Automating Quantitative Weakest Hyper Pre

Project Description for a Practical Work

Christopher Cohnen

Co-Supervised by Dr. Thibault Dardinier & Hongyi Ling

November 6, 2025

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 [7] 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 [4] 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 [3]. 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 [6].

On the other hand, properties have been extended to quantities [10]. 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:

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

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:

$$f(\sigma) \coloneqq \sigma(c).$$

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 it is 5 or not. Hence, we can express some properties more directly using quantities and also enable reasoning about expected values [8].

Zhang et al. [12] 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 [5, 12, 1]. For this, Zhang et al. [12] introduced a weakest-pre transformer that can reason about such properties.

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 [2] and Caesar for quantities [11]. For hyperquantities and the quantitative weakest hyper-pre transformer from Zhang et al. [12], no such automation exists yet. Still, the rules of the weakest-pre transformer are syntax-directed and mechanical for some classes of programs and hyperquantities.Hence, automation of the derivation should be possible in these cases and make the transformer more usable in practice. The main goal of this project is to automate the quantitative weakest hyper pre for relevant classes of programs and hyperquantities where the automation yields useful results with few interactions.

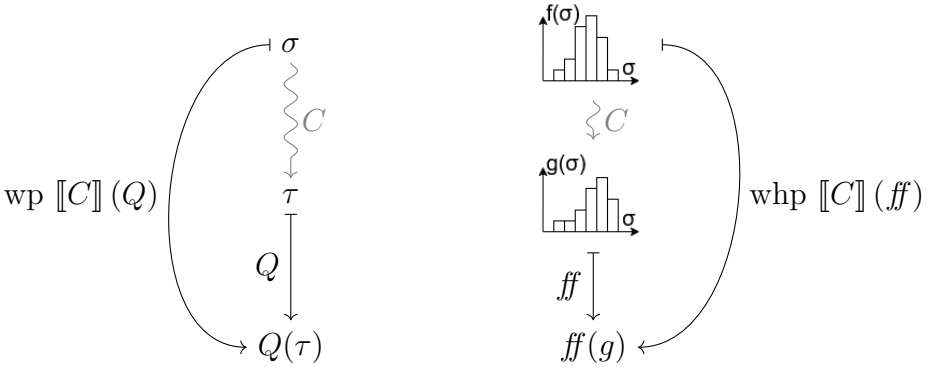

Figure 1: Intuition about wp and whp and the connection between them.

2 Quantitative Weakest Hyper Pre

2.1 Definition

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 1). 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 $\Sigma \to \{\text{true}, \text{false}\}$ (where Σ is the set of program states), hyperquantities are defined as functions $(\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to \mathbb{R}^{\infty}_{\geq 0}$, i.e. functions from distributions over states to real values. 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 $\mathit{gg} = \text{whp}\,[\![C]\!]\,(\mathit{ff})$. 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 1).

2.2 Example

A simple example of a hyperquantity is the expected value of a program variable. Consider e.g. the program in Figure 2 that can be seen as a password-

```
function password_cracker(p) {
    // λf. (∑_σ f(σ) · [p = 1234](σ)) + (∑_σ 0 · [p ≠ 1234](σ))
    =// λf. ∑_σ f(σ) · [p = 1234](σ)
    if (p = 1234) {
        // λf. ∑_σ 1 · f(σ)
        s = 1
    } else {
        // λf. ∑_σ 0 · f(σ) = λf.0
        s = 0
    }
    // 𝔼[s] = λf. ∑_σ σ(s) · f(σ)
}
```

Figure 2: Derivation of whp on a simple program and the expected value of s as the post-hyperquantity.

cracker program that only succeeds (i.e. $s = 1$) if the password is equal to 1234. We are interested in the expected value of s (which is basically the success probability) to evaluate how good the password-cracker is. The green lines in the program show the derivation of whp on the post-hyperquantity $\mathbb{E}[s]$ from the bottom to the top: We start by applying the assignment-rule in each case that gives us the post-hyperquantity with $\sigma(s)$ substituted by the expression of the assignment, i.e. 0 and 1 respectively. Next, we can apply the if-rule: It comes down to adding up the hyperquantities of the two cases and additionally multiplying the if-condition using Iverson brackets[1]. So, the derivation only needs three basic operations here and the automation of whp is trivial in this example.

2.3 Simplifications in Computing of whp

Unfortunately, the derivation of whp is not always that easy: For the generic loop-rule, one needs to find a fixpoint of a function. Additionally, the rule for if-conditionals and assignments is not as easy as above in the general case. However, there are specific classes of programs and hyperquantities that allow simplifications. One of these classes are linear hyperquantities like the expected value above that allows easier rules for if-conditionals [12, Section

[1]For a predicate φ, the Iverson bracket $[\varphi](\sigma)$ evaluates to 1 if σ satisfies φ and to 0 otherwise [9].

6.3]. Another class are programs that only contain invertible assignments, i.e. assignments where the value of the variable before the assignment is uniquely given by the state after the assignment or known from somewhere else [1, Section III.2]. This is also the case above, because s has a default value before the assignments, and we can thus use a simple assignment rule. So there are some combinations of programs and hyperquantities, where the automation seems feasible.

2.4 Interpretation of the Result of whp

The result of the classical wp is easy to interpret: When we encode a correctness property into a postcondition Q, wp $[\![C]\!]\,(Q)$ 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 wp $[\![C]\!]\,(Q)$, i.e. the Hoare triple $\{P\}\ C\ \{Q\}$ holds, then we know that C satisfies the property Q for the initial states that satisfy P.

For whp, the interpretation is more involved and there are three interesting approaches to it:

1. **Simplifier:** In the classical setting, the resulting pre-condition is sometimes already the wanted result, because it shows how the post-condition depends on the initial variables. Similarly, the pre-hyperquantity itself might already be interesting to interpret. For example, when reasoning about the expected runtime of a sorting algorithm, it is interesting to see on what initial values and distributions it depends on [1, Section IV.1.2]. However, we need the hyperquantity to be simplified enough so that humans can interpret it.

2. **Estimator:** When we have additionally an initial distribution over states, we can evaluate the pre-hyperquantity on that distribution. This corresponds to applying a precondition to a state in the classical setting. Consider the example above with the password-cracker, where we might want to investigate the expected probability of success for different password distributions. We can directly input the distributions into the pre-hyperquantity and get the expected probability of success. We call this postprocessing of the whp-result *Estimation*.

3. **Verifier:** When reasoning about differential privacy, the difference of final values between two executions should be bound by the difference of

initial values. Hence, we want to compare a pre-hyperquantity gg with the derived pre-hyperquantity. With less-or-equal, this is the quantitative hyper version of Hoare triples, where the less-or-equal corresponds to the implication. As for the classical setting, we call this entailment checking *Verification*.

3 Goals

Our goal is to automate the computation of whp as much as possible. To this end, we split the task into several subgoals.

3.1 Core Goals

1. **Identify relevant programs and hyperquantities for which the automation of the** whp **transformer is feasible.** These classes should be as big as possible to be able to prove interesting properties of non-trivial programs. At the same time they should be small enough that the derivation of whp is possible with few user interactions and yields useful results. In particular, these classes should contain for-loops, i.e. loops where the number of executions is dependent on some program variable.
2. **Define a syntax for these classes of programs and hyperquantities and for additionally needed hints.** Transform this syntax into a language that can be given to the automated whp computation as input.
3. **Implement a program that automatically computes** whp **for the identified classes of programs and hyperquantities.**
4. One of the following:
   a) **Implementing a *Verifier* that can check entailment of a pre-hyperquantity.** In the classical setting, we usually check for a program C whether a precondition P implies the weakest-pre of a post-condition Q: $P \implies wp\ [\![C]\!]\,(Q)$. Correspondingly, we want to check whether a pre-hyperquantity ff is a point-wise lower-bound of the weakest-pre of a post-hyperquantity gg: $\forall f.\, ff\,(f) \leq \text{whp}\ [\![C]\!]\,(gg)\,(f)$.
   b) **Implementing an *Estimator* that applies the pre-hyperquantity to one specific initial distribution over states.** This way, we

can e.g. get the expected value of a final variable given some distribution over input variables.

5. **Perform relevant case studies** to evaluate the automation.

3.2 Extension Goals

1. **Core goal 4a or 4b that has not been implemented yet.**
2. **Extend the classes of programs and hyperquantities to support more relevant case studies.** One of the extension could be to support arbitrary while loops.
3. **Formalize the simplifications used for automation in a proof-assistant.** Formally proving the rules used for the automation increases confidence in the correctness of the tool. Additionally, limitations and the classes of programs and hyperquantities become clearer and more formal. This might help to discover extensions or find out that the classes cannot be extended without changing simplifications fundamentally.
4. **Explore more challenging case studies.**

References

[1] Christopher Cohnen. "Usability and Applications of Quantitative Weakest Hyper Pre". Bachelor's Thesis. Saarbrücken, Germany: Universität des Saarlandes, 2024.

[2] 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.

[3] Thibault Dardinier and Peter Müller. *Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version)*. 2023. arXiv: 2301.10037 [cs.LO].

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Donald E. Knuth. *Two notes on notation*. 1992. arXiv: math/9205211 [math.HO].

[10] 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. URL: https://www.sciencedirect.com/science/article/pii/0022000085900121.

[11] Philipp Schröer et al. "A Deductive Verification Infrastructure for Probabilistic Programs". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622870. URL: https://doi.org/10.1145/3622870.

[12] Linpeng Zhang et al. *Quantitative Weakest Hyper Pre: Unifying Correctness and Incorrectness Hyperproperties via Predicate Transformers*. 2024. arXiv: 2404.05097 [cs.LO].