

An Automated Verifier for Quantitative Hyperproperties

Project Report of a Practical Work

Christopher Cohnen

Co-Supervised by Dr. Thibault Dardinier & Hongyi Ling

ETH Zurich, Switzerland, April 3, 2026

Abstract

Zhang et al. [17] introduced the quantitative weakest hyper pre transformer to formally reason about hyperquantities, a quantitative generalization of hyperproperties. This transformer can in particular be used for proving expected values of program variables that relate multiple executions of a probabilistic program, e.g. the expected difference of a variable over all executions.

This is the report of a practical work in which we build a verifier for hyperquantities. To this end, we automate the quantitative weakest hyper pre transformer to generate verification conditions and encode these conditions into an SMT formula. We also introduce simpler rules for the transformer to automate it.

1 Introduction

In critical scenarios, ensuring correctness of programs is very important. Contrary to code review and testing, formally proving correctness of programs gives complete guarantees about the absence of bugs. Hoare Logic [9] has become an important technique for formally proving correctness of programs. It can be used to verify whether Hoare triples $\{P\} C \{Q\}$ hold, i.e. whether a program C fulfills some predicate Q over final states, when it is run on initial states that satisfy a pre-condition P . Classical properties supported by Hoare Logic are

thus predicates over states. Additionally, Dijkstra [6] introduced a weakest-pre transformer that takes a program C and a post-condition Q and computes the weakest pre-condition P such that the Hoare triple $\{P\} C \{Q\}$ holds.

Many extensions of Hoare logic have been introduced to be able to prove more general properties. One of these extensions is Hyper Hoare Logic [5]. It enables proving hyperproperties, i.e. properties that relate multiple executions of a program. These hyperproperties are predicates over sets of states. One example is non-interference: It is satisfied if any two executions with the same public input have the same public output [8].

On the other hand, properties have been extended to quantities [12]. In contrast to properties that assign truth values to states, quantities assign real values to states. Consider the case where we want to prove that the running time of a program is equal to 5 loop iterations. For this, we encode the running time as a program variable c that counts up in every loop iteration. Using normal properties, we can embed this property into a post-condition Q that checks whether the value of c in a final state σ is equal to 5:

$$Q(\sigma) := (\sigma(c) == 5).$$

The weakest pre-condition P such that the Hoare Triple $\{P\} C \{Q\}$ holds, tells us under which pre-condition the running time is indeed 5. With quantities, we can instead directly reason about the final value of c by setting the post-quantity f to be the value of c :

$$f(\sigma) := \sigma(c).$$

The weakest pre-quantity g not only tells us whether the running time is always equal to 5, but also gives the exact running time for every initial state, no matter whether it is 5 or not. Hence, we can express some properties more directly using quantities, and it enables reasoning about expected values [10].

Zhang et al. [17] combined these two extensions by introducing hyperquantities, functions that assign values to quantities. One can view the input of a hyperquantity as a distribution over states, so hyperquantities map distributions over states to real values. This is an extension of hyperproperties in two dimensions: Firstly, the input are not sets of states, but distributions over states and secondly, the outputs are not truth values but real values. These hyperquantities enable reasoning about quantitative hyperproperties such as expected values, variances and sensitivity [7, 17, 3]. For this, Zhang et al. [17] introduced a weakest-pre transformer that can reason about such properties.

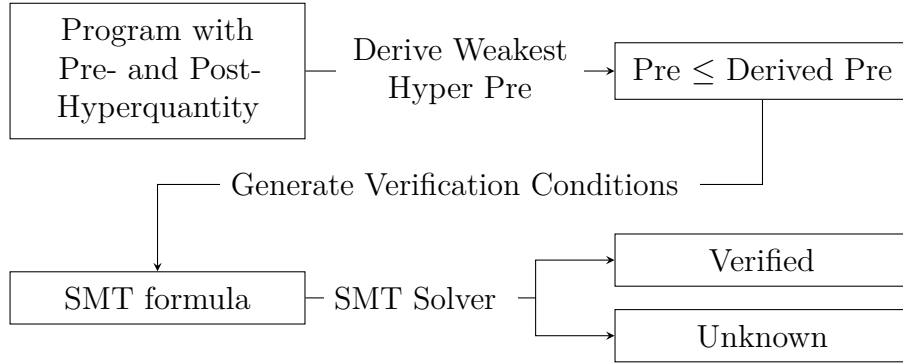


Figure 1: Overview of the Verifier

However, formally verifying any properties by hand can be very tedious and error-prone, especially when the programs and properties become non-trivial. Hence, formal verification is automated by tools, such like Hypra for Hyper Hoare Logic [4] and Caesar for quantities [14]. Until now, no automation existed for hyperquantities and the quantitative weakest hyper-pre transformer of Zhang et al. [17]. Still, the rules of the weakest-pre transformer are syntax-directed and mechanical for some classes of programs and hyperquantities. We built an automated verifier for proving lower- and upper-bounds on the weakest pre-hyperquantity for a specific kind of probabilistic programs and multilinear hyperquantities.

The verifier works in three stages as shown in Figure 1: We give it a program along with a pre- and a post-hyperquantity. Using the post-hyperquantity, the verifier derives a bound on the weakest hyper-pre.¹ In order to compare this derived pre-hyperquantity to the given pre-hyperquantity, it generates an SMT formula representing the negation of the inequality and gives it to an SMT solver. If the formula is unsatisfiable, the verifier can conclude that the given pre-hyperquantity is indeed a bound for the weakest hyper pre.

In this report, we first give some background on quantitative weakest hyper pre and define programs in Section 2. In Section 3, we describe the key idea of the verifier and define the language of hyperquantities that our verifier deals with. Then, we introduce simpler rules for the quantitative weakest hyper pre transformer for loop-free programs in Section 4 and invariant rules for loops

¹For loop-free programs, the verifier derives the precise weakest hyper pre.

in Section 5. In Section 6, we describe the implementation of the transformer and the conversion into an SMT formula. We conclude with an evaluation in Section 7 and limitations and future work in Section 8.

Contributions Our main contributions are:

- A language of hyperquantities that is expressive and for which the verification can be automated.
- Simpler rules for the quantitative weakest hyper pre transformer for the chosen language of hyperquantities.
- Invariant-based rules for the transformer for while-loops.
- An algorithm to encode inequalities of hyperquantities into SMT formulas.
- An implementation of the quantitative weakest hyper pre transformer and the SMT encoding.

2 Background

Our work is based on the quantitative weakest hyper pre transformer from Zhang et al. [17]. In this section, we give the definition of the transformer and the programs the transformer works on.

2.1 Programs

Zhang et al. [17] deal with weighted programs whose semantics is defined on states. A state s is a map from variables to values. Values can be booleans (Bool), unsigned integers (UInt) or elements of $\mathbb{R}_{\geq 0}^{\infty}$ (EUREal), i.e. non-negative reals or ∞ . We denote the set of all states by Σ .

The weighted programs are defined as follows:

$$C ::= x := e \mid * e \mid C ; C \mid \{C\} + \{C\} \mid \text{while } (e) \{C\}.$$

Here, e is an expression. Assignment, sequential composition (later written without semicolon) and while loops are common. The plus-operator is a nondeterministic choice between two programs. And the weighting $* e$ weights a program path by a real value. This can e.g. be used to express a probabilistic choice, i.e. choosing between two programs with a probability:

$$\{ * 1/2 ; C_1 \} + \{ * 1/2 ; C_2 \}.$$

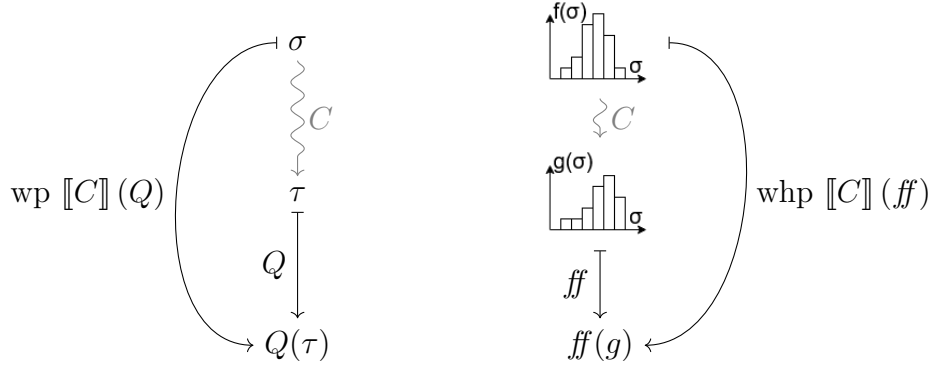


Figure 2: Intuition about wp and whp and the connection between them.

We also define assumptions and if-conditionals as syntactic sugar:

$$\begin{aligned} \text{assume } (\varphi) &:= * [e] \\ \text{if } (e) \{C_1\} \text{ else } \{C_2\} &:= \{\text{assume } (e) ; C_1\} + \{\text{assume } (\neg e) ; C_2\}. \end{aligned}$$

The Iverson bracket $[e]$ converts a boolean expression e into 0 or 1 [11].

2.2 Quantitative Weakest Hyper Pre

The classical weakest pre transformer wp takes a program C and a predicate over final program states Q and maps it to a predicate over initial states $P = \text{wp } [C] (Q)$. It does it in a way that guarantees that when executing C on an initial state that satisfies P , we reach a final state that satisfies Q (cf. Figure 2). Additionally, it ensures that the set of states satisfying P is the biggest (i.e. weakest) set of states that fulfills this property. The precondition P can be seen as a function that takes a state and evaluates the postcondition on all final states that can be reached by it.

While predicates are functions $\Sigma \rightarrow \{\text{true}, \text{false}\}$, hyperquantities are defined as functions $(\Sigma \rightarrow \mathbb{R}_{\geq 0}^\infty) \rightarrow \mathbb{R}_{\geq 0}^\infty$, i.e. functions from distributions over states to real values [17, choosing the semiring $\mathbb{R}_{\geq 0}^\infty$]. Similar to above, the quantitative weakest hyper pre transformer whp takes a program C and a hyperquantity over final states ff and maps it to a hyperquantity over initial states $gg = \text{whp } [C] (ff)$. This pre-hyperquantity gg takes a distribution over initial states and evaluates the post-hyperquantity on the distribution over the final states that is reached by executing the program on the initial distribution (cf. Figure 2).

The whp transformer is defined using rules for the different statements of a program (cf. Section 4). Hence, deriving the weakest pre-hyperquantity works by applying the rule matching the program.

3 Key Ideas

In this section, we give the key ideas of our verifier. This includes the choice of the class of hyperquantities that our verifier supports.

3.1 Verifier Overview

Figure 1 shows the stages in which the verifier works. The first stage of deriving the pre-hyperquantity using whp is given. However, it is not necessarily clear what to do with the result.

3.1.1 Checking Entailment

The result of the classical wp is easy to interpret: When we encode a correctness property into a postcondition Q , $\text{wp } \llbracket C \rrbracket (Q)$ yields a predicate over initial states that has to be fulfilled to satisfy the correctness property. We usually additionally state a precondition P that characterizes all possible initial states. If this precondition implies $\text{wp } \llbracket C \rrbracket (Q)$, i.e. the Hoare triple $\{P\} C \{Q\}$ holds, we know that C satisfies the property Q for the initial states that satisfy P .

We follow a similar approach for the whp transformer. We additionally state a pre-hyperquantity ff_{pre} that expresses the expected pre-hyperquantity and check that the actually derived pre-hyperquantity entails ff_{pre} . However, the whp transformer yields a real-valued function. Hence, we cannot use implications to bound the pre-hyperquantity, but use $\leq$ instead. So we check that $\mathit{ff}_{pre} \leq \text{whp } \llbracket C \rrbracket (\mathit{ff}_{post})$. In some cases, we instead want to verify an upper-bound, i.e. $\geq$ instead of $\leq$.

3.1.2 Example

To illustrate how the verifier works, we look at a simple example. A simple hyperquantity is the expected difference of a program variable across all executions. Consider e.g. the program C in Figure 3 that randomly assigns 1 or 2 to x with probability $\frac{1}{2}$. The hyperquantity after the `post` keyword expresses

```

method random_assignment( $x$ : UInt)
  fprob
  pre 1/2
  post  $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot |s_1(x) - s_2(x)|$ 
{
  {
    * 1/2
     $x := 1$ 
  } + {
    * 1/2
     $x := 2$ 
  }
}

```

Figure 3: A simple program with pre- and post-hyperquantity.

the expected absolute difference of x between any two states s_1 and s_2 . As the two states s_1 and s_2 have the same x with probability $\frac{1}{2}$, and x differs by 1 also with probability $\frac{1}{2}$, the expected difference should be $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$. This is the pre-hyperquantity that we state after the **pre** keyword. Hence, the verification goal is to prove

$$\frac{1}{2} \leq \text{whp } \llbracket C \rrbracket \left(\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot |s_1(x) - s_2(x)| \right).$$

3.2 Hyperquantities

In general, hyperquantities are defined as function $\mathbb{A} \rightarrow \mathbb{R}_{\geq 0}^\infty$ where $\mathbb{A}: \Sigma \rightarrow \mathbb{R}_{\geq 0}^\infty$ are distributions over states. Our verifier deals with a class of hyperquantities that allow reasoning about expectation values. To this end, we define the language as follows:

$$\begin{aligned}
\text{HQ} &::= \sum s.f(s) \cdot \text{HQ} \mid \text{QFHQ} \mid \text{HQ} + \text{HQ} \mid \text{HQ} \cdot \text{HQ} \mid -\text{HQ} \\
\text{QFHQ} &::= c \mid s(e) \mid [\text{QFHQ}] \mid \text{QFHQ } \textit{binary} \text{ QFHQ} \mid \textit{unary} \text{ QFHQ} \\
e &::= c \mid x \mid [e] \mid e \textit{ binary } e \mid \textit{unary } e \mid \exp(e, e) \\
\textit{binary} &\in \{+, -, \cdot, \wedge, \vee, =, \neq, \leq, \geq, <, >\} \\
\textit{unary} &\in \{-, \neg\}
\end{aligned}$$

Here, $s \in \Sigma$ is a state, x a program variable, $c \in \{\text{true}, \text{false}\} \cup \mathbb{R}_{\geq 0}^\infty$ a constant and the exponential function is only defined for integer exponents. The term

$s(e)$ stands for evaluating the expression e for the state s . We can also view e as a function $\Sigma \rightarrow \{\text{true}, \text{false}\} \cup \mathbb{R}_{\geq 0}^\infty$ that evaluates itself on a given state.

The function f corresponds to the argument of the hyperquantity, i.e. $f: \Sigma \rightarrow \mathbb{R}_{\geq 0}^\infty$. Hence, a term ff build from this syntax becomes a hyperquantity by putting λf in front of it, but we usually omit the λf . We call this f “input distribution”.

The sum quantifier $\sum s$ binds state variables and is always paired with $f(s)$. With this, we can express e.g. the expected value of a variable x :

$$\sum s.f(s) \cdot s(x).$$

You can understand it as an infinite sum over all values that the variables in s can have. For s , we only consider variables that appear in the hyperquantity, so a state is basically a tuple of program variables. And additionally, by restricting f to have finite support, we make the sum finite. This is the reason why a sum quantifier is always paired with $f(s)$. It should be noted though that this is not really a restriction:

- If we have a term $\sum s.ff$, where no $f(s)$ appears somewhere in ff , this sum is independent of the argument of the hyperquantity and thus also independent of the program. Thus, we can replace it by a constant (e.g. $\sum s.s(x) = \infty$ for x taking values from $\mathbb{R}_{\geq 0}^\infty$).
- If we have a term $\sum s.ff$, where $f(s)$ appears somewhere in ff , we can rewrite it to take the $f(s)$ out of ff to get $\sum s.f(s) \cdot ff'$ (or rewrite it to get a constant term without an $f(s)$ as above).

The HQ grammar above is the simple version that does not type check the internal expressions. Some operators only work on some types, so not all hyperquantities that can be constructed from the grammar are valid. For the outcome, we assume that HQ has always the type $\mathbb{R}_{\geq 0}^\infty$, whereas QFHQ might also have a boolean type. In the following, we denote by HQ the set of all hyperquantities that can be derived from the grammar and whose internal expressions type check. Additionally, when not stated differently, we assume that all $ff \in \text{HQ}$ are closed with regard to state variables, i.e. all s appearing in ff are bound by a sum quantifier.

The set HQ is a subset of hyperquantities and is chosen such that the elements fulfill a specific linearity property.

Definition 3.1 (k -Linear Hyperquantities): We call a hyperquantity $ff: \mathbb{A} \rightarrow \mathbb{R}_{\geq 0}^\infty$ k -linear, if there exists $ff': \mathbb{A} \times \dots \times \mathbb{A} \rightarrow \mathbb{R}_{\geq 0}^\infty$ with k arguments such that

$ff = \lambda f. ff'(f, \dots, f)$ and ff' is a k -linear function, i.e.

$$ff'(f_1, \dots, \alpha f_i + f'_i, \dots, f_k) = \alpha ff'(f_1, \dots, f_i, \dots, f_k) + ff'(f_1, \dots, f'_i, \dots, f_k)$$

for all $1 \leq i \leq k$, $f_1, \dots, f_k, f'_i \in \mathbb{A}$ and $\alpha \in \mathbb{R}_{\geq 0}^\infty$.

For k -linear ff , we often use ff as being exactly equal to ff' , i.e. we treat ff as a function taking k arguments.

Using this definition, we can characterize the linearity property of hyperquantities in HQ.

Lemma 3.2 (Linearity of sum-free $ff \in \text{HQ}$): *Let $ff \in \text{HQ}$ without using the “+”-constructor of HQ (note that the “+”-constructor of QFHQ is allowed). Let additionally k be the number of sum quantifiers appearing in ff . If $k > 0$, ff is k -linear, and ff is constant otherwise.*

Proof Sketch: Let ff and k be as above. We define the function $ff': \mathbb{A} \times \dots \times \mathbb{A} \rightarrow \mathbb{R}_{\geq 0}^\infty$ taking k arguments by letting ff' have the same form as ff except that the f corresponding to the i -th $\sum$ is renamed to f_i and treated as the i -th argument of ff' for each $i = 1, \dots, k$. We claim that this ff' is k -linear.

Let $1 \leq i \leq k$, $f_1, \dots, f_k, f'_i \in \mathbb{A}$, $\alpha \in \mathbb{R}_{\geq 0}^\infty$ be arbitrary. We have to show for the i -th argument:

$$ff'(f_1, \dots, \alpha f_i + f'_i, \dots, f_k) = \alpha ff'(f_1, \dots, f_i, \dots, f_k) + ff'(f_1, \dots, f'_i, \dots, f_k).$$

We can prove this by structural induction on ff' . The interesting case is the one of the sum quantifier corresponding to the i -th argument of ff' : Here we can push the $+$ and α to the outside:

$$\sum_s (\alpha f_i + f'_i)(s) \cdot gg = \alpha \sum_s f_i(s) \cdot gg + \sum_s f'_i(s) \cdot gg.$$

The other cases show that we can push the $+$ and α further to the outside. $\square$

When we allow the “+”-constructor in HQ, we get linear combinations of k -linear hyperproperties and constants:

Theorem 3.3 (Linearity of elements of HQ): *For all $ff \in \text{HQ}$, there exist pieces $ff_1, \dots, ff_n$ such that $ff = \sum_{i=1}^n ff_i$ and for all $i \in [1, n]$, $ff_i \in \text{HQ}$ and ff_i is a constant or k -linear for some $k \in \mathbb{N}$.*

Proof Sketch: We start by proving that we can split $ff \in \text{HQ}$ into pieces ff_i such that $ff = \sum_{i=1}^n ff_i$ and all ff_i contain no “+” on the HQ level. This can

C	$\text{whp } \llbracket C \rrbracket (ff)$	
$x := e$	$ff[x \mapsto e]$	(Assignment)
$* e$	$\lambda f. ff(f \cdot e)$	(Weighting)
$C_1 ; C_2$	$\text{whp } \llbracket C_1 \rrbracket (\text{whp } \llbracket C_2 \rrbracket (ff))$	(Sequential Composition)
$\{C_1\} + \{C_2\}$	$\text{whp } \llbracket C_1 \rrbracket (ff) + \text{whp } \llbracket C_2 \rrbracket (ff)$	(Nondeterministic Choice)

Table 1: Inductive rules for the quantitative weakest hyper pre transformer for loop-free programs. The nondeterministic choice rule only works for linear hyperquantities. The assignment rule is introduced by us and the others are by Zhang et al. [17].

be proven by structural induction on ff similar as in the previous lemma. In all cases, we can push the “+” to the outside using fundamental arithmetic rules.

After this, all pieces ff_i contain no “+” on the HQ level anymore, and we can apply Lemma 3.2 to conclude that all ff_i are multilinear or constant. $\square$

We will see in Section 4 and Section 6 why this linearity property is so useful.

Remark 3.4 (Iverson Brackets): Iverson brackets $[e]$ convert boolean expressions e into real-valued expressions. We often multiply them to the input distribution f of a hyperquantity ff . Hence, we use the following notation [17]:

$$ff \cdot [e] = [e] \cdot ff := \lambda f. ff(f \cdot [e]).$$

4 Transformer Rules for Loop-Free Programs

To automate the whp transformer, its derivation rules have to be easy to automate. Some rules presented by Zhang et al. [17] are already based on basic arithmetic operations: the rules for weighting, sequential composition and nondeterministic choice. However, the assignment rule introduces an infinite sum and the nondeterministic choice rule is restricted to linear hyperquantities. In this section, we introduce a simpler assignment rule based on a rule introduced by Cohnen [3] that completes the rules for loop-free programs (cf. Table 1). Afterwards, we extend the nondeterministic choice rule to support all hyperquantities in HQ.

4.1 Simpler Assignment Rule

Cohnen [3] presents a simpler assignment rule for a special kind of assignments. Applying this simpler rule on an assignment $x := e$ results in a syntactic replacement of all occurrences of x by e . Because we basically restricted our hyperquantities to sums over states, we can apply this rule on all assignments.

Proposition 4.1 (Simpler Rule for Assignments): *Let $ff \in \text{HQ}$. Then*

$$\text{whp } \llbracket x := e \rrbracket (ff) = ff[x \mapsto e]$$

holds where $ff[x \mapsto e]$ denotes the substitution of x by e in ff .

Proof Sketch: We prove that the original assignment rule is equal to the simpler one by structural induction on ff .

The original assignment rule is the following [17]:

$$\text{whp } \llbracket x := e \rrbracket (ff) = \lambda f. ff \left(\sum_{\alpha} f[x \mapsto \alpha] \cdot [x = e[x \mapsto \alpha]] \right).$$

Hence, it is basically a syntactic replacement of all occurrences of the argument f in ff . In the syntax of HQ , f can only appear together with a sum quantifier. Hence, the only interesting case of the induction is the sum quantifier. We omit the other cases here.

Let $\sum s. f(s) \cdot gg$ be a sum operator in ff . When we substitute f using the original rule, we get:

$$\begin{aligned} & \sum_s \left(\sum_{\alpha} f[x \mapsto \alpha] \cdot [x = e[x \mapsto \alpha]] \right) (s) \cdot gg \\ &= \sum_s \sum_{\alpha} f[x \mapsto \alpha](s) \cdot [x = e[x \mapsto \alpha]](s) \cdot gg \\ &= \sum_s \sum_{\alpha} f(s[x \mapsto \alpha]) \cdot [s(x) = s[x \mapsto \alpha](e)] \cdot gg \end{aligned}$$

Instead of summing over the possible initial values α for the assignment, we can also sum over all possible initial states s' . These s' have to be equal to the final state s up to the variable x (written as $s_{\setminus x} = s'_{\setminus x}$), and have α as the value for x . Using the equality $s' = s[x \mapsto \alpha]$, we get rid of all occurrences of α :

$$\sum_s \sum_{s' \text{ with } s_{\setminus x} = s'_{\setminus x}} f(s') \cdot [s(x) = s'(e)] \cdot gg$$

The condition $s_{\setminus x} = s'_{\setminus x}$ can be taken into the sum using Iverson brackets and directly combined with the condition $s(x) = s'(e)$ as $s = s'[x/e]$. Furthermore, we can use this condition to rewrite s using s' in gg :

$$\begin{aligned}
&= \sum_s \sum_{s'} f(s') \cdot [s = s'[x \mapsto e]] \cdot gg \\
&= \sum_s \sum_{s'} f(s') \cdot [s = s'[x \mapsto e]] \cdot gg[s \mapsto s'[x \mapsto e]] \\
&= \sum_{s'} f(s') \cdot gg[s \mapsto s'[x \mapsto e]] \cdot \sum_s [s = s'[x \mapsto e]]
\end{aligned}$$

Note that there is only one state s that satisfies $s = s'[x \mapsto e]$. Hence, the inner sum is equal to 1. Additionally, we can simplify the formula by renaming s' :

$$\begin{aligned}
&= \sum_{s'} f(s') \cdot gg[s \mapsto s'[x \mapsto e]] \\
&= \sum_s f(s) \cdot gg[s(x) \mapsto s(e)].
\end{aligned}$$

Looking at the syntax of hyperquantities, note that all variables x appearing in ff are part of an expression and thus bound to a state s . And all states s are bound to some sum quantifier. Hence, by substituting $s(x)$ by $s(e)$ for all sums quantifying over some s , we substitute all x by e . This concludes the proof of the simpler rule. $\square$

4.2 Rules for Multilinear Hyperquantities

The rules for weighting and sequential composition work for all hyperquantities [17], and we have shown the simpler assignment rule for all hyperquantities in HQ. The nondeterministic choice rule does not work for multilinear hyperquantities. The reason is that multilinear hyperquantities can relate multiple executions of a program. And these executions can behave asynchronously, i.e. one of the executions nondeterministically chooses the left program and another execution the right one. To take this into account, we propose two ways of dealing with this.

4.2.1 Sequential whp

The first approach uses the fact that a multilinear hyperquantity is linear if we fix all but one of its arguments. This allows us to use the whp rules for linear hyperquantities. For this, we define a new whp transformer.

Definition 4.2 (Sequential whp): For a program C , a k -linear hyperquantity $\mathbb{f}$ and $n \in \mathbb{N}$ with $n \leq k$, the transformer $\text{whp}_n \llbracket C \rrbracket (\mathbb{f})$ is the weakest pre-hyperquantity for $\mathbb{f}$ where all arguments but the n -th one are fixed. Formally, it is defined as:

$$\text{whp}_n \llbracket C \rrbracket (\mathbb{f}) = \lambda g. \text{whp} \llbracket C \rrbracket \left(\lambda f. \mathbb{f}(g, \dots, g, \overbrace{f}^{\text{position } n}, g, \dots, g) \right) (g).$$

This allows us to introduce a whp rule for multilinear hyperquantities for nondeterministic choices:

BILINEAR $\frac{\begin{array}{l} \mathbb{f} = \lambda f. \mathbb{g}_0(f, f), \mathbb{g}_0 \text{ is bilinear} \\ \mathbb{g}_1 = \text{whp}_1 \llbracket C \rrbracket (\mathbb{g}_0) \\ \mathbb{g}_2 = \text{whp}_2 \llbracket C \rrbracket (\mathbb{g}_1) \end{array}}{\text{whp} \llbracket C \rrbracket (\mathbb{f}) = \mathbb{g}_2}$	k-LINEAR $\frac{\begin{array}{l} \mathbb{f} = \lambda f. \mathbb{g}_0(f, \dots, f), \mathbb{g}_0 \text{ is } k\text{-linear} \\ \mathbb{g}_1 = \text{whp}_1 \llbracket C \rrbracket (\mathbb{g}_0) \\ \vdots \\ \mathbb{g}_k = \text{whp}_k \llbracket C \rrbracket (\mathbb{g}_{k-1}) \end{array}}{\text{whp} \llbracket C \rrbracket (\mathbb{f}) = \mathbb{g}_k}$
---	---

Note that the k -LINEAR rule also works for linear and bilinear hyperquantities; the BILINEAR rule just serves as clarification. The rules essentially apply whp sequentially on every input distribution f that appears in the hyperquantity.

Also note that the rule is not restricted to nondeterministic choices, but can be used on all programs. Section 6.2 explains why we only use it for nondeterministic choices.

It remains to show soundness of the new rule.

Proposition 4.3 (Soundness of the k -LINEAR rule): *The k -LINEAR rule is sound, i.e. it can be proven using the original whp rules.*

Proof: Let C be a program and $\mathbb{f}: \mathbb{A} \rightarrow \mathbb{R}_{\geq 0}^\infty$ be a k -linear hyperquantity. We prove the rule by induction on k .

Let $k = 1$, i.e. $\mathbb{f}$ is linear. We have to prove $\text{whp} \llbracket C \rrbracket (\mathbb{f}) = \text{whp}_1 \llbracket C \rrbracket (\mathbb{f})$. This trivially holds as this is the definition of whp_1 for linear hyperquantities.

Let $k + 1 > 1$ and assume the induction hypothesis that the rule is sound for k . Additionally, we have the premises of the $(k + 1)$ -LINEAR rule given. We have to prove that $\text{whp} \llbracket C \rrbracket (\mathbb{f}) = \mathbb{g}_{k+1}$.

For the proof, we need the strongest-post semantics of programs from Zhang et al. [17]. It can be understood as the semantics of the program that map a

distribution over initial states to a distribution over final states. We use the following fact [17, Theorem 4.12]:

$$\text{whp } \llbracket C \rrbracket (ff) = \lambda f. ff(\text{sp } \llbracket C \rrbracket (f)). \quad (1)$$

With this and the fact that ff is $(k+1)$ -linear, i.e. it can be understood as a function taking k arguments, we can derive:

$$\begin{aligned} \text{whp } \llbracket C \rrbracket (ff) &= \lambda g. ff(\text{sp } \llbracket C \rrbracket (g)) \\ &= \lambda g. ff(\text{sp } \llbracket C \rrbracket (g), \dots, \text{sp } \llbracket C \rrbracket (g), \text{sp } \llbracket C \rrbracket (g)). \end{aligned}$$

Now, by fixing arguments, we can convert the last argument into a separate whp derivation than the first k arguments using Eq. (1):

$$\begin{aligned} &= \lambda g. \text{whp } \llbracket C \rrbracket (\lambda f. ff(\text{sp } \llbracket C \rrbracket (g), \dots, \text{sp } \llbracket C \rrbracket (g), f)) (g) \\ &= \lambda g. \text{whp } \llbracket C \rrbracket (\lambda f. \text{whp } \llbracket C \rrbracket (\lambda h. ff(h, \dots, h, f)) (g)) (g). \end{aligned}$$

Note that $\lambda h. ff(h, \dots, h, f)$ is k -linear, because the last argument is fixed. Hence, we can use the induction hypothesis and use the k -LINEAR rule. This gives us $\mathcal{G}_k$ as the weakest pre-hyperquantity:

$$= \lambda g. \text{whp } \llbracket C \rrbracket (\lambda f. \mathcal{G}_k(g, \dots, g, f)) (g).$$

With definition of whp_{k+1} and the last premise of the $(k+1)$ -LINEAR rule, we can conclude:

$$\begin{aligned} &= \text{whp}_{k+1} \llbracket C \rrbracket (\mathcal{G}_k) \\ &= \mathcal{G}_{k+1}. \end{aligned} \quad \square$$

For a k -linear hyperquantity, the k -linear rule requires $O(k)$ whp derivations on linear hyperquantities. It should be noted that this does not necessarily represent the actual complexity, because the hyperquantities $\mathcal{G}_i$ might syntactically grow. Hence, the input size is not equal for all linear whp derivations.

4.2.2 Alternative Approach: Parallel whp

We also propose an alternative approach that naturally extends the nondeterministic choice rule for linear hyperquantities. Assume we have a bilinear hyperquantity that sums over two states. When reaching a nondeterministic choice, these two states can either behave synchronously or asynchronously.

Hence, there are four cases: Both states take the left program, both the right or one the left and the other the right program. These cases correspond to splitting the set of involved states $\{s_1, s_2\}$ into two subsets S_1 and S_2 in all possible ways, and executing the left program for S_1 and the right program for S_2 . E.g. both states taking the left program corresponds to $S_1 = \{s_1, s_2\}$ and $S_2 = \emptyset$. And s_1 executing the left program and s_2 the right one corresponds to $S_1 = \{s_2\}$ and $S_2 = \{s_1\}$.

We can extend this idea to k -linear hyperquantities with $k > 2$. We split the k states into two sets in all possible ways. By this, we consider all possibilities of some states taking the left program and some the right one.

To make this rule formal, we first have to introduce a variant of whp again:

Definition 4.4 (Parallel whp): For a program C , a k -linear hyperquantity $\mathbb{f}$ and $n_1, \dots, n_l \in [1, k]$, the transformer $\text{whp}_{[n_1, \dots, n_l]} \llbracket C \rrbracket (\mathbb{f})$ is the weakest pre-hyperquantity for $\mathbb{f}$ where all arguments but the n_i -th ones for all $1 \leq i \leq l$ are fixed:

$$\text{whp}_{[n_1, \dots, n_l]} \llbracket C \rrbracket (\mathbb{f}) = \lambda g. \text{whp} \llbracket C \rrbracket \left(\lambda f. \mathbb{f}'(g, \dots, g, \overbrace{f}^{\text{positions } n_i}, g, \dots, g) \right) (g).$$

With this transformer, we can derive whp only on specifically selected input distributions f . Note that these input distributions directly correspond to the states s that f is applied to.

For a nondeterministic choice, we can then split the number of arguments of the hyperquantity into two sets and derive whp for the left and right program for the two sets respectively:

$$\begin{array}{c} \text{NONDET-}k\text{-LINEAR} \\ \mathcal{G} = \sum_{(l_1, l_2) \text{ s.t. } l_1 \cup l_2 = l} \text{whp}_{l_1} \llbracket C_1 \rrbracket (\text{whp}_{l_2} \llbracket C_2 \rrbracket (\mathbb{f})) \\ \hline \text{whp}_l \llbracket \{C_1\} + \{C_2\} \rrbracket (\mathbb{f}) = \mathcal{G} \end{array}$$

$$\begin{array}{c} \text{NONDET-BILINEAR} \\ \mathcal{G}_1 = \text{whp}_{[1,2]} \llbracket C_1 \rrbracket (\mathbb{f}) \\ \mathcal{G}_2 = \text{whp}_{[1,2]} \llbracket C_2 \rrbracket (\mathbb{f}) \\ \mathcal{G}_3 = \text{whp}_{[1]} \llbracket C_1 \rrbracket (\text{whp}_{[2]} \llbracket C_2 \rrbracket (\mathbb{f})) \\ \mathcal{G}_4 = \text{whp}_{[2]} \llbracket C_1 \rrbracket (\text{whp}_{[1]} \llbracket C_2 \rrbracket (\mathbb{f})) \\ \hline \text{whp}_{[1,2]} \llbracket \{C_1\} + \{C_2\} \rrbracket (\mathbb{f}) = \mathcal{G}_1 + \mathcal{G}_2 + \mathcal{G}_3 + \mathcal{G}_4 \end{array}$$

Again, the NONDET-BILINEAR rule is just stated for clarification and is just a special case of the NONDET- k -LINEAR rule. We leave the rules without proof here, because we did not follow this approach further. The reason is that this rule is more complicated to use, and the complexity is higher: For a k -linear hyperquantity, applying the NONDET- k -LINEAR results in $O(2^k)$ linear whp derivations, if we assume that there is no nested nondeterministic choice. One should note that the programs of these linear whp derivations is smaller than for the sequential approach, as they only work on the left or right program of the nondeterministic choice.

4.3 Extension to Linear Combinations of Multilinear Hyperquantities

We now have whp rules for multilinear hyperquantities, but our syntax HQ allows linear combinations of multilinear hyperquantities and constants (cf. Theorem 3.3). This can easily be solved, as Zhang et al. [17] showed that for all programs C , hyperquantities $\mathbb{H}_i$ and constants $c \in \mathbb{R}_{\geq 0}^\infty$:

$$\text{whp } \llbracket C \rrbracket \left(\sum_{i=1}^n \mathbb{H}_i \right) = \sum_{i=1}^n \text{whp } \llbracket C \rrbracket (\mathbb{H}_i) \quad \text{and} \quad (1)$$

$$\text{whp } \llbracket C \rrbracket (c) = c. \quad (2)$$

Hence, for all hyperquantities $\mathbb{H} \in \text{HQ}$, we can split $\mathbb{H}$ into pieces $\mathbb{H}_i$ that are multilinear or constant using Theorem 3.3, and Eq. (1) allows deriving whp on the pieces $\mathbb{H}_i$ instead of the whole hyperquantity. Then, we can use Eq. (2) for the constant pieces and the rules from this section for the multilinear pieces.

5 Transformer Rules for Loops

The whp rules for loop-free programs and multilinear hyperquantities use basic hyperquantity transformations and are thus straight forward to implement as described in Section 6. However, the loop rule as given by Zhang et al. [17] requires reasoning about infinite sums or fixed points, which makes them inconvenient for automation. To solve this, we develop the loop rules in Figure 4.

When reasoning about multiple executions of a program, we have to consider that two executions can behave asynchronously, i.e. they execute the loop for a different number of iterations. We present rules that take this into account in the first part of this section. However, this makes the reasoning more involved

$$\begin{array}{c}
\text{WHILE-ASYNC} \\
\frac{I \leq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (I) \quad I \cdot [\neg\varphi] \leq ff \quad \text{bounded loop iterations}}{I \leq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)}
\end{array}$$

$$\begin{array}{c}
\text{WHILE-ASYNC-CO} \\
\frac{I \geq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (I) \quad I \geq ff \cdot [\neg\varphi]}{I \geq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)}
\end{array}$$

$$\begin{array}{c}
\text{WHILE-SYNC} \\
\frac{I \cdot [\varphi] \leq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] \quad I \cdot [\neg\varphi] \leq ff \quad \text{bounded loop iterations}}{\text{low}(I, \varphi) \leq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)}
\end{array}$$

$$\begin{array}{c}
\text{WHILE-SYNC-CO} \\
\frac{I \cdot [\varphi] \geq \text{whp } \llbracket C \rrbracket (\text{colow}(I, \varphi)) \cdot [\varphi] \quad I \cdot [\neg\varphi] \geq ff \cdot [\neg\varphi]}{\text{colow}(I, \varphi) \geq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)}
\end{array}$$

$$\begin{aligned}
\text{low}(ff, \varphi) &= ff \cdot [\varphi] + ff \cdot [\neg\varphi] \\
\text{colow}(ff, \varphi) &= ff \cdot [\varphi] + ff \cdot [\neg\varphi] + \text{async}_\varphi(\infty)
\end{aligned}$$

Figure 4: The whp rules for while-loops and the definition of $\text{low}(\cdot, \cdot)$ and $\text{colow}(\cdot, \cdot)$. All loop rules require choosing an invariant I .

and there are loops that always behave synchronously, e.g. a for-loop with a constant number of iterations. Hence, we additionally introduce simpler rules for such synchronous cases.

Also, the rules we presented so far derive the exact result of whp. Instead, the loop rules will derive lower- and upper-bounds on the weakest pre-hyperquantity. This allows us to reason about loops using invariants that have to be manually specified by the user.

We again only introduce rules for multilinear hyperquantities and later extend them to arbitrary hyperquantities in HQ using Section 4.3.

5.1 Asynchronous Loop Rules

5.1.1 Idea

The asynchronous loop rules are useful for loops where the number of loop iterations might differ for different states. We model this possibly asynchronous behavior by modeling one loop iteration as an if-conditional that only executes the loop body C , if the loop guard φ holds:

$$\text{if } (\varphi) \{C\} \text{ else } \{\}.$$

A loop is then intuitively equal to the infinite concatenation of this if-conditional.

Understanding the if-conditional this way has the advantage that a loop executes the same if-conditional on all states infinitely often, no matter whether the states behave synchronously or asynchronously. Hence, computing $\text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)$ is then equal to infinitely often applying $\text{whp } \llbracket \text{if } (e) \{C\} \text{ else } \{\} \rrbracket (\cdot)$ on ff . This lets us specify an invariant-based rule: If we find an invariant such that $I \leq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (I)$, we can use this to prove that $I \leq \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^\infty \rrbracket (I)$. If additionally $I \leq ff$, we know that $I \leq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (ff)$ holds and have a bound on the pre-hyperquantity.

Making this formal yields the asynchronous lower-bound and upper-bound rules as given in Figure 4. The idea and rules are based on the WHILE- $\forall^* \exists^*$ rule from Dardinier and Müller [5] and the WHILE-D rule from Barthe et al. [2].

5.1.2 Loop Semantics

To prove the loop rules, we use the following strongest-post semantics:

$$\text{sp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (f) = \lim_{n \rightarrow \infty} \text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n; \text{assume } (\neg \varphi) \rrbracket (f).$$

The sp transformer can be understood as the semantics of programs. It maps a distribution over initial states to a distribution over final states that is reached by executing the program [17].

This equation is not given in the original paper, and we leave the proof open as future work. For the usual semantics of similar programming languages, this formula has been proven [5].

5.1.3 Lower-Bound Rule

We start by proving the lower-bound rule WHILE-ASYNC (cf. Figure 4). Let the invariant I be a hyperquantity that satisfies the premises of the rule:

$$I \leq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (I) \quad (1)$$

$$I \cdot [\neg\varphi] \leq \text{ff} \quad (2)$$

We additionally assume that the number of loop iterations is bounded, i.e. that there exists $n \in \mathbb{N}$ such that

$$\text{sp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (f) = \text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n; \text{assume } (\neg\varphi) \rrbracket (f) \quad (3)$$

This also means that $\neg\varphi$ trivially holds after the loop, so:

$$\text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (f) \cdot [\neg\varphi] = \text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (f) \quad (4)$$

With these assumptions, we show the conclusion $I \leq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (\text{ff})$ of the rule:

$$\begin{aligned} & \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (\text{ff}) \\ & \geq \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (I \cdot [\neg\varphi]) && \text{(by Eq. (2))} \\ & = \lambda f. (I \cdot [\neg\varphi]) (\text{sp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (f)) && ([17, \text{Theorem 4.12}]) \\ & = \lambda f. I (\text{sp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (f) \cdot [\neg\varphi]) && \text{(def. of } [\cdot]) \\ & = \lambda f. I (\text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n; \text{assume } (\neg\varphi) \rrbracket (f) \cdot [\neg\varphi]) && \text{(by Eq. (3))} \\ & = \lambda f. I (\text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (f) \cdot [\neg\varphi] \cdot [\neg\varphi]) \\ & = \lambda f. I (\text{sp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (f)) && \text{(by Eq. (4))} \\ & = \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (I) && ([17, \text{Theorem 4.12}]) \\ & \geq I && (*) \end{aligned}$$

It remains to show the last step (*). We prove this by induction on n .

Base case: $n = 0$.

$$\begin{aligned} & \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^0 \rrbracket (I) \\ & = \text{whp } \llbracket \rrbracket (I) \\ & = I \end{aligned}$$

Induction case: $n + 1$.

$$\begin{aligned}
& \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^{n+1} \rrbracket (I) \\
&= \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\}) \rrbracket (\text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (I)) \\
&\geq \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\}) \rrbracket (I) & \text{(IH)} \\
&\geq I & \text{(by Eq. (1))}
\end{aligned}$$

5.1.4 Upper-Bound Rule

To prove the upper-bound rule WHILE-ASYNC-CO (cf. Figure 4), we do not have to assume boundedness or even termination of the loop. The reason for that is that the pre-hyperquantity of a k -linear hyperquantity for a diverging loop is 0 [17]. Hence, any pre-hyperquantity is an upper-bound for a non-terminating loop. So a pre-hyperquantity derived by only considering terminating executions of the loop is also a valid upper-bound for the non-terminating executions.

Let the invariant I be a hyperquantity that satisfies the premises of the WHILE-ASYNC-CO rule:

$$I \geq \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\}) \rrbracket (I) \quad (1)$$

$$I \geq \text{ff} \cdot [\neg\varphi] \quad (2)$$

We prove the rule by showing that for every n

$$\text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n ; \text{assume } (\neg\varphi) \rrbracket (\text{ff}) \leq I.$$

With this, we can then conclude the limit case:

$$\begin{aligned}
& \text{whp } \llbracket \text{while } (\varphi) \{C\} \rrbracket (\text{ff}) \\
&= \lim_{n \rightarrow \infty} \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n ; \text{assume } (\neg\varphi) \rrbracket (\text{ff}) \\
&\leq I
\end{aligned}$$

Let n be arbitrary. With Eq. (2) and the whp rules for sequential composition and assume, we can derive:

$$\begin{aligned}
& \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n ; \text{assume } (\neg\varphi) \rrbracket (\text{ff}) \\
&= \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (\text{whp } \llbracket \text{assume } (\neg\varphi) \rrbracket (\text{ff})) \\
&= \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (\text{ff} \cdot [\neg\varphi]) \\
&\leq \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (I) & \text{(by Eq. (2))} \\
&\leq I & (*)
\end{aligned}$$

It remains to show the last step (*). We prove this by induction on n .

Base case: $n = 0$.

$$\begin{aligned} & \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^0 \rrbracket (I) \\ &= \text{whp } \llbracket \rrbracket (I) \\ &= I \end{aligned}$$

Induction case: $n + 1$.

$$\begin{aligned} & \text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^{n+1} \rrbracket (I) \\ &= \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (\text{whp } \llbracket (\text{if } (\varphi) \{C\} \text{ else } \{\})^n \rrbracket (I)) \\ &\leq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (I) \tag{IH} \\ &\leq I \tag{by Eq. (1)} \end{aligned}$$

5.2 Synchronous Loop Rules

5.2.1 Checking Synchronicity

The synchronous rules only give correct results for executions that execute the loop the same number of times. For other executions, we can only give trivial bounds (0 for lower-bounds and ∞ for upper-bounds). Assume we have a bilinear pre-hyperquantity for a loop with two sum quantifiers that has been derived with the synchronous rules: $\text{ff} = \sum_{s_1} \sum_{s_2} \text{ff}'$. When both s_1 and s_2 satisfy or do not satisfy the loop guard φ , the synchronous loop rules will be correct and ff' indeed the correct value for these two states. However, when $s_1(\varphi) \neq s_2(\varphi)$, ff' most likely gives a wrong value, and we should replace it by the trivial bound.

To make this formal, note that we can split the sum into three parts:

$$\left(\sum_{s_1, s_2 \text{ s.t. } s_1(\varphi) \wedge s_2(\varphi)} \text{ff}' \right) + \left(\sum_{s_1, s_2 \text{ s.t. } \neg s_1(\varphi) \wedge \neg s_2(\varphi)} \text{ff}' \right) + \left(\sum_{s_1, s_2 \text{ s.t. } s_1(\varphi) \neq s_2(\varphi)} \text{ff}' \right)$$

The first two sums sum over synchronous states, where either φ holds for both or no state and the third part sums over all states that are asynchronous. By using Iverson brackets and the input distribution, we get this split with the following formula, where the last two summands correspond to the asynchronous part:

$$\begin{aligned} \text{ff} = & \lambda f. \text{ff}(f \cdot [\varphi], f \cdot [\varphi]) + \text{ff}(f \cdot [\neg\varphi], f \cdot [\neg\varphi]) \\ & + (\text{ff}(f \cdot [\varphi], f \cdot [\neg\varphi]) + \text{ff}(f \cdot [\neg\varphi], f \cdot [\varphi])) \end{aligned}$$

We can use this separation of synchronous and asynchronous states to give trivial bounds for the asynchronous parts. For this, we introduce the $\mathbf{async}_\varphi^n(c)$ hyperquantity for $c \in \{0, \infty\}$ that gives the bound c to all asynchronous states:

$$\begin{aligned}\mathbf{async}_\varphi^2(c) &= \sum_{s_1, s_2} f(s_1) \cdot f(s_2) \cdot [\neg(s_1(\varphi) = s_2(\varphi))] \cdot c \\ \mathbf{async}_\varphi^3(c) &= \sum_{s_1, s_2, s_3} f(s_1) \cdot f(s_2) \cdot f(s_3) \cdot [\neg(s_1(\varphi) = s_2(\varphi) = s_3(\varphi))] \cdot c\end{aligned}$$

Here, n corresponds to the linearity, i.e. the number of sum quantifiers of $\mathbf{async}_\varphi^n(c)$. We will match it to the linearity of ff following the intuition above to replace the ff' for the asynchronous states by c . This has two advantages: Firstly, for linear hyperquantities we do not need to care about asynchronous states, because we cannot relate two states. Hence, we can define $\mathbf{async}_\varphi^1(c) = 0$. Secondly, the **low**- and **colow**-hyperquantities that we define below are still k -linear for the same k as ff .

We use this to define the **low**- and **colow**-hyperquantities that combine the synchronous and asynchronous parts:

Definition 5.1 (low- and colow-Function): For a k -linear hyperquantity ff and expression φ , we define:

$$\begin{aligned}\mathbf{low}(ff, \varphi) &= ff \cdot [\varphi] + ff \cdot [\neg\varphi] + \mathbf{async}_\varphi^{\text{Lin}(ff)}(0) \\ &= ff \cdot [\varphi] + ff \cdot [\neg\varphi] \\ \mathbf{colow}(ff, \varphi) &= ff \cdot [\varphi] + ff \cdot [\neg\varphi] + \mathbf{async}_\varphi^{\text{Lin}(ff)}(\infty)\end{aligned}$$

Here, $\text{Lin}(ff)$ denotes the k for which ff is k -linear.

Note that **low** and **colow** do not alter the hyperquantity, if the input distribution is only non-zero for states that behave synchronously with regard to the loop guard φ :

Lemma 5.2:

$$\begin{aligned}\mathbf{low}(ff, \varphi) \cdot [\varphi] &= \mathbf{colow}(ff, \varphi) \cdot [\varphi] = ff \cdot [\varphi] \\ \mathbf{low}(ff, \varphi) \cdot [\neg\varphi] &= \mathbf{colow}(ff, \varphi) \cdot [\neg\varphi] = ff \cdot [\neg\varphi]\end{aligned}$$

Proof: We can show this with a simple derivation:

$$\begin{aligned}
\text{low}(\text{ff}, \varphi) \cdot [\varphi] &= \lambda f. \text{low}(\text{ff}, \varphi)(f \cdot [\varphi]) \\
&= \lambda f. \text{ff}(f \cdot [\varphi] \cdot [\varphi]) + \text{ff}(f \cdot [\neg\varphi] \cdot [\varphi]) \\
&= \lambda f. \text{ff}(f \cdot [\varphi]) + 0 \\
&= \text{ff} \cdot [\varphi]
\end{aligned}$$

$$\begin{aligned}
\text{colow}(\text{ff}, \varphi) \cdot [\varphi] &= \lambda f. \text{colow}(\text{ff}, \varphi)(f \cdot [\varphi]) \\
&= \lambda f. \text{ff}(f \cdot [\varphi] \cdot [\varphi]) + \text{ff}(f \cdot [\neg\varphi] \cdot [\varphi]) + \text{async}_\varphi^{\text{Lin}(\text{ff})}(\infty) \\
&\geq \lambda f. \text{ff}(f \cdot [\varphi]) + 0 \\
&= \text{ff} \cdot [\varphi]
\end{aligned}$$

The proof with $\neg\varphi$ works analogously. $\square$

Remark 5.3: Note that $\text{colow}(\text{ff}, \varphi)$ is equal to infinity, if it is applied to an input distribution that gives non-zero probability to asynchronous states. However, it is still useful, because we compare hyperquantities point-wise i.e. universally quantify over the input distribution f . Hence, we also quantify over input distributions that disallow synchronous executions. For these input distributions, the infinite part of the colow -hyperquantity is zero, $\text{colow}(\text{ff}, \varphi)$ is not equal to infinity, and we can reason about the part of ff that we are interested in.

5.2.2 Lower-Bound Rule

The synchronous loop rules require the user to choose an invariant hyperquantity I . The soundness of the rules directly follow from the asynchronous loop rules with the invariant $\text{low}(I, \varphi)$ and $\text{colow}(I, \varphi)$.

We start by proving the lower-bound rule WHILE-SYNC (cf. Figure 4): We assume the premises of the rule:

$$I \cdot [\varphi] \leq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] \quad (1)$$

$$I \cdot [\neg\varphi] \leq \text{ff} \quad (2)$$

To apply the rule WHILE-ASYNC, we have to show that the two premises of the rule hold. The second premise follows directly from Eq. (2) and Lemma 5.2:

$$\begin{aligned}
\text{low}(I, \varphi) \cdot [\neg\varphi] &= I \cdot [\neg\varphi] && \text{(Lemma 5.2)} \\
&\leq \text{ff} && \text{(by Eq. (2))}
\end{aligned}$$

We can derive the first premise of the WHILE-ASYNC rule from the sum of Eq. (1) and Lemma 5.2 by applying the definition of low :

$$\begin{aligned} I \cdot [\varphi] + I \cdot [\neg\varphi] &\leq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] + \text{low}(I, \varphi) \cdot [\neg\varphi] \\ \text{low}(I, \varphi) &\leq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] + \text{low}(I, \varphi) \cdot [\neg\varphi] \\ \text{low}(I, \varphi) &\leq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] + \text{low}(I, \varphi) \cdot [\neg\varphi] + \mathcal{G}_{\text{async}} \\ \text{low}(I, \varphi) &\leq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (\text{low}(I, \varphi)) \end{aligned}$$

Here, $\mathcal{G}_{\text{async}}$ stands for the pre-hyperquantities of the if-conditional that correspond to asynchronous executions. As $\mathcal{G}_{\text{async}}$ is a hyperquantity, we know that it is non-negative, and we can add it on the right side of the inequality. If $\text{low}(I, \varphi)$ is linear, $\mathcal{G}_{\text{async}} = 0$, because there are no asynchronous executions and if $\text{low}(I, \varphi)$ is bilinear, then

$$\begin{aligned} \mathcal{G}_{\text{async}} &= \lambda f. \text{whp}_1 \llbracket C \rrbracket (\text{low}(I, \varphi)) (f \cdot [\varphi], f \cdot [\neg\varphi]) \\ &\quad + \text{whp}_2 \llbracket C \rrbracket (\text{low}(I, \varphi)) (f \cdot [\neg\varphi], f \cdot [\varphi]). \end{aligned}$$

In general, $\mathcal{G}_{\text{async}}$ is defined as:

$$\begin{aligned} \mathcal{G}_{\text{async}} &= \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (\text{low}(I, \varphi)) \\ &\quad - ([\varphi] \cdot \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) + [\neg\varphi] \cdot \text{low}(I, \varphi)). \end{aligned}$$

5.2.3 Upper-Bound Rule

Next, we can prove the upper-bound rule WHILE-SYNC-CO (cf. Figure 4). We can assume the premises of the rule:

$$I \cdot [\varphi] \geq \text{whp } \llbracket C \rrbracket (\text{low}(I, \varphi)) \cdot [\varphi] \quad (1)$$

$$I \cdot [\neg\varphi] \geq \text{ff} \cdot [\neg\varphi] \quad (2)$$

We prove the synchronous loop rule using the WHILE-ASYNC-CO rule with the invariant $\text{colow}(I, \varphi)$. To use the asynchronous rule, we have to show that the two premises of the rule hold. The second premise follows directly from Eq. (2) and the fact that hyperquantities are non-negative:

$$\begin{aligned} \text{colow}(I, \varphi) &= I \cdot [\varphi] + I \cdot [\neg\varphi] + \infty \cdot \text{async}_\varphi(\text{ff}) \\ &\geq I \cdot [\neg\varphi] \\ &\geq \text{ff} \cdot [\neg\varphi] \end{aligned} \quad (\text{by Eq. (2)})$$

For the second premise of the WHILE-ASYNC-CO rule, note that the asynchronous part of $\text{colow}(I, \varphi)$ is a trivial upper-bound for the asynchronous cases of the if-conditional rule $\mathcal{G}_{\text{async}}$ (cf. the definition of $\mathcal{G}_{\text{async}}$ in Section 5.2.2):

$$\text{async}_{\varphi}^{\text{Lin}(\mathcal{G}_{\text{async}})}(\infty) \geq \mathcal{G}_{\text{async}} \quad (3)$$

This inequality holds, because $\mathcal{G}_{\text{async}}$ is only non-zero, if the input distribution allows asynchronous executions. And for asynchronous executions, $\text{async}_{\varphi}(\infty)$ is infinite by definition. Additionally note that $\text{Lin}(\mathcal{G}_{\text{async}}) = \text{Lin}(I)$, because $\mathcal{G}_{\text{async}}$ is the result of deriving whp on I and whp does not change the linearity.

With this, we can derive the first premise of the asynchronous rule from the sum of Eq. (1), Lemma 5.2 and Eq. (3):

$$\begin{aligned} I \cdot [\varphi] + I \cdot [\neg\varphi] &\geq \text{whp } \llbracket C \rrbracket (\text{colow}(I, \varphi)) \cdot [\varphi] + \text{colow}(I, \varphi) \cdot [\neg\varphi] \\ + \text{async}_{\varphi}^{\text{Lin}(I)}(\infty) &+ \mathcal{G}_{\text{async}} \\ \text{colow}(I, \varphi) &\geq \text{whp } \llbracket C \rrbracket (\text{colow}(I, \varphi)) \cdot [\varphi] + \text{colow}(I, \varphi) \cdot [\neg\varphi] \\ &+ \text{async}_{\varphi}(\mathcal{G}_{\text{async}}) \\ \text{colow}(I, \varphi) &\geq \text{whp } \llbracket \text{if } (\varphi) \{C\} \text{ else } \{\} \rrbracket (\text{colow}(I, \varphi)) \end{aligned}$$

6 Implementation

We split the implementation into three parts. We first describe the input of the verifier to get an overview. Then we explain how whp is implemented and how inequalities of hyperquantities are converted into SMT formulas.

6.1 Verifier Input

The verifier should derive the weakest pre-hyperquantity given a program and a post-hyperquantity and compare it to a given pre-hyperquantity. To this end, the implementation accepts files in a language similar to Viper [13] and Caesar [14]: An input file includes a method containing a program and additionally a pre- and post-hyperquantity. Program variables are always defined globally and have one of three types: boolean, non-negative integers and $\mathbb{R}_{\geq 0}^{\infty}$. Figure 5 show an example input.

To use the loop rules, the verifier needs an invariant that is supplied with the annotation `@async_invariant` or `@sync_invariant` for the synchronous and

```

method random_assignment( $x$ : UInt)
  fprob
  pre 1/2
  post  $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot |s_1(x) - s_2(x)|$ 
{
  {
    * 1/2
     $x := 1$ 
  } + {
    * 1/2
     $x := 2$ 
  }
}

```

Figure 5: A simple program with pre- and post-hyperquantity.

asynchronous rule respectively. By default, the verifier checks lower-bounds. For upper-bounds, you have to replace **method** by **comethod**.

Additionally, you can set the modifiers **fprob** to tell the verifier to handle the f as a probability distribution, i.e. assume that $\sum_s f(s) = 1$. And the modifier **infty** can be used to indicate that ∞ should be supported. This is usually needed for proving upper-bounds. We will later elaborate on the reasons behind these modifiers.

6.2 Quantitative Weakest Hyper Pre

The whp rules presented in Section 4 and Section 5 directly give an algorithm for deriving the weakest pre-hyperquantity. Given a post-hyperquantity and a program, the verifier splits the hyperquantity into pieces of equal linearity (cf. Section 4.3) and then derives whp on the pieces by applying the whp rules matching the program. The loop rules additionally create side conditions to check the premises of the rules.

The whp transformer requires three basic operations on hyperquantities:

- Adding up two hyperquantities for the linear nondeterministic-choice rule.
- Multiplying the input distribution by an expression for the weighting rule. We need this for the general whp rule, but also for the whp_n rule where we only want to multiply a specific input distribution by an expression.

```

whp( $C$ ,  $ff$ ) {
  match  $C$  with
  |  $C_1 ; C_2$        $\Rightarrow$  whp( $C_1$ , whp( $C_2$ ,  $ff$ ))
  |  $* e$             $\Rightarrow$  multiplyWithInputDistribution( $ff$ ,  $e$ )
  |  $x := e$          $\Rightarrow$  substituteVariable( $ff$ ,  $x$ ,  $e$ )
  |  $\{C_1\} + \{C_2\}$   $\Rightarrow$  { for  $i=1..Lin(ff)$ :
                         $ff := whpn(i, C, ff)$ 
                         $ff$  }
  | while ( $e$ )  $\{C\}$   $\Rightarrow$  // add side conditions...
                        //  $I$ , low( $e, I$ ) or colow( $e, I$ )
}

whpn( $n$ ,  $C$ ,  $ff$ ) {
  match  $C$  with
  |  $C_1 ; C_2$        $\Rightarrow$  whpn( $n$ ,  $C_1$ , whpn( $n$ ,  $C_2$ ,  $ff$ ))
  |  $* e$             $\Rightarrow$  multiplyWithInputDistributionN( $n, ff$ ,  $e$ )
  |  $x := e$          $\Rightarrow$  substituteVariableN( $n$ ,  $ff$ ,  $x$ ,  $e$ )
  |  $\{C_1\} + \{C_2\}$   $\Rightarrow$  add(whpn( $n$ ,  $C_1$ ,  $ff$ ), whpn( $n$ ,  $C_2$ ,  $ff$ ))
  | while ( $e$ )  $\{C\}$   $\Rightarrow$  // not supported
}

```

Figure 6: The implementation of whp and whp_{*n*} using basic hyperquantity operations in pseudo code.

- Substituting a variable by an expression. Again, we need this in two variants: The general one substituting all occurrences of the variable and one for whp_{*n*} substituting only the variables corresponding to the *n*-th sum quantifier.

Using these basic operations, the algorithm looks as in Figure 6.

We omit the details of the while rule here. They also only use the basic operations and the low- and colow-functions. The low- and colow-functions are implemented as they are defined, but instead of implementing $ff \cdot [\varphi] + ff \cdot [\neg\varphi]$, i.e. adding the parts where φ holds in all states and $\neg\varphi$ holds in all states, we can combine the two Iverson brackets by requiring that the value of φ is equal for all states. This makes the resulting hyperquantities syntactically smaller.

What we also see in the algorithm is that while loops inside nondeterministic choices (or if-conditionals) are not supported. In general, we could reason about them, but this would require the user to give multiple invariants: For a

non-deterministic choice, we apply whp_n k -times where k is the linearity of the hyperquantity. Hence, we also apply the while rule k times and need a different invariant every time. While loops inside if-conditionals are rather rare, so this is not a big restriction. For the future, note that the invariants of all derivation of whp_n are usually similar. We could try to find a way of using one invariant to generate invariants for all derivations of whp_n .

6.2.1 Alternative Approaches

We additionally implemented two alternative approaches. Firstly, we can use the parallel nondeterministic choice rules presented in Section 4.2.2. They can be implemented similarly as above as they also only use the basic operations described above with minor modifications.

Secondly, note that the k -LINEAR rule (cf. Section 4.2.1) can actually be applied to all programs: We can always treat a multilinear hyperquantity as linear by fixing all but one input distribution. Hence, deriving whp is then implemented as deriving whp_n k -times. However, this has the drawback that loops cannot be supported for the same reason as above that loops are not supported inside nondeterministic choices.

6.3 Conversion into SMT formulas

Given a bound on the pre-hyperquantity and the derived pre-hyperquantity, it remains to show entailment. So, we have to check an inequality $\text{ff}_1 \leq \text{ff}_2$ for $\text{ff}_1, \text{ff}_2 \in \text{HQ}$. The inequality is defined point-wise, so this is equal to showing $\forall f \in \mathbb{A}. \text{ff}_1(f) \leq \text{ff}_2(f)$. We check this by encoding it into an SMT formula. As it is common in verification, we do so by encoding f as an uninterpreted function and check satisfiability of the negated inequality $\neg(\text{ff}_1(f) \leq \text{ff}_2(f))$. If the formula is unsatisfiable, we know that the inequality holds and if the formula is satisfiable, we get a concrete counterexample for an f that violates the inequality.

6.3.1 Encoding Hyperquantities

When we apply a hyperquantity, we get value in $\mathbb{R}_{\geq 0}^\infty$. Hence, encoding the inequality $\text{ff}_1(f) \leq \text{ff}_2(f)$ mostly boils down to comparing two real values. As SMT solvers have native support for reals and the language of hyperquantities

is mostly based on fundamental arithmetic operations, we can trivially convert most parts. The only exception is the sum quantifier and infinity.

To handle infinity, we encode extended unsigned reals x as tuples of a real number and a boolean as in the Caesar verifier [14]. If $x = \infty$, the boolean is set to true and if $x \neq \infty$, we can directly encode x into the real number. Operators are directly encoded using these tuple. For example, multiplication is defined as follows:

$$(r_1, inf_1) \cdot (r_2, inf_2) = \begin{cases} (0, \text{false}) & \text{if } r_1 = 0 \text{ or } r_2 = 0 \\ (0, \text{true}) & \text{if } (inf_1 \vee inf_2) \wedge r_1 \neq 0 \wedge r_2 \neq 0 \\ (r_1 \cdot r_2, \text{false}) & \text{otherwise.} \end{cases}$$

Note that this can impose a big overhead for the SMT solver, although infinity is not needed in many cases. To account for that, infinity has to be manually enabled. Otherwise, the verifier encodes extended unsigned reals directly as reals ignoring infinity.

Remark 6.1 (Negative numbers): Note that hyperquantities are defined as non-negative, but the syntax also allows negative numbers and subtraction. We add additional constraints that all variables appearing in the hyperquantities should be non-negative. However, there might still appear negative numbers through constants or subtractions. We decided to ignore this problem for now and figure out a solution in the future. This also means that you have to be careful when using unary minus and subtraction as the verifier might give unsound results.

There are two solutions that could be implemented in the future: The first one is to disallow unary minus and subtraction. However, there are many programs and hyperquantities where this feature is important and does not result in negative numbers (e.g. showing the expected absolute difference of variables). The other solution is to make sure that subtraction never yields a negative number. This is what the Caesar verifier does [14], but this way real arithmetic does not always behave as expected (e.g. $5 + (1 - 2) \neq (5 + 1) - 2$).

6.3.2 Encoding the Sum Quantifier

The sum quantifier sums over all variables that are mentioned in the hyperquantity. So, we can express a state s as a tuple of program variables, e.g. $(s_x, s_y, \dots)$, and the sum quantifier sums over all values that these variables can have. We made the sum finite by restricting f to have finite support, but

we universally quantify over f , so we do not know which states we should sum over. Hence, the quantifier still sums over infinitely many values and cannot be trivially encoded into an SMT formula. Instead, we simplify the inequality. As we are building a verifier, a verifying result should imply that the original inequality also holds. So we strengthen the inequality by using the fact that $\sum s.\mathit{ff}_1 \leq \sum s.\mathit{ff}_2$ if $\forall s.\mathit{ff}_1 \leq \mathit{ff}_2$.

This also has a different advantage: As said above, we check satisfiability of the negation of the formula. So the for all quantifier becomes an existential quantifier and a falsified verification gives us concrete states for which the verification fails.

6.3.3 Normal Form

We can only use the strengthening from Section 6.3.2, if the hyperquantities on both sides start with a sum quantifier. This is not a big restriction as we will see. In fact, we can convert a hyperquantity into a normal form that has all quantifiers in leading position.

Lemma 6.2 (Normal Form): *Let $\mathit{ff} \in \text{HQ}$ be k -linear or constant. There exists a quantifier-free $\mathit{ff}' \in \text{HQ}$ such that $\mathit{ff} = \sum_{s_1} \dots \sum_{s_k} \mathit{ff}'$.*

For general $\mathit{ff} \in \text{HQ}$, there exist quantifier-free $\mathit{ff}_1, \dots, \mathit{ff}_n$ such that

$$\mathit{ff} = \left(\sum_{s_{1,1}} \dots \sum_{s_{1,k_1}} \mathit{ff}_1 \right) + \dots + \left(\sum_{s_{n,1}} \dots \sum_{s_{n,k_n}} \mathit{ff}_n \right).$$

Proof Sketch: The normal-form for k linear or constant hyperquantities, can be shown similar as Lemma 3.2 by showing that we can take the sums out of the hyperquantity. Proving this is very technical, so we omit it here.

The general normal form follows from the fact that we can split a hyperquantity into constant and k -linear pieces ff_i (cf. Theorem 3.3). $\square$

Note that this normal form is usually not unique, and we discuss what consequences this has in Section 6.3.4.

When we convert both sides of the verification goal into the normal form, the goal looks like this:

$$\begin{aligned} & \left(\sum_{s_{l,1,1}} \cdots \sum_{s_{l,1,k_1}} ff_{l,1} \right) + \cdots + \left(\sum_{s_{l,n_l,1}} \cdots \sum_{s_{l,n_l,k_{n_l}}} ff_{l,n_l} \right) \\ & \leq \left(\sum_{s_{r,1,1}} \cdots \sum_{s_{r,1,k_1}} ff_{r,1} \right) + \cdots + \left(\sum_{s_{r,n_r,1}} \cdots \sum_{s_{r,n_r,k_{n_r}}} ff_{r,n_r} \right) \end{aligned}$$

Note that we additionally annotated all variables with l on the left side and r on the right side.

In order to apply the strengthening to get rid of the sum quantifiers, both sides of the inequality should have the sum quantifiers in leading position. There are two ways to achieve this. The first one is to assume that the input distribution f is a real probability distribution, i.e. it satisfies the equality $\sum s.f(s) = 1$. We can indicate this by a flag in the input of the verifier. In this case, we can increase the linearity of a k -linear hyperquantity by multiplying it by $\sum s.f(s)$. When all pieces ff_i in the normal form have the same linearity, we can then combine them by unifying the sum quantifiers with the associativity rule $\sum s.ff_i + \sum s.ff_j = \sum s.(ff_i + ff_j)$. It should be noted that this transformation is unique up to changing the order of sum quantifiers and thus identifying different states with each other. We introduce our approach to dealing with non-uniqueness in Section 6.3.4.

Example 6.3: We start with the following hyperquantity in normal form:

$$ff = \sum_{s_{1,1}} \sum_{s_{1,2}} f(s_{1,1}) \cdot f(s_{1,1}) \cdot (s_{1,1}(x) + s_{1,2}(x)) + \sum_{s_{2,1}} f(s_{2,1}) \cdot s_{1,1}(y).$$

It has one 2-linear piece and one 1-linear piece. We increase the linearity of the second piece by multiplying it by $\sum_{s_{2,2}} f(s_{2,2})$ and bringing it into normal form again:

$$ff = \sum_{s_{1,1}} \sum_{s_{1,2}} f(s_{1,1}) \cdot f(s_{1,1}) \cdot (s_{1,1}(x) + s_{1,2}(x)) + \sum_{s_{2,1}} \sum_{s_{2,2}} f(s_{2,1}) \cdot f(s_{2,2}) \cdot s_{1,1}(y).$$

Now, we can combine the pieces by identifying $s_{1,1}$ with $s_{2,1}$ and $s_{1,2}$ with $s_{2,2}$:

$$ff = \sum_{s_{1,1}} \sum_{s_{1,2}} f(s_{1,1}) \cdot f(s_{1,1}) \cdot (s_{1,1}(x) + s_{1,2}(x) + s_{1,1}(y)).$$

This is not the only possible way to combine them. We can also identify $s_{1,1}$ with $s_{2,2}$ and $s_{1,2}$ with $s_{2,1}$:

$$ff = \sum_{s_{1,1}} \sum_{s_{1,2}} f(s_{1,1}) \cdot f(s_{1,1}) \cdot (s_{1,1}(x) + s_{1,2}(x) + s_{1,2}(y)).$$

If f is not a real probability distribution, i.e. $\sum s.f(s) = 1$ does not hold, we follow a different approach: For pieces of same linearity, we can combine them as above. For the remaining pieces, we compare the hyperquantities piece-wise, i.e. we use the fact that:

$$\begin{aligned} & \forall f. (\text{ff}_{l,1}(f) \leq \text{ff}_{r,1}(f)) \wedge (\text{ff}_{l,2} \leq \text{ff}_{r,2}) \\ \implies & \forall f. \text{ff}_{l,1}(f) + \text{ff}_{l,2}(f) \leq \text{ff}_{r,1}(f) + \text{ff}_{r,2}(f). \end{aligned}$$

We compare pieces of same linearity to be able to use the strengthening from Section 6.3.2. And abort the verification, if we do not find hyperquantities whose linearity matches on both sides of the inequality. We believe that this is not a restriction and an inequality $\forall f. \text{ff}_l(f) \leq \text{ff}_r(f)$ does not hold, if ff_l and ff_r do not have the same linearity. We leave open to investigate, how big the restriction is.

Both ways change the verification goal such that the sum quantifiers are in leading position on both sides of the inequality. Hence, we can use the strengthening $\sum s. \text{ff}_l \leq \sum s. \text{ff}_r$ if $\forall s. \text{ff}_l \leq \text{ff}_r$ as described above. However, we can apply the strengthening in different ways, because we can change the order of the sum quantifiers and thus identify states on the left and right side of the inequality to each other in different ways.

6.3.4 Non-Uniqueness of the Encoding

Some transformations we presented can be done in different ways by changing the order of sum quantifiers. If we use the exact definition of the sums, this non-uniqueness does not matter. However, we replace the sum quantifiers by $\forall$ for all quantifiers and this can influence the verification result:

Example 6.4: Consider the following inequality:

$$\sum_{s_{l,1}} \sum_{s_{l,2}} f(s_{l,1}) \cdot f(s_{l,2}) \cdot s_{l,1}(x) \leq \sum_{s_{r,1}} \sum_{s_{r,2}} f(s_{r,1}) \cdot f(s_{r,1}) \cdot s_{r,2}(x).$$

It can be strengthened in two ways:

$$\begin{aligned} & \forall s_1, s_2. f(s_1) \cdot f(s_2) \cdot s_1(x) \leq f(s_1) \cdot f(s_2) \cdot s_1(x) \\ & \forall s_1, s_2. f(s_1) \cdot f(s_2) \cdot s_1(x) \leq f(s_1) \cdot f(s_2) \cdot s_2(x) \end{aligned}$$

The first inequality trivially holds, because both sides are equal. The second one does not hold in general.

As the non-uniqueness matters, we check all possibilities of identifying states with each other. If one possibility verifies, we know that the inequality holds.

Hence, the input for the SMT solver is a big disjunction over all possibilities. The number of possibilities is exponentially dependent on the number of sum quantifiers and thus dependent on the number of pieces and the linearity of the pieces. To deal with this exponential overhead, we use two measures in the implementation.

Firstly, all formulas in the disjunction are equal up to the name of the states. Hence, we generate the encoding once for one possibility and just replace the state names to get the encoding for the other possibilities. This saves time for the SMT formula generation. We also experimented with encoding the possibilities directly into SMT using higher-order functions. This way, the input size for the SMT solver would be of polynomial size again and only the SMT solver would have to deal with the exponential overhead. However, the SMT solver performed worse with this encoding than with explicitly giving it the disjunction.

Secondly, we do not consider all possibilities. If the user explicitly writes a hyperquantity in two pieces (e.g. $\sum s.\mathit{ff}_1 + \sum s.\mathit{ff}_2$), we consider all possible ways of combining the sum quantifiers. If the user gives a hyperquantity already in one piece (e.g. $\sum s.(\mathit{ff}_1 + \mathit{ff}_2)$), we do not split it up. So, the user can choose which possibilities are considered.

6.3.5 Alternative Approaches

Instead of encoding the sum quantifier as in Section 6.3.2, we can also treat the sum as an uninterpreted function. Additionally, we can formulate axioms describing valid transformations of sums, e.g. taking factors inside a sum: $\mathit{ff}_1 \cdot \sum s.\mathit{ff}_2 = \sum s.(\mathit{ff}_1 \cdot \mathit{ff}_2)$. The strengthening described in Section 6.3.2 is then also an axiom. The advantage is that we do not have to manually transform hyperquantities in normal-form and consider all different possibilities of changing the order of sums.

This approach has two drawbacks. Firstly, the SMT solver performs worse with this encoding, because it is hard to guide it which axioms to use when. Our approach basically solves this by doing all transformations before giving the inequality to the SMT solver.

The second problem is that index-shift axioms cannot be easily encoded into an SMT formula. Consider e.g. the axiom that we can rename a state,

i.e. $\sum s.\text{ff} = \sum s'.\text{ff}[s \mapsto s']$. The axiom uses a syntactic replacement of variables, and it is unclear how to do this using SMT. One option is to encode hyperquantities as functions taking state variables. This way, we can easily exchange them, but the SMT solver does perform poorly with this encoding.

One solution to both problems would be to use e-graph [18] to implement transformations including the ones that involve index-shifts of the sums. Once these e-graphs yield an encoding without sum quantifiers, it could be given to an SMT solver. The generation of the e-graphs could be guided towards doing useful transformations. However, it still seems to be more efficient to explicitly implement the transformations at the moment.

7 Evaluation

7.1 Case Studies

The verifier supports many interesting programs and hyperquantities. We give two examples here to showcase what the verifier can be used for.

7.1.1 Quantitative k -Safety-Properties

Firstly, assume we have a program that uses operations that impose small errors, and we want to prove that a qualitative k -safety hyperproperty [15, Figure 3] holds with a certain probability. One example is the power function in Figure 7 that computes the power x^y by iteratively multiplying x to the result res y -times. However, the loop body has a small error and additionally adds 1 to the result with probability $\frac{1}{2}$. In this example, the error might make no sense, but we can think of real examples where we use operations that have a small error and that can be modelled this way.

We want to prove that this power function is monotonic with regard to x and fixed y . We can encode the probability that the power function is monotonic into the post-hyperquantity. The error-free power function is monotonic, so the erroneous one is monotonic, if we always choose the left side of the nondeterministic choice. This happens with probability $\frac{1}{2}$, so the function is monotonic with probability 0.5^{2y} . Hence, we encode this overall probability into the pre-hyperquantity.

Additionally, we have to choose an invariant for the while-loop. Because y is fixed, we only have to consider synchronous executions and can use the

```

method power( $x$ : EReal,  $y$ : UInt,  $res$ : EReal,  $i$ : UInt)
  fprob
  pre  $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot s_1(\exp(0.5, y)) \cdot s_2(\exp(0.5, y))$ 
       $\cdot [s_1(y) = s_2(y)]$ 
  post  $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot [(s_1(x) \leq s_2(x) \wedge s_1(y) = s_2(y))$ 
       $\implies s_1(res) \leq s_2(res)]$ 
{
   $res := 1$ 
   $i := 0$ 

  @sync_invariant ( $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2)$ 
       $\cdot s_1(\exp(0.5, y - i)) \cdot s_2(\exp(0.5, y - i))$ 
       $\cdot [((s_1(x) \leq s_2(x)) \vee s_1(res) \leq s_2(res))$ 
       $\implies (s_1(y) = s_2(y) \wedge s_1(i) = s_2(i) \wedge s_1(i) \leq s_1(y))]$ )

  while ( $i < y$ ) {
    {
       $* 1/2$ 
       $res := res \cdot x$ 
    } + {
       $* 1/2$ 
       $res := res \cdot x + 1$ 
    }
     $i := i + 1$ 
  }
}

```

Figure 7: Verifier input to prove that the erroneous power function is monotonic with probability at least 0.5^{2y} .

synchronous rule. Most parts of the invariant are similar to invariants for non-hyper verifiers: We formulate the exponential in terms of the loop variable i and add additional conditions in Iverson brackets. However, we additionally have to show that the loop indeed always behaves synchronously, i.e. that the value of the loop guard is equal for all executions. To prove this, we have to add the information to the invariant that i and y are equal across all executions.

7.1.2 Expected Differences

The second example is the program in Figure 8 that represents a game that we play for 6 rounds, and we either win and our winnings w increase or we loose. The program is inspired by Wang et al. [16, Figure 4]. The property we want to prove is that the expected absolute difference of the winnings for different

```

comethod game( $n$ : UInt,  $w$ : UInt,  $i$ : UInt)
  fprob infty
  pre 3
  post  $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot |s_1(w) - s_2(w)|$ 
{
   $n := 6$ 
   $i := 0$ 
   $w := 0$ 

  @sync_invariant ( $\sum s_1.f(s_1) \cdot \sum s_2.f(s_2) \cdot ($ 
     $(|s_1(w) - s_2(w)| + s_1((n - i) \cdot 0.5)) \cdot [s_1(i \leq n)]$ 
     $+ \infty \cdot [s_1(i > n) \vee s_2(i > n) \vee s_1(i) \neq s_2(i) \vee s_1(n) \neq s_2(n)])$ )
  while ( $i < n$ ) {
    {
      * 1/2
       $w := w + 1$ 
    } + {
      * 1/2
    }
     $i := i + 1$ 
  }
}

```

Figure 8: Verifier input to prove that the expected absolute difference of the winning of a game is upper-bounded by 3.

executions of the game is upper-bounded by 3. Hence, our post-hyperquantity is the expected absolute difference and the pre-hyperquantity is 3.

We prove an upper-bound, so we use **comethod** and the **infty** modifier to enable infinity. We also have to take the upper-bound into account when formulating the loop-invariant: For the lower-bound rule, it was enough to characterize synchronous loop states with an Iverson bracket, because all asynchronous states automatically get the trivial zero bound then. With the upper-bound rule, the asynchronous states get the trivial bound ∞ . So, to convince the verifier that the loop behaves synchronously, we additionally need a condition that characterizes asynchronous states and gives them the value ∞ .

Note that the loop invariant contains some redundant conditions (e.g. $[s_1(i \leq n)]$ and $\infty \cdot [s_1(i > n)]$) and the program could be simplified by replacing n by 6 everywhere. We did not do that, because the SMT solver struggles with the simplifications and does not terminate in a few seconds. We leave open to investigate this problem and find ways to guide the SMT solver.

Case Study	Linear	Time of the Verification Steps (in s)			
		whp	SMT Gen	SMT Check	Total
Lower-Bounds, Loop-Free:					
Monotonicity	2	<0.1	0.2	3.5	3.7
Deterministic (complex)	2	0.2	14.5	7.4	22.1
Transitivity	3	<0.1	0.3	0.8	1.1
Associativity	4	<0.1	0.2	0.2	0.4
Lower-Bounds with Loops:					
Monotonicity-Power-Sync	2	<0.1	0.2	0.5	0.7
Monotonicity-Power-Async	2	<0.1	0.2	1.8	2.0
Upper-Bounds with Loops:					
Expected-Winning-Sync	2	<0.1	0.3	2.7	3.0
Expected-Winning-Async	2	<0.1	0.3	7.2	7.5

Figure 9: The running-time of the verification steps (whp derivation, SMT formula generation and SMT formula satisfiability check) for a selection of different case studies. The times were measured on AMD Ryzen 5500U and Z3 CLI 4.16.0.

7.2 Performance

We evaluate the performance of the verifier with sample programs and hyper-quantities of different linearity (cf. Figure 9). We can measure the verification time in three parts: whp derivation, conversion into an SMT formula and satisfiability check of the formula. The times were only measured once, so they should not be understood as an exact benchmark, but as an orientation on how long the verification takes.

The major time is spent by the SMT solver to check for satisfiability. Note that there is room for improvements as we did not configure the SMT solver. E.g. specifying the logics could bring a speed-up here.

Especially for the upper-bound case studies the running-time for the satisfiability check is high. This might come from the fact that the upper-bound case studies require reasoning about infinity and this means that $\mathbb{R}_{\geq 0}^{\infty}$ has to be encoded into the SMT solver using tuples (cf. Section 6). This imposes many additional checks for the SMT solver.

We can also see that the running-time of the whp derivation and SMT generation mostly depends on the complexity of the program. The “Deterministic” case study contains more nested if-conditionals than “Monotonicity” and needs

much more time for these steps. Also, the loop case studies have complex hyperquantities, but their running-time for the first two steps is similar to “Monotonicity”. Hence, the time seems to depend more on the complexity of the program rather the hyperquantity. This makes sense as every if-conditional at least doubles the size of the hyperquantity (cf. Table 1).

Lastly, the benchmark only shows the time of successful verifications. Unsuccessful verifications often take more time and might not terminate. Especially when using uninterpreted functions with axioms such as for the exponential function, the verifier typically does not terminate, if the given pre-hyperquantity is not a valid bound or the loop-invariants do not hold.

8 Conclusion

We built a verifier for a class of hyperquantities that allows summing over states and relating program variables of multiple executions. To this end, we developed and implemented simple whp rules for multilinear hyperquantities for assignments and nondeterministic choices and invariant-based rules for while-loops. Additionally, we introduced a method to convert an inequality of hyperquantities into an SMT formula that can be automatically checked by an SMT solver. With this, the verifier is able to derive a bound on the pre-hyperquantity of a program and check that it is bounded by a given pre-hyperquantity.

8.1 Limitations

The verifier has some limitations. Firstly, it only supports a specific class of hyperquantities that only contains sum quantifiers and are defined with the semiring $\mathbb{R}_{\geq 0}^{\infty}$. This makes sense for reasoning about expected values or linear combinations of expected values, but some use-cases require `min` and `max` quantifiers and reasoning about other semirings that e.g. allow negative numbers [17]. However, we expect that most of our results can be generalized to arbitrary semirings.

Furthermore, the language of programs is restricted to basic constructs and does not support the use of arrays or other datastructures. In the future, we could add them using uninterpreted functions and axioms as Caesar is doing it [14]. Also, local variables and method calls are missing.

Lastly, the SMT encoding of hyperquantity inequalities is not exact, i.e. a negative verification result does not imply that the inequality does not hold. However, this is often the case for verifiers and the counterexamples can still be helpful in finding the error or convincing the verifier that there is no error.

8.2 Future Work

We proposed many new whp rules and other theoretic background for the verifier that is not proven formally. Formalizing the theory would give a sound basis for the verifier, and would allow identifying the requirements to generalize the theory, e.g. to arbitrary semirings.

Furthermore, there are still some unsolved problems that appeared during the implementation: Negative numbers are allowed by the implementation at the moment, although they can make the verification result unsound. Also, while-loops nested in if-conditionals cannot be supported nicely at the moment.

We presented some whp rules for loops, but they might not be useful in all cases and the lower-bound rules require termination of the loop. Hence, we need to develop more general loop rules to account for nonterminating loops.

The SMT encoding of inequalities of hyperquantities is not exact. This was no problem for all case studies that we looked at, but we might need a more precise encoding in the future.

Also, the SMT solver performs poorly on some simple case studies, while it is fast on some complex ones. Additionally, different solvers and solver configurations perform very differently on the case studies. Hence, we should investigate which configuration best fits the verifier and how we can guide it, e.g. by listing the exact SMT theories that it should use.

Lastly, our verifier theoretically allows proving some form of sensitivity of programs. However, to get tight results, we need a feature like couplings similar as in Aguirre et al. [1]. It seems like a rather easy extension that would allow the verifier to be used for an interesting class of hyperquantities.

Data Availability Statement

The implementation of the verifier and the case studies are publicly available on Github: <https://github.com/Sarius121/whp-automation>.

References

- [1] Alejandro Aguirre et al. “A pre-expectation calculus for probabilistic sensitivity”. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434333. URL: <https://doi.org/10.1145/3434333>.
- [2] Gilles Barthe et al. “An Assertion-Based Program Logic for Probabilistic Programs”. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 117–144. ISBN: 978-3-319-89884-1.
- [3] Christopher Cohnen. “Usability and Applications of Quantitative Weakest Hyper Pre”. Bachelor’s Thesis. Saarbrücken, Germany: Universität des Saarlandes, 2024.
- [4] Thibault Dardinier, Anqi Li, and Peter Müller. “Hypra: A Deductive Program Verifier for Hyper Hoare Logic”. In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689756. URL: <https://doi.org/10.1145/3689756>.
- [5] Thibault Dardinier and Peter Müller. *Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version)*. 2023. arXiv: 2301.10037 [cs.LO].
- [6] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: <https://doi.org/10.1145/360933.360975>.
- [7] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. “Model Checking Quantitative Hyperproperties”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 144–163. ISBN: 978-3-319-96145-3.
- [8] J. A. Goguen and J. Meseguer. “Security Policies and Security Models”. In: *1982 IEEE Symposium on Security and Privacy*. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.
- [9] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: <https://doi.org/10.1145/363235.363259>.
- [10] Benjamin Lucien Kaminski et al. “Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms”. In: *J. ACM* 65.5 (Aug. 2018). ISSN: 0004-5411. DOI: 10.1145/3208102. URL: <https://doi.org/10.1145/3208102>.

- [11] Donald E. Knuth. *Two notes on notation*. 1992. arXiv: math/9205211 [math.HO].
- [12] Dexter Kozen. “A probabilistic PDL”. In: *Journal of Computer and System Sciences* 30.2 (1985), pp. 162–178. ISSN: 0022-0000. DOI: [https://doi.org/10.1016/0022-0000\(85\)90012-1](https://doi.org/10.1016/0022-0000(85)90012-1). URL: <https://www.sciencedirect.com/science/article/pii/0022000085900121>.
- [13] P. Müller, M. Schwerhoff, and A. J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62. URL: https://doi.org/10.1007/978-3-662-49122-5_2.
- [14] Philipp Schröder et al. “A Deductive Verification Infrastructure for Probabilistic Programs”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 2052–2082. DOI: 10.1145/3622870. URL: <https://doi.org/10.1145/3622870>.
- [15] Marcelo Sousa and Isil Dillig. “Cartesian hoare logic for verifying k-safety properties”. In: *SIGPLAN Not.* 51.6 (June 2016), pp. 57–69. ISSN: 0362-1340. DOI: 10.1145/2980983.2908092. URL: <https://doi.org/10.1145/2980983.2908092>.
- [16] Peixin Wang et al. “Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371093. URL: <https://doi.org/10.1145/3371093>.
- [17] Linpeng Zhang et al. *Quantitative Weakest Hyper Pre: Unifying Correctness and Incorrectness Hyperproperties via Predicate Transformers*. 2024. arXiv: 2404.05097 [cs.LO].
- [18] Yihong Zhang et al. “Better Together: Unifying Datalog and Equality Saturation”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591239. URL: <https://doi.org/10.1145/3591239>.