \varphi_1'(h) \Rightarrow \varphi'(l) \neq \varphi_2'(l)\}$

The postcondition implies the negation of the postcondition we used previously to express GNI. As before, we had to strengthen the precondition to prove this violation.

GNI is a ∀∀∃-hyperproperty, whereas its negation is an ∃∃∀-hyperproperty. To the best of our knowledge, Hyper Hoare Logic is the only Hoare logic that can prove and disprove GNI. In fact, we will see in Sect. 3.5 that all hyperproperties over terminating program executions can be proven or disproven with Hyper Hoare Logic.

## 3 HYPER HOARE LOGIC

In this section, we present the programming language used in this paper (Sect. 3.1), formalize hyper-triples (Sect. 3.2), present the core rules of Hyper Hoare Logic (Sect. 3.3), prove soundness and completeness of the logic w.r.t. hyper-triples (Sect. 3.4), formally characterize the expressivity of hyper-triples (Sect. 3.5), and discuss additional rules for composing proofs (Sect. 3.6). All technical results presented in this section have been formalized in Isabelle/HOL.

### 3.1 Language and Semantics

We present Hyper Hoare Logic for the following imperative programming language:

Definition 1. ***Program states and programming language.*** *A* program state *(ranged over by $\sigma$) is a mapping from local variables (in the set PVars) to values (in the set PVals): The set of program states PStates is defined as the set of total functions from PVars to PVals: $PStates \triangleq PVars \rightarrow PVals$.*

*Program commands $C$ are defined by the following syntax, where $x$ ranges over variables in the set PVars, $e$ over expressions (modeled as total functions from PStates to PVals), and $b$ over predicates over states (total functions from PStates to Booleans):*

$$C ::= \mathbf{skip} \mid x := e \mid x := \mathit{nonDet}() \mid \mathbf{assume}\ b \mid C; C \mid C + C \mid C^*$$

The **skip**, assignment, and sequential composition commands are standard. The command **assume** $b$ acts like **skip** if $b$ holds and otherwise stops the execution. Instead of including *deterministic* if-statements and while loops, we consider a *non-deterministic* choice $C_1 + C_2$ and a *non-deterministic* iteration $C^*$, which are more expressive. Combined with the **assume** command, they can express deterministic if-statements and while loops as follows:

$$\mathbf{if}(b)\{C_1\}\mathbf{else}\{C_2\} \triangleq (\mathbf{assume}\ b; C_1) + (\mathbf{assume}\ \neg b; C_2)$$
$$\mathbf{while}\ (b)\ \{C\} \triangleq (\mathbf{assume}\ b; C)^*;\ \mathbf{assume}\ \neg b$$

Our language also includes a non-deterministic assignment $y := \mathit{nonDet}()$ (also called *havoc*), which allows us to model unbounded non-determinism. Together with **assume**, it can for instance model the generation of random numbers between bounds: $y := \mathit{randIntBounded}(a, b)$ can be modeled as $y := \mathit{nonDet}();\ \mathbf{assume}\ a \leq y \leq b$.

The big-step semantics of our language is standard, and formally defined in Fig. 2. The rule for $x := \mathit{nonDet}()$ allows $x$ to be updated with any value $v$. **assume** $b$ leaves the state unchanged if $b$ holds; otherwise, the semantics gets stuck to indicate that their is no execution in which $b$ does *not*

[5]Still assuming that $h$ is not modified.

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \rightarrow \sigma} \quad \frac{}{\langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto e(\sigma)]} \quad \frac{}{\langle x := nonDet(), \sigma \rangle \rightarrow \sigma[x \mapsto v]} \quad \frac{\langle C_1, \sigma \rangle \rightarrow \sigma' \quad \langle C_2, \sigma' \rangle \rightarrow \sigma''}{\langle C_1 ; C_2, \sigma \rangle \rightarrow \sigma''}$$

$$\frac{\langle C_1, \sigma \rangle \rightarrow \sigma'}{\langle C_1 + C_2, \sigma \rangle \rightarrow \sigma'} \quad \frac{\langle C_2, \sigma \rangle \rightarrow \sigma'}{\langle C_1 + C_2, \sigma \rangle \rightarrow \sigma'} \quad \frac{b(\sigma)}{\langle \textbf{assume } b, \sigma \rangle \rightarrow \sigma} \quad \frac{\langle C, \sigma \rangle \rightarrow \sigma' \quad \langle C^*, \sigma' \rangle \rightarrow \sigma''}{\langle C^*, \sigma \rangle \rightarrow \sigma''} \quad \frac{}{\langle C^*, \sigma \rangle \rightarrow \sigma}$$

Fig. 2. Big-step semantics. Since expressions are functions from states to values, $e(\sigma)$ denotes the evaluation of expression $e$ in state $\sigma$. $\sigma[x \mapsto v]$ is the state that yields $v$ for $x$ and the value in $\sigma$ for all other variables.

hold. The command $C_1 + C_2$ non-deterministically executes either $C_1$ or $C_2$. $C^*$ non-deterministically either performs another loop iteration or terminates.

Note that our language does not contain any command that could fail (in particular, expression evaluation is total, such that division-by-zero and other errors cannot occur). Runtime failures could easily be modeled by instrumenting the program with a special Boolean variable *err* that tracks whether a runtime error has occurred and skips the rest of the execution if this is the case.

## 3.2 Hyper-Triples, Formally

As explained in Sect. 2, the key idea behind Hyper Hoare Logic is to use *properties of sets of states* as pre- and postconditions, whereas traditional Hoare logics use properties of individual states (or of a given number $k$ of states in logics for hyperproperties). Considering arbitrary sets of states increases the expressivity of triples substantially; for instance, universal and existential quantification over these sets corresponds to over- and underapproximate reasoning, respectively. Moreover, combining both forms of quantification allows one to express advanced hyperproperties, such as generalized non-interference (see Sect. 2.3).

To allow the assertions of Hyper Hoare Logic to refer to logical variables (motivated in Sect. 2.2), we include them in our notion of state.

Definition 2. ***Extended states.*** *An* extended state *(ranged over by $\varphi$) is a pair of a* logical state *(a total mapping from logical variables to logical values) and a program state:*

$$ExtStates \triangleq (LVars \rightarrow LVals) \times PStates$$

*Given an extended state $\varphi$, we write $\varphi^L$ to refer to the logical state and $\varphi^P$ to refer to the program state, that is, $\varphi = (\varphi^L, \varphi^P)$.*

We use the same meta variables $(x, y, z)$ for program and logical variables. When it is clear from the context that $x \in PVars$ (resp. $x \in LVars$), we often write $\varphi(x)$ to denote $\varphi^P(x)$ (resp. $\varphi^L(x)$).

The assertions of Hyper Hoare Logic are predicates over sets of extended states:

Definition 3. ***Hyper-assertions.*** *A* hyper-assertion *(ranged over by $P$, $Q$, $R$) is a total function from $\mathbb{P}(ExtStates)$ to Booleans.*
*A hyper-assertion $P$* entails *a hyper-assertion $Q$, written $P \models Q$, iff all sets that satisfy $P$ also satisfy $Q$:*

$$(P \models Q) \triangleq (\forall S.\, P(S) \Rightarrow Q(S))$$

Following Incorrectness Logic and others, we formalize hyper-assertions as semantic properties, which allows us to focus on the key ideas of our logic. In Sect. 4, we will define a syntax for hyper-assertions, which will allow us to derive simpler rules than the ones presented in this section.

To formalize the meaning of hyper-triples, we need to relate them formally to the semantics of our programming language. Since hyper-triples are defined over extended states, we first define a semantic function *sem* that lifts the operational semantics to extended states; it yields the set of extended states that can be reached by executing a command $C$ from a set of extended states $S$:

$$\frac{}{\vdash \{P\}\ \mathit{skip}\ \{P\}}\ (\mathit{Skip}) \qquad \frac{\vdash \{P\}\ C_1\ \{R\} \quad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}\ (\mathit{Seq}) \qquad \frac{\vdash \{P\}\ C_1\ \{Q_1\} \quad \vdash \{P\}\ C_2\ \{Q_2\}}{\vdash \{P\}\ C_1 + C_2\ \{Q_1 \otimes Q_2\}}\ (\mathit{Choice})$$

$$\frac{P \models P' \quad Q' \models Q \quad \vdash \{P'\}\ C\ \{Q'\}}{\vdash \{P\}\ C\ \{Q\}}\ (\mathit{Cons}) \qquad \frac{}{\vdash \{\lambda S.\, P(\{\varphi \mid \varphi \in S \wedge b(\varphi^P)\})\}\ \textbf{assume}\ b\ \{P\}}\ (\mathit{Assume})$$

$$\frac{\forall x.\ (\vdash \{P(x)\}\ C\ \{Q(x)\})}{\vdash \{\exists x.\, P(x)\}\ C\ \{\exists x.\, Q(x)\}}\ (\mathit{Exist}) \qquad \frac{}{\vdash \{\lambda S.\, P(\{\varphi \mid \exists \alpha \in S.\, \varphi^L = \alpha^L \wedge \varphi^P = \alpha^P[x \mapsto e(\varphi^P)]\})\}\ x := e\ \{P\}}\ (\mathit{Assign})$$

$$\frac{\vdash \{I_n\}\ C\ \{I_{n+1}\}}{\vdash \{I_0\}\ C^*\ \{\bigotimes_{n \in \mathbb{N}} I_n\}}\ (\mathit{Iter}) \qquad \frac{}{\vdash \{\lambda S.\, P(\{\varphi \mid \exists \alpha \in S.\, \exists v.\, \varphi^L = \alpha^L \wedge \varphi^P = \alpha^P[x \mapsto v]\})\}\ x := \mathit{nonDet}()\ \{P\}}\ (\mathit{Havoc})$$

Fig. 3. Core rules of Hyper Hoare Logic. The meaning of the operators $\otimes$ and $\bigotimes_{n\in\mathbb{N}}$ are defined in Def. 6 and Def. 7, respectively.

Definition 4. ***Extended semantics.***

$$\mathit{sem}(C, S) \triangleq \{\varphi \mid \exists \sigma.\, (\varphi^L, \sigma) \in S \wedge \langle C, \sigma \rangle \to \varphi^P\}$$

The following lemma states several useful properties of the extended semantics.

Lemma 1. ***Properties of the extended semantics.***

(1) $\mathit{sem}(C, S_1 \cup S_2) = \mathit{sem}(C, S_1) \cup \mathit{sem}(C, S_2)$
(2) $S \subseteq S' \implies \mathit{sem}(C, S) \subseteq \mathit{sem}(C, S')$
(3) $\mathit{sem}(C, \bigcup_x f(x)) = \bigcup_x \mathit{sem}(C, f(x))$
(4) $\mathit{sem}(\textbf{skip}, S) = S$
(5) $\mathit{sem}(C_1; C_2, S) = \mathit{sem}(C_2, \mathit{sem}(C_1, S))$
(6) $\mathit{sem}(C_1 + C_2, S) = \mathit{sem}(C_1, S) \cup \mathit{sem}(C_2, S)$
(7) $\mathit{sem}(C^*, S) = \bigcup_{n\in\mathbb{N}} \mathit{sem}(C^n, S)$ *where* $C^n \triangleq \underbrace{C;\ \dots;\ C}_{n\ \mathit{times}}$

Using the extended semantics, we can now define the meaning of hyper-triples.

Definition 5. ***Hyper-triples.*** *Given two hyper-assertions $P$ and $Q$, and a command $C$, the* hyper-triple $\{P\}\ C\ \{Q\}$ *is* valid, *written* $\models \{P\}\ C\ \{Q\}$, *iff for any set $S$ of initial extended states that satisfies $P$, the set $\mathit{sem}(C, S)$ of extended states reachable by executing $C$ in some state from $S$ satisfies $Q$:*

$$\models \{P\}\ C\ \{Q\} \triangleq (\forall S.\, P(S) \Rightarrow Q(\mathit{sem}(C, S)))$$

This definition is similar to classical Hoare logic, where the initial and final states have been replaced by *sets* of extended states. As we have seen in Sect. 2, hyper-assertions over sets of states allow our hyper-triples to express properties of single executions (trace properties) and of multiple executions (hyperproperties), as well as to perform overapproximate reasoning (like e.g., Hoare Logic) and underapproximate reasoning (like e.g., Incorrectness Logic).

## 3.3 Core Rules

Fig. 3 shows the core rules of Hyper Hoare Logic. *Skip*, *Seq*, *Cons*, and *Exist* are analogous to traditional Hoare logic. *Assume*, *Assign*, and *Havoc* are straightforward given the semantics of these commands. All three rules work backward. In particular, the precondition of *Assume* applies the postcondition $P$ only to those states that satisfy the assumption $b$. By leaving the value $v$ unconstrained, *Havoc* considers as precondition the postcondition $P$ for all possible values for $x$.

The three rules *Assume*, *Assign*, and *Havoc* are optimized for expressivity; we will derive in Sect. 4 syntactic versions of these rules, which are less expressive, but easier to apply.

The rule *Choice* (for non-deterministic choice) is more involved. Most standard Hoare logics use the same assertion $Q$ as postcondition of all three triples. However, such a rule would not be sound in Hyper Hoare Logic. Consider for instance an application of this hypothetical *Choice* rule where both $P$ and $Q$ are defined as $\lambda S. |S| = 1$, expressing that there is a single pre- and post-state. If commands $C_1$ and $C_2$ are deterministic, the antecedents of the rule can be proved because a single pre-state leads to a single post-state. However, the non-deterministic choice will in general produce *two* post-states, such that the postcondition is violated.

To account for the effects of non-determinism on the sets of states described by hyper-assertions, we obtain the postcondition of the non-deterministic choice by combining the postconditions of its branches. As shown by Lemma 1(6), executing the non-deterministic choice $C_1 + C_2$ in the set of states $S$ amounts to executing $C_1$ in $S$ and $C_2$ in $S$, and taking the union of the two resulting sets of states. Thus, if $Q_1(sem(C_1, S))$ and $Q_2(sem(C_2, S))$ hold then the postcondition of $C_1 + C_2$ must characterize the union $sem(C_1, S) \cup sem(C_2, S)$ The postcondition of the rule *Choice*, $Q_1 \otimes Q_2$, achieves that:

Definition 6. *A set $S$ satisfies $Q_1 \otimes Q_2$ iff it can be split into two (potentially overlapping) sets $S_1$ and $S_2$ (the sets of post-states of the branches), such that $S_1$ satisfies $Q_1$ and $S_2$ satisfies $Q_2$:*

$$(Q_1 \otimes Q_2)(S) \triangleq (\exists S_1, S_2. S = S_1 \cup S_2 \land Q_1(S_1) \land Q_2(S_2))$$

The rule *Iter* for non-deterministic iterations generalizes our treatment of non-deterministic choice. It employs an indexed loop invariant $I$, which maps a natural number $n$ to a hyper-assertion $I_n$. $I_n$ characterizes the set of states reached after executing $n$ times the command $C$ in a set of initial states that satisfies $I_0$. Analogously to the rule *Choice*, the indexed invariant avoids using the same hyper-assertion for all non-deterministic choices. The precondition of the rule's conclusion and its premise prove (inductively) that the triple $\{I_0\}\ C^n\ \{I_n\}$ holds for all $n$. $I_n$ thus characterizes the set of reachable states after exactly $n$ iterations of the loop. Since our loop is non-deterministic (i.e., has no loop condition), the set of reachable states after the loop is the union of the sets of reachable states after each iteration. The postcondition of the conclusion captures this intuition, by using the generalized version of the $\otimes$ operator to an indexed family of hyper-assertions:

Definition 7. *A set $S$ satisfies $\bigotimes_{n \in \mathbb{N}} I_n$ iff it can be split into $\bigcup_i f(i) = f(0) \cup \ldots \cup f(i) \cup \ldots$, where $f(i)$ (the set of reachable states after exactly $i$ iterations) satisfies $I_i$ (for each $i \in \mathbb{N}$):*

$$(\bigotimes_{n \in \mathbb{N}} I_n)(S) \triangleq (\exists f. (S = \bigcup_{n \in \mathbb{N}} f(n)) \land (\forall n \in \mathbb{N}. I_n(f(n))))$$

Note that this rule makes Hyper Hoare Logic a partial correctness logic: it only considers an unbounded, but finite number $n$ of loop iterations. In App. E, we discuss an alternative rule for total correctness, which proves that all executions terminate. We also discuss a possible extension of Hyper Hoare Logic to prove non-termination, i.e., the existence of non-terminating executions.

## 3.4 Soundness and Completeness

We have proved in Isabelle/HOL that Hyper Hoare Logic is sound and complete. That is, every hyper-triple that can be derived in the logic is valid, and vice versa. Note that Fig. 3 contains only the *core rules* of Hyper Hoare Logic. These are sufficient to prove completeness; all rules presented later in this paper are only useful to make proofs more succinct and natural.

Theorem 1. ***Soundness.*** *Hyper Hoare Logic is sound:*

$$\text{If} \vdash \{P\}\ C\ \{Q\}\ \text{then} \models \{P\}\ C\ \{Q\}.$$

Theorem 2. ***Completeness.*** *Hyper Hoare Logic is complete:*

$$\text{If} \models \{P\}\ C\ \{Q\} \text{ then} \vdash \{P\}\ C\ \{Q\}.$$

Note that our completeness theorem is not concerned with the expressivity of the assertion language because we use *semantic* hyper-assertions (i.e., functions, see Def. 3). Similarly, by using semantic entailments in the rule *Cons*, we decouple the completeness of Hyper Hoare Logic from the completeness of the logic used to derive entailments.

Interestingly, the logic would *not* be complete without the core rule *Exist*, as we illustrate with the following simple example:

Example 1. *Let $\varphi_v$ be the state that maps $x$ to $v$ and all other variables to $0$. Let $P_v \triangleq (\lambda S.\, S = \{\varphi_v\})$. Clearly, the hyper-triples $\{P_0\}$ **skip** $\{P_0\}$, $\{P_2\}$ **skip** $\{P_2\}$, $\{P_0\}$ $x := x+1$ $\{P_1\}$, and $\{P_2\}$ $x := x+1$ $\{P_3\}$ are all valid. We would like to prove the hyper-triple $\{P_0 \vee P_2\}$ **skip** $+ (x := x+1)$ $\{\lambda S.\, S = \{\varphi_0, \varphi_1\} \vee S = \{\varphi_2, \varphi_3\}\}$. That is, either $P_0$ holds before, and then we have $S = \{\varphi_0, \varphi_1\}$ afterwards, or $P_2$ holds before, and then we have $S = \{\varphi_2, \varphi_3\}$ afterwards. However, using the rule Choice only, the most precise triple we can prove is*

$$\frac{\{P_0 \vee P_2\}\ \textbf{skip}\ \{P_0 \vee P_2\} \quad \{P_0 \vee P_2\}\ x := x+1\ \{P_1 \vee P_3\}}{\{P_0 \vee P_2\}\ \textbf{skip} + (x := x+1)\ \{(P_0 \vee P_2) \otimes (P_1 \vee P_3)\}}\ (\textit{Choice})$$

*The postcondition $(P_0 \vee P_2) \otimes (P_1 \vee P_3)$ is equivalent to $(P_0 \otimes P_1) \vee (P_0 \otimes P_3) \vee (P_2 \otimes P_1) \vee (P_2 \otimes P_3)$, i.e., $\lambda S.\, S = \{\varphi_0, \varphi_1\} \vee S = \{\varphi_0, \varphi_3\} \vee S = \{\varphi_2, \varphi_1\} \vee S = \{\varphi_2, \varphi_3\}$. We thus have two spurious disjuncts, $P_0 \otimes P_3$ (i.e., $S = \{\varphi_0, \varphi_3\}$) and $P_2 \otimes P_1$ (i.e., $S = \{\varphi_2, \varphi_1\}$).*

This example shows that the rule *Choice* on its own is not precise enough for the logic to be complete; we need at least a *disjunction* rule to distinguish the two cases $A$ and $B$. In general, however, there might be an infinite number of cases to consider, which is why we need the rule *Exist*. The premise of this rule allows us to *fix* a set of states $S$ that satisfies some precondition $P$, and to prove the most precise postcondition for the precondition $\lambda S'.\, S = S'$; combining these precise postconditions with an existential quantifier in the conclusion of the rule allows us to obtain the most precise postcondition for the precondition $P$.

## 3.5 Expressivity of Hyper-Triples

In the previous subsection, we have shown that Hyper Hoare Logic is sound and complete to establish the validity of hyper-triples, and, thus, Hyper Hoare Logic is as expressive as hyper-triples. We now show that hyper-triples are expressive enough to capture arbitrary hyperproperties over finite program executions. A *hyperproperty* [Clarkson and Schneider 2008] is traditionally defined as a property of sets of *traces* of a system, that is, of sequences of system states. Since Hoare logics typically consider only the initial and final state of a program execution, we use a slightly adapted definition here:

Definition 8. ***Program hyperproperties.*** *A* program hyperproperty *is a set of sets of pairs of program states, i.e., an element of $\mathbb{P}(\mathbb{P}(\mathit{PStates} \times \mathit{PStates}))$.*

*A command $C$ satisfies the program hyperproperty $\mathcal{H}$ iff the set of all pairs of pre- and post-states of $C$ is an element of $\mathcal{H}$: $\{(\sigma, \sigma') \mid \langle C, \sigma \rangle \to \sigma'\} \in \mathcal{H}$.*

Equivalently, a program hyperproperty can be thought of as a predicate over $\mathbb{P}(\mathit{PStates} \times \mathit{PStates})$. Note that this definition subsumes properties of single executions (trace properties), such as functional correctness properties.

In contrast to traditional hyperproperties, our program hyperproperties describe only the *finite* executions of a program, that is, those that reach a final state. An extension of Hyper Hoare Logic to infinite executions might be possible by defining hyper-assertions over sets of traces rather than sets of states; we leave this as future work. In the rest of this paper, when the context is clear, we use *hyperproperties* to refer to *program hyperproperties*.

Any program hyperproperty can be expressed as a hyper-triple in Hyper Hoare Logic:[6]

THEOREM 3. ***Expressing hyperproperties as hyper-triples.*** *Let $\mathcal{H}$ be a program hyperproperty. Assume that the cardinality of LVars is at least the cardinality of PVars, and that the cardinality of LVals is at least the cardinality of PVals.*

*Then there exist hyper-assertions $P$ and $Q$ such that, for any command $C$, $C \in \mathcal{H}$ iff $\models \{P\}\ C\ \{Q\}$.*

PROOF SKETCH. We define the precondition $P$ such that the initial set of states $S$ contains all program states, and the values of all program variables in these states are recorded in logical variables (which is possible due to the cardinality assumptions). Since the logical variables are not affected by the execution of $C$, they allow $Q$ to refer to the initial values of any program variable, in addition to their values in the final state. Consequently, $Q$ can describe all possible pairs of pre- and post-states. We simply define $Q$ to be true iff the set of these pairs is contained in $\mathcal{H}$. □

Combined with our completeness result (Thm. 2), this theorem implies that, if a command $C$ satisfies a hyperproperty $\mathcal{H}$ then there exists a proof of it in Hyper Hoare Logic. More surprisingly, our logic also allows us to *disprove* any hyperproperty: If $C$ does *not* satisfy $\mathcal{H}$ then $C$ satisfies the *complement* of $\mathcal{H}$, which is also a hyperproperty, and thus can also be proved. Consequently, Hyper Hoare Logic can prove or disprove any *program hyperproperty* as defined in Def. 8.

Since hyper-triples can exactly express hyperproperties (Thm. 3 and footnote 6), the ability to disprove any hyperproperty implies that Hyper Hoare Logic can also disprove any *hyper-triple*. More precisely, one can *always* use Hyper Hoare Logic to prove that some hyper-triple $\{P\}\ C\ \{Q\}$ is *invalid*, by proving the validity of another hyper-triple $\{P'\}\ C\ \{\neg Q\}$ (where $P'$ is a satisfiable hyper-assertion that entails $P$). Conversely, the validity of such a hyper-triple $\{P'\}\ C\ \{\neg Q\}$ implies that all hyper-triples $\{P\}\ C\ \{Q\}$ (with $P$ weaker than $P'$) are *invalid*. The following theorem precisely expresses this observation:

THEOREM 4. ***Disproving hyper-triples.*** *Given $P$, $C$, and $Q$, the following two propositions are equivalent:*

(1) *$\models \{P\}\ C\ \{Q\}$ does not hold.*
(2) *There exists a hyper-assertion $P'$ that is satisfiable, entails $P$, and $\models \{P'\}\ C\ \{\neg Q\}$.*

We need to strengthen $P$ to $P'$ in point (2), because there might be some sets $S, S'$ that both satisfy $P$, such that $Q(sem(C, S))$ holds, but $Q(sem(C, S'))$ does not. This was the case for our examples in Sect. 2.2 and Sect. 2.3; for instance, one of the preconditions there was strengthened to include $\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(h) \neq \varphi_2(h)$.

Thm. 4 is another illustration of the expressivity of Hyper Hoare Logic. The corresponding result does *not* hold in traditional Hoare logics. For example, the classical Hoare triple $\{\top\}\ x := nonDet()\ \{x \geq 5\}$ does not hold, but there is no satisfiable $P$ such that $\{P\}\ x := nonDet()\ \{\neg(x \geq 5)\}$ holds. In contrast, Hyper Hoare Logic can disprove the classical Hoare triple by proving the hyper-triple $\{\top\}\ x := nonDet()\ \{\neg(\forall\langle\varphi\rangle.\ \varphi(x) \geq 5)\}$.

The correspondence between hyper-triples and program hyperproperties (Thm. 3 and footnote 6), together with our completeness result (Thm. 2) precisely characterizes the expressivity of Hyper Hoare Logic. In App. C, we also show how to express the judgments of existing over- and underapproximating Hoare logics as hyper-triples, in systematic ways.

[6]We also proved the converse: every hyper-triple describes a program hyperproperty. That is, hyper-triples capture exactly the hyperproperties over finite executions.

## 3.6 Compositionality

The core rules of Hyper Hoare Logic allow one to prove any valid hyper-triple, but not necessarily *compositionally*. As an example, consider the sequential composition of a command $C_1$ that satisfies *generalized* non-interference (GNI) with a command $C_2$ that satisfies non-interference (NI). We would like to prove that $C_1$; $C_2$ satisfies GNI (the weaker property). As discussed in Sect. 2.3, a possible postcondition for $C_1$ is $GNI_l^h \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \exists\langle\varphi\rangle.\, \varphi_1(h) = \varphi(h) \wedge \varphi(l) = \varphi_2(l))$, while a possible precondition for $C_2$ is $low(l) \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(l) = \varphi_2(l))$. However, the corresponding hyper-triples for $C_1$ and $C_2$ cannot be composed using the core rules. In particular, rule *Seq* cannot be applied (even in combination with *Cons*), since the postcondition of $C_1$ does not imply the precondition of $C_2$. Note that this observation does *not* contradict completeness: By Thm. 2, it is possible to prove *more precise* triples for $C_1$ and $C_2$, such that the postcondition of $C_1$ matches the precondition of $C_2$. However, to enable modular reasoning, our goal is to construct the proof by composing the given triples for the individual commands rather than deriving new ones.

We have thus proven a number of useful *compositionality rules* for hyper-triples, which are presented in App. D. These rules are *admissible* in Hyper Hoare Logic, in the sense that they do not modify the set of valid hyper-triples that can be proved. Rather, they enable flexible compositions of hyper-triples, as we illustrate in App. D.2 on two challenging examples, including the composition of GNI with NI mentioned above.

## 4 SYNTACTIC RULES

The core rules presented in Sect. 3 are optimized for expressivity: They are sufficient to prove *any* valid hyper-triple (Thm. 2), but not necessarily in the simplest way. In particular, the rules for atomic statements *Assume*, *Assign*, and *Havoc* require a set comprehension in the precondition, which is necessary when dealing with arbitrary semantic hyper-assertions. However, by imposing syntactic restrictions on hyper-assertions, we can derive simpler rules, as we show in this section. In Sect. 4.1, we define a syntax for hyper-assertions, in which the set of states occurs only as range of universal and existential quantifiers. As we have seen in Sect. 2 and show in App. C, this syntax is sufficient to capture many useful hyperproperties. Moreover, it allows us to derive simple rules for assignments (Sect. 4.2) and assume statements (Sect. 4.3). All rules presented in this section have been proven sound in Isabelle/HOL.

### 4.1 Syntactic Hyper-Assertions

We define a restricted class of syntactic hyper-assertions, which can interact with the set of states only through universal and existential quantification over its states:

Definition 9. ***Syntactic hyper-expressions and hyper-assertions.***
Hyper-expressions $e$ *are defined by the following syntax, where* $\varphi$ *ranges over states,* $x$ *over (program or logical) variables,* $y$ *over quantified variables,* $c$ *over literals,* $\oplus$ *over binary operators (such as* $+, -, *$ *for integers,* $++$ *for lists, etc.), and* $f$ *denotes functions from values to values (such as* len *for lists):*

$$e ::= c \mid y \mid \varphi^P(x) \mid \varphi^L(x) \mid e \oplus e \mid f(e)$$

Syntactic hyper-assertions $A$ *are defined by the following syntax, where* $e$ *ranges over hyper-expressions,* $b$ *over boolean literals, and* $\succeq$ *over binary operators (such as* $=, <, >, \leq, \geq, \ldots$*):*

$$A ::= b \mid e \succeq e \mid A \vee A \mid A \wedge A \mid \forall y.\, A \mid \exists y.\, A \mid \forall\langle\varphi\rangle.\, A \mid \exists\langle\varphi\rangle.\, A$$

Note that *hyper-expressions* are different from *program* expressions, since the latter can only refer to program variables of a *single* implicit state (e.g., $x = y + z$), while the former can explicitly refer to different states (e.g., $\varphi(x) = \varphi'(x)$). Negation $\neg A$ is defined recursively in the standard

$$\frac{}{\vdash \{\mathcal{A}_x^e\,[P]\}\ x := e\ \{P\}}\,(AssignS) \qquad \frac{}{\vdash \{\mathcal{H}_x\,[P]\}\ x := nonDet()\ \{P\}}\,(HavocS) \qquad \frac{}{\vdash \{\Pi_b\,[P]\}\ \textbf{assume}\ b\ \{P\}}\,(AssumeS)$$

Fig. 4. Some syntactic rules of Hyper Hoare Logic. The syntactic transformations $\mathcal{A}_x^e\,[A]$ and $\mathcal{H}_x\,[A]$ are defined in Def. 10, and the syntactic transformation $\Pi_b\,[\_]$ is defined in Def. 11.

way. We also define $(A \Rightarrow B) \triangleq (\neg A \vee B)$, $emp \triangleq (\forall\langle\varphi\rangle.\,\bot)$, and $\Box p \triangleq (\forall\langle\varphi\rangle.\,p(\varphi))$, where $p$ is a *state*[7] expression. The evaluation of hyper-expressions and satisfiability of hyper-assertions are formally defined in Def. 12 (App. A).

## 4.2 Syntactic Rules for Deterministic and Non-Deterministic Assignments

In classical Hoare logic, we obtain the precondition of the rule for the assignment $x := e$ by substituting $x$ by $e$ in the postcondition. The Hyper Hoare Logic syntactic rule for assignments *AssignS* (Fig. 4) generalizes this idea by repeatedly applying this substitution for *every quantified state*. This syntactic transformation, written $\mathcal{A}_x^e\,[\_]$ is defined below. As an example, for the assignment $x := y + z$ and postcondition $\exists\langle\varphi\rangle.\,\forall\langle\varphi'\rangle.\,\varphi(x) \leq \varphi'(x)$, we obtain the precondition $\mathcal{A}_x^{y+z}\,[\exists\langle\varphi\rangle.\,\forall\langle\varphi'\rangle.\,\varphi(x) \leq \varphi'(x)] = (\exists\langle\varphi\rangle.\,\forall\langle\varphi'\rangle.\,\varphi(y) + \varphi(z) \leq \varphi'(y) + \varphi'(z))$.

Similarly, our syntactic rule for non-deterministic assignments *HavocS* substitutes every occurrence of $\varphi(x)$, for every quantified state $\varphi$, by a fresh quantified variable $v$. This variable is universally quantified for universally-quantified states, capturing the intuition that we must consider all possible assigned values. In contrast, $v$ is existentially quantified for existentially-quantified states, because it is sufficient to find one suitable behavior of the non-deterministic assignment. As an example, for the non-deterministic assignment $x := nonDet()$ and the aforementioned postcondition, we obtain the precondition $\mathcal{H}_x\,[\exists\langle\varphi\rangle.\,\forall\langle\varphi'\rangle.\,\varphi(x) \leq \varphi'(x)] = (\exists\langle\varphi\rangle.\,\exists v.\,\forall\langle\varphi'\rangle.\,\forall v'.\,v \leq v')$.

Definition 10. ***Syntactic transformations for assignments.***
*$\mathcal{A}_x^e\,[A]$ yields the hyper-assertion $A$, where $\varphi(x)$ is syntactically substituted by $e(\varphi)$, for all (existentially or universally) quantified states $\varphi$. The two main cases are:*

$$\mathcal{A}_x^e\,[\forall\langle\varphi\rangle.\,A] \triangleq \big(\forall\langle\varphi\rangle.\,\mathcal{A}_x^e\,[A[e(\varphi)/\varphi(x)]]\big) \qquad \mathcal{A}_x^e\,[\exists\langle\varphi\rangle.\,A] \triangleq \big(\exists\langle\varphi\rangle.\,\mathcal{A}_x^e\,[A[e(\varphi)/\varphi(x)]]\big)$$

*where $A[y/x]$ refers to the standard syntactic substitution of $x$ by $y$. Other cases apply $\mathcal{A}_x^e$ recursively (e.g., $\mathcal{A}_x^e\,[A \wedge B] \triangleq \mathcal{A}_x^e\,[A] \wedge \mathcal{A}_x^e\,[B]$). The full definition is in App. A.*

*$\mathcal{H}_x\,[A]$ yields the hyper-assertion $A$ where $\varphi(x)$ is syntactically substituted by a fresh quantified variable $v$, universally (resp. existentially) quantified for universally (resp. existentially) quantified states. The two main cases are:*

$$\mathcal{H}_x\,[\forall\langle\varphi\rangle.\,A] \triangleq (\forall\langle\varphi\rangle.\,\forall v.\,\mathcal{H}_x\,[A[v/\varphi(x)]]) \qquad \mathcal{H}_x\,[\exists\langle\varphi\rangle.\,A] \triangleq (\exists\langle\varphi\rangle.\,\exists v.\,\mathcal{H}_x\,[A[v/\varphi(x)]])$$

*where $v$ is fresh. Other cases apply $\mathcal{H}_x$ recursively. The full definition is in App. A.*

## 4.3 Syntactic Rules for Assume Statements

Intuitively, **assume** $b$ provides additional information when proving properties *for all* states, but imposes an additional requirement when proving *the existence* of a state. This intuition is captured by rule *AssumeS* shown in Fig. 4. The syntactic transformation $\Pi_b$ adds the state expression $b$ as an assumption for universally-quantified states, and as a proof obligation for

[7] *State* expressions refer to a single (implicit) state. In contrast to program expressions, they may additionally refer to logical variables and use quantifiers over values.

$$\{\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(h) \neq \varphi_2(h)\}$$

$$\{\exists\langle\varphi_1\rangle.\, (\exists\langle\varphi_2\rangle.\, (\forall\langle\varphi\rangle.\, \forall v.\, v \leq 9 \Rightarrow (\varphi(h) = \varphi_1(h) \Rightarrow \varphi_2(h) + 9 > \varphi(h) + v)))\} \qquad \text{(Cons)}$$

$$\{\exists\langle\varphi_1\rangle.\, \exists v_1.\, v_1 \leq 9 \wedge (\exists\langle\varphi_2\rangle.\, \exists v_2.\, v_2 \leq 9 \wedge (\forall\langle\varphi\rangle.\, \forall v.\, v \leq 9 \Rightarrow ((\varphi(h) \neq \varphi_1(h)) \vee (\varphi(h) + v \neq \varphi_2(h) + v_2))))\} \qquad \text{(Cons)}$$

$y \coloneqq nonDet()$;

$$\{\exists\langle\varphi_1\rangle.\, \varphi_1(y) \leq 9 \wedge (\exists\langle\varphi_2\rangle.\, \varphi_2(y) \leq 9 \wedge (\forall\langle\varphi\rangle.\, \varphi(y) \leq 9 \Rightarrow (\varphi(h) \neq \varphi_1(h) \vee \varphi(h) + \varphi(y) \neq \varphi_2(h) + \varphi_2(y))))\} \qquad \text{(HavocS)}$$

**assume** $y \leq 9$;

$$\{\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \forall\langle\varphi\rangle.\, \varphi(h) \neq \varphi_1(h) \vee \varphi(h) + \varphi(y) \neq \varphi_2(h) + \varphi_2(y)\} \qquad \text{(AssumeS)}$$

$l \coloneqq h + y$

$$\{\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \forall\langle\varphi\rangle.\, \varphi(h) \neq \varphi_1(h) \vee \varphi(l) \neq \varphi_2(l)\} \qquad \text{(AssignS)}$$

Fig. 5. Proof outline showing that the program *violates* generalized non-interference. The rules used at each step of the derivation are shown on the right (the use of rule *Seq* is implicit).

existentially-quantified states. As an example, for the statement **assume** $x \geq 0$ and the postcondition $\forall\langle\varphi\rangle.\, \exists\langle\varphi'\rangle.\, \varphi(x) \leq \varphi'(x)$, we obtain the precondition $\Pi_{x\geq 0}\left[\forall\langle\varphi\rangle.\, \exists\langle\varphi'\rangle.\, \varphi(x) \leq \varphi'(x)\right] = (\forall\langle\varphi\rangle.\, \varphi(x) \geq 0 \Rightarrow (\exists\langle\varphi'\rangle.\, \varphi'(x) \geq 0 \wedge \varphi(x) \leq \varphi'(x)))$.

Definition 11. ***Syntactic transformation for assume statements.*** *The two main cases of $\Pi_p$ are*

$$\Pi_p\left[\forall\langle\varphi\rangle.\, A\right] \triangleq \left(\forall\langle\varphi\rangle.\, p(\varphi) \Rightarrow \Pi_p\left[A\right]\right) \qquad \Pi_p\left[\exists\langle\varphi\rangle.\, A\right] \triangleq \left(\exists\langle\varphi\rangle.\, p(\varphi) \wedge \Pi_p\left[A\right]\right)$$

*Other cases apply $\Pi_p$ recursively. The full definition is in App. A.*

*Example.* We now illustrate the use of our three syntactic rules for atomic statements in Fig. 5, to prove that the program $C_4 \triangleq (y \coloneqq nonDet()$; **assume** $y \leq 9$; $l \coloneqq h + y)$ from Sect. 2.2 violates GNI. This program leaks information about the secret $h$ through its public output $l$ because the pad it uses (variable $y$) is upper bounded. From the output $l$, we can derive a lower bound for the secret value of $h$, namely $h \geq l - 9$.

To see why $C_4$ violates GNI, consider two executions with different secret values for $h$, and where the execution for the larger secret value sets $y$ to exactly 9. This execution will produce a larger public output $l$ (since the other execution adds at most 9 to its smaller secret). Hence, these executions can be *distinguished* by their public outputs.

Our proof outline in Fig. 5 captures this intuitive reasoning in a natural way. We start with the postcondition that corresponds to the negation of GNI, and work our way backward, by successively applying our syntactic rules *AssignS*, *AssumeS*, and *HavocS*. We conclude using the rule *Cons*: Since the precondition implies the existence of two states with different values for $h$, we first instantiate $\varphi_1$ and $\varphi_2$ such that $\varphi_1$ and $\varphi_2$ are both members of the initial set of states, and $\varphi_2(h) > \varphi_1(h)$.[8] We then instantiate $v_2 = 9$, such that, for any $v \leq 9$, $\varphi_2(h) + v_2 > \varphi(h) + v$, which concludes the proof.

## 5 PROOF PRINCIPLES FOR LOOPS

To reason about standard while loops, we can derive from the core rule *Iter* in Fig. 3 the rule *WhileDesugared*, shown in Fig. 6 (recall that **while** $(b)$ $\{C\} \triangleq ((\textbf{assume } b;\ C)^*;\ \textbf{assume } \neg b))$. While this derived rule is expressive, it has two main drawbacks for usability: (1) Because of the use of the infinitary $\bigotimes_{n\in\mathbb{N}}$, it requires non-trivial *semantic* reasoning (via the consequence rule),

[8]Note that the quantified states $\varphi_1$, $\varphi_2$ and $\varphi$ from different hyper-assertions can be unrelated. That is, the witnesses for $\varphi_1$ and $\varphi_2$ in the first hyper-assertion $[\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\, \varphi_1(h) \neq \varphi_2(h)]$ are not necessarily the same as the ones in the second hyper-assertion $[\exists\langle\varphi_1\rangle.\, \exists\langle\varphi_2\rangle.\, \varphi_2(h) > \varphi_1(h)]$, which is why the entailment holds.

$$\frac{\vdash \{I_n\}\ \textbf{assume}\ b; C\ \{I_{n+1}\} \qquad \vdash \{\bigotimes_{n \in \mathbb{N}} I_n\}\ \textbf{assume}\ \neg b\ \{Q\}}{\vdash \{I_0\}\ \textbf{while}\ (b)\ \{C\}\ \{Q\}}\ (\mathit{WhileDesugared})$$

$$\frac{I \models \mathit{low}(b) \qquad \vdash \{I \land \Box b\}\ C\ \{I\}}{\vdash \{I\}\ \textbf{while}\ (b)\ \{C\}\ \{(I \lor \mathit{emp}) \land \Box(\neg b)\}}\ (\mathit{WhileSync}) \qquad \frac{P \models \mathit{low}(b) \qquad \vdash \{P \land \Box b\}\ C_1\ \{Q\} \qquad \vdash \{P \land \Box(\neg b)\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{Q\}}\ (\mathit{IfSync})$$

$$\frac{\vdash \{I\}\ \textbf{if}\ (b)\ \{C\}\ \{I\} \qquad \vdash \{I\}\ \textbf{assume}\ \neg b\ \{Q\} \qquad \text{no } \forall\langle\_\rangle \text{ after any } \exists \text{ in } Q}{\vdash \{I\}\ \textbf{while}\ (b)\ \{C\}\ \{Q\}}\ (\mathit{While}\text{–}\forall^*\exists^*)$$

$$\frac{\forall v. \vdash \{\exists\langle\varphi\rangle. P_\varphi \land b(\varphi) \land v = e(\varphi)\}\ \textbf{if}\ (b)\ \{C\}\ \{\exists\langle\varphi\rangle. P_\varphi \land e(\varphi) \prec v\} \qquad \forall\varphi. \vdash \{P_\varphi\}\ \textbf{while}\ (b)\ \{C\}\ \{Q_\varphi\} \qquad \prec \text{wf}}{\vdash \{\exists\langle\varphi\rangle. P_\varphi\}\ \textbf{while}\ (b)\ \{C\}\ \{\exists\langle\varphi\rangle. Q_\varphi\}}\ (\mathit{While}\text{–}\exists)$$

Fig. 6. Hyper Hoare Logic rules for while loops (and branching). Recall that $\mathit{low}(b) \triangleq (\forall\langle\varphi\rangle, \langle\varphi'\rangle. b(\varphi) = b(\varphi'))$ and $\Box(b) \triangleq (\forall\langle\varphi\rangle. b(\varphi))$. In the rule *WhileSync*, $\prec$ must be *well-founded* (wf).

and (2) the invariant $I_n$ relates only the executions that perform *at least* $n$ iterations, but ignores executions that perform fewer.

To illustrate problem (2), imagine that we want to prove that the hyper-assertion $\mathit{low}(l) \triangleq (\forall\langle\varphi\rangle. \forall\langle\varphi'\rangle. \varphi(l) = \varphi'(l))$ holds after a while loop. A natural choice for our loop invariant $I_n$ would be $I_n \triangleq \mathit{low}(l)$ (independent of $n$). However, this invariant does *not* entail our desired postcondition $\mathit{low}(l)$. Indeed, $\bigotimes_{n \in \mathbb{N}} \mathit{low}(l)$ holds for a set of states iff it is a *union of* sets of states that all *individually* satisfy $\mathit{low}(l)$. This property holds trivially in our example (simply choose one set per possible value of $l$) and, in particular, does not express that the entire set of states after the loop satisfies $\mathit{low}(l)$. Note that this does not contradict completeness (Thm. 2), but simply means that a stronger invariant $I_n$ is needed.

In this section, we thus present three more convenient loop rules, shown in Fig. 6, which capture powerful reasoning principles, and overcome those limitations: The rule *WhileSync* (Sect. 5.1) is the easiest to use, and can be applied whenever all executions of the loop have the same control flow. Two additional rules for while loops can be applied whenever the control flow differs. The rule *While*-$\forall^*\exists^*$ (Sect. 5.2) supports $\forall^*\exists^*$ postconditions, while the rule *While*-$\exists$ (Sect. 5.3) handles postconditions with a top-level existential quantifier. In our experience, these loop rules cover all practical hyper-assertions that can be expressed in our syntax. We are not aware of any practical hyperproperties that require multiple quantifier alternations.

## 5.1 Synchronized Control Flow

Standard loop invariants are sound in relational logics if all executions exit the loop *simultaneously*. In our logic, this synchronized control flow can be enforced by requiring that the loop guard $b$ has the same value in all states (1) before the loop and (2) after every loop iteration, as shown by the rule *WhileSync* shown in Fig. 6. After the loop, we get to assume $(I \lor \mathit{emp}) \land \Box(\neg b)$. That is, the loop guard $b$ is false in all executions, and the invariant $I$ holds, or the set of states is empty. The *emp* disjunct corresponds to the case where the loop does not terminate (i.e., *no* execution terminates). Going back to our motivating example, the natural invariant $I \triangleq \mathit{low}(l)$ with the rule *WhileSync* is now sufficient for our example, since we get the postcondition $(\mathit{low}(l) \lor \mathit{emp}) \land \Box(\neg b)$, which implies our desired (universally-quantified) postcondition $\mathit{low}(l)$. In the case where the desired postcondition quantifies existentially over states at the top-level, it is necessary to prove that the loop terminates. We show the corresponding rules in App. E.

We also provide a rule for if statements with synchronized control flow (rule *IfSync* in Fig. 6), which can be applied when all executions take the same branch. This rule is simpler to apply than the core rule *Choice*, since it avoids the $\otimes$ operator, which usually requires semantic reasoning.

```
$\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ len(\varphi_1(h)) = len(\varphi_2(h))\}$
$\{\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ 0 = 0 \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land 0 = 0)\}$    (Cons)
$s := 0$
$l := []$
$i := 0$
$\{\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))\}$    (AssignS)
while $(i < len(h))$ {
  $\{(\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))) \land \Box(i < len(h))\}$
  $\{\forall\langle\varphi_1\rangle.\forall v_1.\forall\langle\varphi_2\rangle.\forall v_2.\ \varphi_1(i) + 1 = \varphi_2(i) + 1 \land len(\varphi_1(h)) = len(\varphi_2(h)) \land$
  $(\exists\langle\varphi\rangle.\ \exists v.\ \varphi(h) = \varphi_1(h) \land \varphi(l) \mathbin{++} [(\varphi(s) + \varphi(h)[\varphi(i)]) \oplus v] = \varphi_2(l) \mathbin{++} [(\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2])\}$    (Cons)
  $s := s + h[i]$;
  $k := nonDet()$;
  $l := l \mathbin{++} [s \oplus k]$;
  $i := i + 1$;
  $\{\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))\}$    (HavocS, AssignS)
}
$\{((\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))) \lor emp) \land \Box(i \geq len(h))\}$    (WhileSync)
$\{\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l)\}$    (Cons)
```

Fig. 7. A proof that the program in black satisfies generalized non-interference (where the elements of list $h$ are secret, but its length is public), using the rule *WhileSync*. [] represents the empty list, ++ represents list concatenation, $h[i]$ represents the i-th element of list $h$, and $\oplus$ represents the XOR operator.

*Example.* The program in Fig. 7 takes as input a list $h$ of secret values (but whose length is public), computes its prefix sum $[h[0], h[0] + h[1], \dots]$, and encrypts the result by performing a one-time pad on each element of this prefix sum, resulting in the output $[h[0] \oplus k_0, (h[0] + h[1]) \oplus k_1, \dots]$. The keys $k_0, k_1, \dots$ are chosen non-deterministically at each iteration, via the variable $k$.[9]

Our goal is to prove that the encrypted output $l$ does not leak information about the secret elements of $h$, provided that the attacker does not have any information about the non-deterministically chosen keys. We achieve this by formally proving that this program satisfies GNI. Since the length of the list $h$ is public, we start with the precondition $\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ len(\varphi_1(h)) = len(\varphi_2(h))$. This implies that all our executions will perform the same number of loop iterations. Thus, we use the rule *WhileSync*, with the natural loop invariant $I \triangleq (\forall\langle\varphi_1\rangle.\forall\langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l)))$. The last conjunct corresponds to the postcondition we want to prove, while the former entails $low(i < len(h))$, as required by the rule *WhileSync*.

The proof of the loop body starts at the end with the loop invariant $I$, and works backward, using the syntactic rules *HavocS* and *AssignS*. From $I \land \Box(i < len(h))$, we have to prove that there exists a value $v$ such that $\varphi(l) \mathbin{++} [(\varphi(s) + \varphi(h)[\varphi(i)]) \oplus v] = \varphi_2(l) \mathbin{++} [(\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2]$. Since $\varphi(l) = \varphi_2(l)$, this boils down to $(\varphi(s) + \varphi(h)[\varphi(i)]) \oplus v = (\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2$, which we achieve by choosing $v \triangleq (\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2 \oplus (\varphi(s) + \varphi(h)[\varphi(i)])$.

## 5.2 $\forall^*\exists^*$-Hyperproperties

Let us now turn to the more general case, where different executions might exit the loop at different iterations. As explained at the start of this section, the main usability issue of the rule *WhileDesugared* is the precondition $\bigotimes_{n \in \mathbb{N}} I_n$ in the second premise, which requires non-trivial semantic reasoning. The $\bigotimes_{n \in \mathbb{N}}$ operator is required, because $I_n$ ignores executions that exited the

[9]In practice, the keys used in this program should be stored somewhere, so that one is later able to decrypt the output.

loop earlier; it relates only the executions that have performed *at least n* iterations. In particular, it would be unsound to replace the precondition $\bigotimes_{n\in\mathbb{N}} I_n$ by $\exists n.\, I_n$.

The rule *While*-$\forall^*\exists^*$ in Fig. 6 solves this problem for the general case of $\forall^*\exists^*$ postconditions. The key insight is to reason about the successive *unrollings* of the while loop: the rule requires to prove an invariant $I$ for the conditional statement **if** $(b)$ $\{C\}$, as opposed to **assume** $b$; $C$ in the rule *WhileDesugared*. This allows the invariant $I$ to refer to *all* executions, i.e., executions that are still running the loop (which will execute $C$), and executions that have already exited the loop (which will not execute $C$).

*Example.* The program $C_{fib}$ in Fig. 8 takes as input an integer $n \geq 0$ and computes the $n$-th Fibonacci number (in variable $a$). We want to prove that $C_{fib}$ is monotonic, i.e., that the $n$-th Fibonacci number is greater than or equal to the $m$-th Fibonacci number whenever $n \geq m$, without making explicit what $C_{fib}$ computes. Formally, we want to prove the hyper-triple $\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\, \varphi_1(t){=}1 \land \varphi_2(t){=}2 \Rightarrow \varphi_1(n) \geq \varphi_2(n)\}$ $C_{fib}$ $\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\, \varphi_1(t){=}1 \land \varphi_2(t){=}2 \Rightarrow \varphi_1(a) \geq \varphi_2(a)\}$, where $t$ is a logical variable used to track the execution (as explained in Sect. 2.2). Intuitively, this program is monotonic because both executions will perform at least $\varphi_2(n)$ iterations, during which they will have the same values for $a$ and $b$. The first execution will then perform $\varphi_1(n) - \varphi_2(n)$ additional iterations, during which $a$ and $b$ will increase, thus resulting in larger values for $a$ and $b$.

We cannot use the rule *WhileSync* to make this intuitive argument formal, since both executions might perform a different number of iterations. Moreover, we cannot express this intuitive argument with the rule *WhileDesugared* either, since the invariant $I_k$ only relates executions that perform *at least k iterations*, as explained earlier: After the first $\varphi_2(n)$ iterations, the loop invariant $I_k$ cannot refer to the values of $a$ and $b$ in the second execution, since this execution has already exited the loop.

However, we can use the rule *While*-$\forall^*\exists^*$ to prove that $C_{fib}$ is monotonic, with the intuitive loop invariant $I \triangleq (\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\, \varphi_1(t){=}1 \land \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \land \varphi_1(a) \geq \varphi_2(a) \land \varphi_1(b) \geq \varphi_2(b)) \land \Box(b \geq a \geq 0))$. The first part captures the relation between the two executions: $a$ and $b$ are larger in the first execution than in the second one, and the first execution does at least as many iterations as the second one. The second part $\Box(b \geq a \geq 0)$ is needed to prove that the additional iterations lead to larger values for $a$ and $b$. The proof of this example is in the appendix (App. F).

```
a := 0;
b := 1;
i := 0;
while (i < n) {
    tmp := b;
    b := a + b;
    a := tmp;
    i := i + 1
}
```

Fig. 8. The program $C_{fib}$, which computes the $n$-th Fibonacci number.

*Restriction to* $\forall^*\exists^*$*-hyperproperties.* The rule *While*-$\forall^*\exists^*$ is quite general and powerful, since it can be applied to prove any postcondition of the shape $\forall^*\exists^*$, which includes *all* safety hyperproperties, as well as liveness hyperproperties such as GNI. However, it cannot be applied for postconditions with a top-level existential quantification over states, because this would be unsound. Indeed, a triple such as $\vdash \{\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, I\}$ $C$ $\{\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, I\}$ implies that, for any $n$, there exists a state $\varphi$ such that $I$ holds for all states $\varphi'$ reached after *unrolling the loop n times*. The key issue is that $\varphi$ might not be a valid witness for states $\varphi'$ reached after *more than n loop unrollings*, and therefore we might have different witnesses for $\varphi$ for different $n$. We thus have no guarantee that there is a *global* witness that works for all states $\varphi'$ after any *number* of loop unrollings. To handle such examples, we present a rule for $\exists^*\forall^*$-hyperproperties next.

## 5.3 $\exists^*\forall^*$-Hyperproperties

The rule *While*-$\forall^*\exists^*$ can be applied for any postcondition of the form $\forall^*\exists^*$, which includes all safety hyperproperties as well as liveness hyperproperties such as GNI, but cannot be applied to prove postconditions with a top-level existential quantifier, such as postconditions of the shape $\exists^*\forall^*$ (e.g., to prove the existence of minimal executions, or to prove that a $\forall^*\exists^*$-hyperproperty is violated). In this case, we can apply the rule *While*-$\exists$ in Fig. 6. To the best of our knowledge, this is the first program logic rule that can deal with $\exists^*\forall^*$-hyperproperties for loops. This rule splits the reasoning into two parts: First, we prove that there is a *terminating* state $\varphi$ such that the hyper-assertion $P_\varphi$ holds after some number of loop unrollings. This is achieved via the first premise of the rule, which requires a well-founded relation $\prec$, and a variant $e(\varphi)$ that strictly decreases at each iteration, until $b(\varphi)$ becomes false and $\varphi$ exits the loop.[10] In a second step, we fix the state $\varphi$ (since it has exited the loop), which corresponds to our global witness, and prove $\vdash \{P_\varphi\}$ **while** $(b)$ $\{C\}$ $\{Q_\varphi\}$ using any loop rule. For example, if $P_\varphi$ has another top-level existential quantifier, we can apply the rule *While*-$\exists$ once more; if $P_\varphi$ is a $\forall^*\exists^*$ hyper-assertion, we can apply the rule *While*-$\forall^*\exists^*$.

As an example, consider proving that the program $C_m$ in Fig. 9 has a final state with a minimal value for $x$ and $y$, a hyperproperty that cannot be expressed in any other Hoare logic. Formally, we want to prove the triple $\{\neg emp \land \square(k \geq 0)\}$ $C_m$ $\{\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, \varphi(x) \leq \alpha(x) \land \varphi(y) \leq \alpha(y)\}$. Since the set of initial states is not empty and $k$ is always non-negative, we know that there is an initial state with a minimal value for $k$. We prove that this state leads to a final state with minimal values for $x$ and $y$, using the rule *While*-$\exists$. For the first premise, we choose the variant[11] $k - i$, and the invariant $P_\varphi \triangleq (\forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \land 0 \leq \varphi(y) \leq \alpha(y) \land \varphi(k) \leq \alpha(k) \land \varphi(i) = \alpha(i))$, capturing both that $\varphi$ has minimal values for $x$ and $y$, but also that $\varphi$ will be the first state to exit the loop. We prove that this is indeed an invariant for the loop, by choosing $r = 2$ for the non-deterministic assignment for $\varphi$. Finally, we prove the second premise with $Q_\varphi \triangleq (\forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \land 0 \leq \varphi(y) \leq \alpha(y))$ and the rule *While*-$\forall^*\exists^*$. The proof of this example is in the appendix (App. G).

```
x := 0;
y := 0;
i := 0;
while (i < k) {
  r := nonDet();
  assume r ≥ 2;
  t := x;
  x := 2 * x + r;
  y := y + t * r;
  i := i + 1
}
```

Fig. 9. A program with a final state with minimal values for $x$ and $y$.

## 6 RELATED WORK

*Overapproximate (relational) Hoare logics.* Hoare Logic originated with the seminal works of Floyd [Floyd 1967] and Hoare [Hoare 1969], with the goal of proving programs functionally correct. Relational Hoare Logic [Benton 2004] (RHL) extends Hoare Logic to reason about (2-safety) hyperproperties of a single program as well as properties relating the executions of two different programs (e.g., semantic equivalence). RHL's ability to relate the executions of two different programs is also useful in the context of proving 2-safety hyperproperties of a single program, in particular, when the two executions take different branches of a conditional statement. In comparison, Hyper Hoare Logic can prove and disprove hyperproperties of a single program (Sect. 3.5), but requires a program transformation to express relational properties (see end of App. C.3). Extending Hyper Hoare Logic to multiple programs is interesting future work.

RHL has been extended in many ways, for example to deal with heap-manipulating [Yang 2007] and higher-order programs [Aguirre et al. 2017]. A family of Hoare and separation logics [Amtoft et al. 2006; Costanzo and Shao 2014; Eilers et al. 2023; Ernst and Murray 2019] designed to prove non-interference [Volpano et al. 1996] specializes RHL by considering triples with a single program, similar to Hyper Hoare Logic. Naumann [2020] provides an overview of the principles underlying

[10]Note that the existentially-quantified state $\varphi$ in the postcondition of the first premise of the rule *While*-$\exists$ does *not* have to be from the same execution as the one in the precondition.

[11]We interpret $\prec$ as $<$ between natural numbers, i.e., $a \prec b$ iff $0 \leq a$ and $a < b$, which is well-founded.

relational Hoare logics. Cartesian Hoare Logic [Sousa and Dillig 2016] (CHL) extends RHL to reason about hyperproperties of $k$ executions, with a focus on automation and scalability. CHL has recently been reframed [D'Osualdo et al. 2022] as a weakest-precondition calculus, increasing its support for proof compositionality. Hyper Hoare Logic can express the properties supported by CHL, in addition to many other properties; automating Hyper Hoare Logic is future work.

*Underapproximate program logics.* Reverse Hoare Logic [de Vries and Koutavas 2011] is an underapproximate variant of Hoare Logic, designed to prove the existence of good executions. The recent Incorrectness Logic [O'Hearn 2019] adapts this idea to prove the presence of bugs. Incorrectness Logic has been extended with concepts from separation logic to reason about heap-manipulating sequential [Raad et al. 2020] and concurrent [Raad et al. 2022] programs. It has also been extended to prove the presence of insecurity in a program (i.e., to disprove 2-safety hyperproperties) [Murray 2020]. Underapproximate logics have been successfully used as foundation of industrial bug-finding tools [Blackshear et al. 2018; Distefano et al. 2019; Gorogiannis et al. 2019; Le et al. 2022]. Hyper Hoare Logic enables proving and disproving hyperproperties within the same logic.

Several recent works have proposed approaches to unify over- and underapproximate reasoning. Exact Separation Logic [Maksimović et al. 2023] can establish both overapproximate and (backward) underapproximate properties over single executions of heap-manipulating programs, by employing triples that describe *exactly* the set of reachable states. Local Completeness Logic [Bruni et al. 2021, 2023] unifies over- and underapproximate reasoning in the context of abstract interpretation, by building on Incorrectness Logic, and enforcing a notion of *local completeness* (no false alarm should be produced relatively to some fixed input). HL and IL have been both embedded in a Kleene algebra with diamond operators and countable joins of tests [Möller et al. 2021]. Dynamic Logic [Harel 1979] is an extension of modal logic that can express both overapproximate and underapproximate guarantees over single executions of a program. To the best of our knowledge, dynamic logic has not been extended to properties of multiple executions.

Outcome Logic [Zilberstein et al. 2023] (OL) unifies overapproximate and (forward) underapproximate reasoning for heap-manipulating and probabilistic programs, by combining and generalizing the standard overapproximate Hoare triples with forward underapproximate triples (see App. C.2). OL (instantiated to the powerset monad) uses a semantic model similar to our extended semantics (Def. 4), and a similar definition for triples (Def. 5). Moreover, a theorem similar to our Thm. 4 holds in OL, i.e., invalid OL triples can be disproven within OL. The key difference with Hyper Hoare Logic is that OL does not support reasoning about hyperproperties. OL assertions are composed of atomic unary assertions, which allow it to express the existence and the absence of certain states, but not to relate states with each other, which is key to expressing hyperproperties. OL does not provide logical variables, on which we rely to express certain hyperproperties (see Sect. 2.2).

*Logics for $\forall^*\exists^*$-hyperproperties.* Maillard et al. [2019] present a general framework for defining relational program logics for arbitrary monadic effects (such as state, input-output, nondeterminism, and discrete probabilities), for two executions of two (potentially different) programs. Their key idea is to map *pairs* of (monadic) computations to relational specifications, using relational *effect observations*. In particular, they discuss instantiations for $\forall\forall$-, $\forall\exists$-, and $\exists\exists$-hyperproperties. RHLE [Dickerson et al. 2022] supports overapproximate and (a limited form of) underapproximate reasoning, as it can establish $\forall^*\exists^*$-hyperproperties, such as generalized non-interference (Sect. 2.3) and program refinement. Both logics can reason about relational properties of multiple programs, whereas Hyper Hoare Logic requires a program transformation to handle such properties. On the other hand, our logic supports a wider range of underapproximate reasoning and can express properties not handled by any of them, e.g., $\exists^*\forall^*$-hyperproperties. Moreover, even for $\forall^*\exists^*$-hyperproperties, Hyper Hoare Logic provides while loop rules that have no equivalent in these logics, such as the rules *While*-$\exists$ (useful in this context for $\exists^*$-hyperproperties) and *While*-$\forall^*\exists^*$ (Sect. 5).

*Probabilistic Hoare logics.* Many assertion-based logics for probabilistic programs have been proposed [Barthe et al. 2018, 2019b; Corin and Den Hartog 2006; Den Hartog and de Vink 2002; Ramshaw 1979; Rand and Zdancewic 2015]. These logics typically employ assertions over *probability (sub-)distributions* of states, which bear some similarities to hyper-assertions: Asserting the existence (resp. absence) of an execution is analogous to asserting that the probability of this execution is strictly positive (resp. zero). Notably, our loop rule *While*-$\forall^*\exists^*$ draws some inspiration from the rule *While* of Barthe et al. [2018], which also requires an invariant that holds for all *unrollings* of the loop. These probabilistic logics have also been extended to the relational setting [Barthe et al. 2009], for instance to reason about the equivalence of probabilistic programs.

*Verification of hyperproperties.* The concept of hyperproperties has been formalized by Clarkson and Schneider [2008]. Verifying that a program satisfies a $k$-safety hyperproperty can be reduced to verifying a trace property of the *self-composition* of the program [Barthe et al. 2011b] (e.g., by sequentially composing the program with renamed copies of itself). Self-composition has been generalized to product programs [Barthe et al. 2011a; Eilers et al. 2019]. (Extensions of) product programs have also been used to verify relational properties such as program refinement [Barthe et al. 2013] and probabilistic relational properties such as differential privacy [Barthe et al. 2014]. The temporal logics LTL, CTL, and CTL*, have been extended to HyperLTL and HyperCTL [Clarkson et al. 2014] to specify hyperproperties, and model-checking algorithms [Beutner and Finkbeiner 2022, 2023; Coenen et al. 2019; Hsu et al. 2021] have been proposed to verify hyperproperties expressed in these logics, including hyperproperties outside of the safety class. Unno et al. [2021] propose an approach to automate relational verification based on an extension of constrained Horn-clauses. Relational properties of imperative programs can be verified by reducing them to validity problems in trace logic [Barthe et al. 2019a]. Finally, the notion of hypercollecting semantics [Assaf et al. 2017] (similar to our extended semantics) has been proposed to statically analyze information flow using abstract interpretation [Cousot and Cousot 1977].

## 7 CONCLUSION AND FUTURE WORK

We have presented Hyper Hoare Logic, a novel, sound, and complete program logic that supports reasoning about a wide range of hyperproperties. It is based on a simple but powerful idea: reasoning directly about the *set* of states at a given program point, instead of a fixed number of states. We have demonstrated that Hyper Hoare Logic is very expressive: It can be used to prove or disprove *any* program hyperproperty over terminating executions, including $\exists^*\forall^*$-hyperproperties and hyperproperties relating an unbounded or infinite number of executions, which goes beyond the properties handled by existing Hoare logics. Moreover, we have presented syntactic rules, compositionality rules, and rules for loops that capture important proof principles naturally.

We believe that Hyper Hoare Logic is a powerful foundation for reasoning about the correctness and incorrectness of program hyperproperties. We plan to build on this foundation in our future work. First, we will explore automation for Hyper Hoare Logic by developing an encoding into an SMT-based verification system such as Boogie [Leino 2008]. Second, we will extend the language supported by the logic, in particular, to include a heap. The main challenge will be to adapt concepts from separation logic to hyper-assertions, e.g., to find a suitable definition for the separating conjunction of two hyper-assertions. Third, we will explore an extension of Hyper Hoare Logic that can relate multiple programs.

## REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29.

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. *SIGPLAN Not.* 41, 1 (jan 2006), 91–102. https://doi.org/10.1145/1111320.1111046

Mounir Assaf, David A Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. *ACM SIGPLAN Notices* 52, 1 (2017), 874–887.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011a. Relational verification using product programs. In *International Symposium on Formal Methods*. 200–214.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-safety: Asymmetric product programs for relational program verification. In *International Symposium on Logical Foundations of Computer Science*. 29–43.

Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011b. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. 2019a. Verifying relational properties using trace logic. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 170–178.

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An assertion-based program logic for probabilistic programs. In *European Symposium on Programming*. 117–144.

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving differential privacy in Hoare logic. In *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 411–424.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 90–101.

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019b. A Probabilistic Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 55 (dec 2019), 30 pages. https://doi.org/10.1145/3371123

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Cham, 341–362.

Raven Beutner and Bernd Finkbeiner. 2023. AutoHyper: Explicit-State Model Checking for HyperLTL. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 145–163.

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470608

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2, Article 15 (mar 2023), 45 pages. https://doi.org/10.1145/3582267

Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*. 265–284.

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *21st IEEE Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2008.7

Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. 2019. Verifying hyperliveness. In *International Conference on Computer Aided Verification*. 121–139.

Ricardo Corin and Jerry Den Hartog. 2006. A probabilistic Hoare-style logic for game-based cryptographic proofs. In *International Colloquium on Automata, Languages, and Programming*. 252–263.

David Costanzo and Zhong Shao. 2014. A Separation Logic for Enforcing Declarative Information Flow Control Policies. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). 179–198.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). 155–171.

JI Den Hartog and Erik P de Vink. 2002. Verifying probabilistic programs using a Hoare like logic. *International journal of foundations of computer science* 13, 03 (2002), 315–340.

Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings* (Auckland, New Zealand). 67–87. https://doi.org/10.1007/978-3-031-21037-2_4

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

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

Marco Eilers, Thibault Dardinier, and Peter Müller. 2023. CommCSL: Proving Information Flow Security for Concurrent Programs Using Abstract Commutativity. *Proc. ACM Program. Lang.* 7, PLDI, Article 175 (jun 2023), 26 pages. https://doi.org/10.1145/3591289

Marco Eilers, Peter Müller, and Samuel Hitz. 2019. Modular product programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 1 (2019), 1–37.

Gidon Ernst and Toby Murray. 2019. SecCSL: Security Concurrent Separation Logic. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Cham, 208–230.

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium in Applied Mathematics* (1967), 19–32.

Nissim Francez. 1983. Product properties and their direct verification. *Acta informatica* 20, 4 (1983), 329–344.

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (jan 2019), 29 pages. https://doi.org/10.1145/3290370

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-Termination. *SIGPLAN Not.* 43, 1 (jan 2008), 147–158. https://doi.org/10.1145/1328897.1328459

David Harel. 1979. *First-order dynamic logic.* Springer.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 94–112.

Thomas Kleymann. 1999. Hoare Logic and Auxiliary Variables. *Form. Asp. Comput.* 11, 5 (dec 1999), 541–566. https://doi.org/10.1007/s001650050057

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. https://doi.org/10.1145/3527325

K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (dec 2019), 33 pages. https://doi.org/10.1145/3371072

Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, Vol. 263. 19:1–19:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

Daryl McCullough. 1987. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy*. IEEE, 161–161.

John McLean. 1996. A general theory of composition for a class of" possibilistic" properties. *IEEE Transactions on Software Engineering* 22, 1 (1996), 53–67.

Bernhard Möller, Peter O'Hearn, and Tony Hoare. 2021. On Algebra of Program Correctness and Incorrectness. In *Relational and Algebraic Methods in Computer Science*, Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter (Eds.). Cham, 325–343.

Toby Murray. 2020. An Under-Approximate Relational Logic: Heralding Logics of Insecurity, Incorrect Implementation and More. https://doi.org/10.48550/ARXIV.2003.04791

David A. Naumann. 2020. Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, Tiziana Margaria and Bernhard Steffen (Eds.). Cham, 93–116.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, Berlin, Heidelberg.

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. https://doi.org/10.1145/3371078

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. Cham, 225–252.

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (jan 2022), 29 pages. https://doi.org/10.1145/3498695

Lyle Harold Ramshaw. 1979. *Formalizing the Analysis of Algorithms*. PhD thesis. Stanford University.

Robert Rand and Steve Zdancewic. 2015. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. *Electronic Notes in Theoretical Computer Science* 319 (2015), 351–367. https://doi.org/10.1016/j.entcs.2015.12.021 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures*, Luca de Alfaro (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–302.

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/2908080.2908092

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 742–766.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

Hirotoshi Yasuoka and Tachio Terauchi. 2010. On Bounding Problems of Quantitative Information Flow. In *Computer Security – ESORICS 2010*, Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 357–372.

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. https://www.cs.cornell.edu/~noamz/files/pubs/outcome.pdf