ETH zürich

Bachelor's Thesis

Automating Hyper Hoare Logic via Predicate Transformers

David Hagen

December 19, 2025

Supervisors: Dr. Thibault Dardinier
Prof. Dr. Peter Müller

Department of Computer Science
ETH Zürich

Abstract

Hyper Hoare Logic (HHL) is a powerful proof system that is expressive enough to prove or disprove any program hyperproperty. Hypra is the first deductive verifier for HHL. Its current implementation constructs verification conditions via a semantic pipeline that models sets of program states by lower- and upper-bound approximations. However, the resulting verification conditions quickly become bulky and demanding on backend solvers and scale poorly as program size increases.

This thesis develops a syntactic backend for HHL as an alternative verification engine in Hypra. The starting point is a syntactic pipeline consisting of a path-based characterizer and a resulting weakest-precondition (WP) construction for split-free programs, i.e., programs consisting only of loop-free and call-free statements. This core is then extended to handle method calls and loops by introducing various splitting schemas that decompose complex statements into families of split-free triples, which can be discharged by our core pipeline and together imply correctness of the original triple. It also includes an error-propagation mechanism that maintains soundness across triple boundaries when reasoning about error-states, as well as an optional automatic framing feature.

To make the split-based approach practical, the thesis integrates the calculus into Hypra as a fully functional syntactic backend implemented in Scala. It exposes several SMT configurations, letting the user choose whether to use Z3 or cvc5 as the SMT solver for verification, and even provides a race mode that runs both solvers in parallel to leverage the strengths of each solver.

An extensive empirical evaluation shows that the syntactic backend is competitive with the existing implementation. It achieves a high success rate while, on average (geometric mean), discharging verification obligations about ten times faster than the existing semantic pipeline.

Contents

1 Introduction

Modern software systems increasingly handle sensitive data, interact with untrusted environments, and are expected to satisfy strong correctness and security guarantees. Software testing is the traditional approach of evaluating program correctness, but it is limited, as it can only explore finitely many concrete executions of an implementation. In contrast, formal verification reasons about all possible executions within a given model, and can therefore provide mathematical assurance that a program behaves as intended[1].

Classical *Hoare Logic* [1] offers a well-established framework for reasoning about individual executions of a program. It does this by establishing *Hoare triples* of the form $\{P\}\ C\ \{Q\}$, where P is a precondition, C is a statement (or program), and Q is a postcondition. This triple holds if and only if, whenever the precondition holds before executing C, then the postcondition Q holds afterwards (if C terminates).

However, many relevant security and correctness requirements one wants to argue about are inherently relational: They do not only reason about individual executions, but relate multiple executions of the program. Examples of such properties are determinism[2], or non-interference[3] [2], an important security property. These relational properties are instances of *hyperproperties* [3], which describe relationships among multiple execution traces, rather than only talking about individual traces.

Hyper Hoare Logic (HHL) [4] is a recent extension of Hoare Logic designed to reason about such hyperproperties for an unbounded number of executions. It lifts pre- and postconditions from classic assertions to hyper-assertions over sets of states and provides proof rules that are expressive enough to prove or disprove program hyperproperties with arbitrary quantifier alternations. *Hypra* [5] is the first deductive verifier for HHL and works by *semantically* encoding hyperproperties and the corresponding proof rules into an intermediate verification language, namely Viper [6].

This thesis explores a complementary, *syntactic* automation strategy for HHL based on predicate transformers. Rather than relying on a separate, fully semantic encoding of programs and their executions, we derive verification conditions directly from the HHL proof rules by computing weakest preconditions. The goal is to obtain a more efficient verification pipeline while preserving the expressiveness of HHL and the range of hyperproperties that can be verified.

The remainder of this chapter is structured as follows: Section 1.1 discusses the limitations of the current semantic encoding and motivates our syntactic alternative. Section 1.2 then outlines the overall approach of this project on a high level and summarizes its main contributions.

1: Here, *"intended"* refers to the mathematically specified behavior of the program as given by the user's formal specification, rather than any informal or implicit notion of correctness.

[1]: Hoare (1969), 'An axiomatic basis for computer programming'

2: Different runs from the same input produce the same observable behaviour.

3: Changes in secret inputs must not affect public outputs.

[2]: Volpano et al. (1996), 'A Sound Type System for Secure Flow Analysis'

[3]: Clarkson et al. (2010), 'Hyperproperties'

[4]: Dardinier et al. (2024), 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

[6]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

1.1 Problem Statement and Motivation

The current automation strategy implemented in Hypra is based on a *semantic* encoding of HHL into Viper syntax, which inherently doesn't

support hyperproperties. This is done by explicitly modeling sets of program states in Viper's first-order states by computing both a lower and an upper bound for these sets of states as an approximation. While this allows Hypra to reuse mature verification infrastructure, the resulting verification conditions can become large and complex, even for comparatively small programs. This, in turn, imposes a significant burden on Viper's backend solvers, can lead to long verification times, and scales poorly as program size increases.

Listing 1.1: A simple HHL method inc whose pre- and postcondition state that if the value of the input x is not bound from above, then neither is the value of the output y.

```
method inc(x: Int) returns (y: Int)
    requires forall <_s1> :: (exists <_s2> :: _s1[x] < _s2[x])
    ensures  forall <_s1> :: (exists <_s2> :: _s1[y] < _s2[y])
{
    y := x + 1
}
```

To make these drawbacks tangible, consider the relatively simple method `inc` in Listing 1.1. Starting from a small HHL proof obligation, Hypra translates the program into the following Viper encoding, given in Listing 1.2. Even for this toy example, most of the proof effort comes from the encoding's bookkeeping rather than from the actual property: Hypra must introduce explicit representations for sets of initial and final states and track two approximations (a lower and an upper bound) of each set. The single assignment `y := x + 1` is then lifted to the set level via quantified constraints relating pre- and post-state sets. As a result, the SMT backend is confronted with a comparatively large and entangled first-order query whose size and difficulty are out of proportion to the simplicity of the original HHL goal.

Listing 1.2: The semantic encoding of the example in Listing 1.1 in Viper syntax.

```
var S0∀, S0∃: Set[Map[Var, Int]]  // set of initial states
var S1∀, S1∃: Set[Map[Var, Int]]  // set of final states

assume ∀σ ∈ S0∀. ∃σ' ∈ S0∃. σ(x) < σ'(x)  // precondition

assume ∀σ1 ∈ S1∀. ∃σ0 ∈ S0∀. σ1 = σ0[y ↦ σ0(x) + 1]  // y := x + 1

assume ∀σ0 ∈ S0∃. ∃σ1 ∈ S1∃. σ1 = σ0[y ↦ σ0(x) + 1]  // y := x + 1

assert ∀σ ∈ S1∀. ∃σ' ∈ S1∃. σ(y) < σ'(y)  // postcondition
```

Moreover, the additional encoding layer makes it harder to interpret and debug failures, because users are confronted with errors in the encoded Viper program rather than directly in the HHL proof obligations they started from[4].

4: This issue has been addressed for the current semantic Hypra in a separate project [7].

These observations motivate the search for a more direct, syntactic automation strategy for HHL. One would like to compute verification conditions directly at the level of HHL and feed them to an SMT solver in a more lightweight form. The main question of this thesis is whether a syntactic framework can provide an efficient and practical alternative to Hypra's existing semantic pipeline, without sacrificing the expressiveness of HHL.

1.2 Thesis Overview and Contributions

This project investigates a syntactic automation strategy for Hyper Hoare Logic based on predicate transformers. The central idea is to compute

the *weakest precondition (WP)* [8] directly for HHL programs, rather than going to an intermediate representation.

[8]: Dijkstra (1975), 'Guarded commands, nondeterminacy and formal derivation of programs'

Given the fact that hyper-postconditions can relate different executions, the weakest precondition must account for every control-flow path that any of those executions might follow. Only then can we guarantee that the postcondition will hold no matter which path each execution actually takes. The need to account for every control-flow path in turn forces us to introduce *splits*[5] in the program, as soon as we deal with more complex language features. These requirements impose significant challenges for our syntactic automation approach, and in the following sections, we explain in detail how we address these challenges and motivate our design choices.

5: We will introduce splits and explain when they are needed in Chapter 3.

1.2.1 Contributions

In summary, our work makes the following contributions:

- Development of a syntactic WP construction for HHL for loop-free and call-free (which we will call *split-free*) statements.
- Extension of this construction to support method calls and loops.
- Integration into the existing Hypra implementation as a fully functioning alternative backend with multiple solver modes.
- Extensive empirical comparison with the existing semantic pipeline.

The thesis is structured in the way the project itself was incrementally extended. The remainder of this thesis is organized as follows: Chapter 2 presents the syntactic weakest-precondition pipeline: it defines the core approach of how we syntactically construct weakest preconditions. Chapter 3 extends this methodology to more complex structures like loops and method calls. Chapter 4 describes the actual implementation of the new backend and explains key design and engineering decisions. It also reports on the empirical evaluation and compares the syntactic and semantic pipelines on a range of benchmarks. Finally, Chapter 5 concludes and outlines directions for future work.

2 Approach for Split-free Programs

This chapter presents the core of our syntactic approach: The weakest-precondition-based approach for **split-free programs**. In this chapter, we first focus on a fragment of the language where no further structural decomposition of the code is needed. In Chapter 3, we then extend the approach to arbitrary programs by splitting them into multiple parts. This allows us to cleanly introduce the syntactic pipeline without having to deal with the complications of loops and method calls for now. We define a *split-free statement* as follows:

Definition 2.0.1 (Split-free statement) *We call a statement* split-free *if it does not contain any loops or method calls*[a].

[a] In addition, we treat `hyperAssume` and `hyperAssert` as splitting constructs. Any statement containing them is also not split-free.

In other words, a split-free statement contains only assignments[1] within (possibly nested) if-else branches and `assume` and `assert` statements.

1: These can also be nondeterministic assignments through the `havoc` keyword.

We first give a bird's-eye view of the verification pipeline developed throughout this chapter. Figure 2.1 gives a preview on the end-to-end flow. Starting from an input triple $\{P\}\ C\ \{Q\}$, the pipeline computes a characterizer Ψ summarizing the execution paths of C. The characterizer is then used to derive a weakest precondition $\mathsf{E}_\Psi[Q]$. Finally, the verification conditions are encoded into an SMT formula and discharged by an SMT solver.

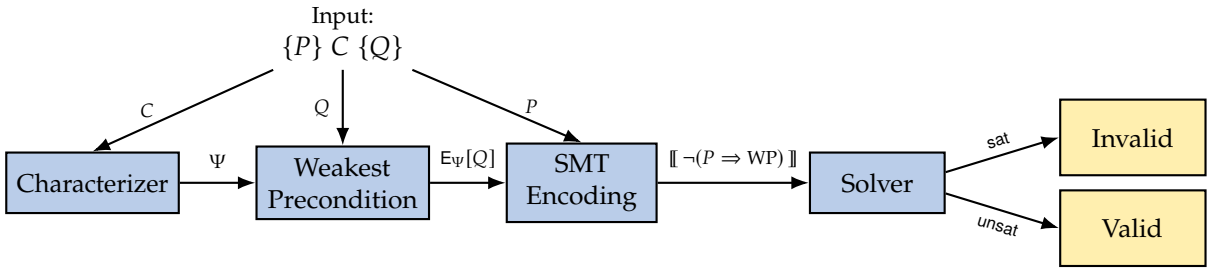


Figure 2.1: Verification pipeline for split-free programs. From the input triple $\{P\}\ C\ \{Q\}$ we compute a characterizer Ψ, derive the weakest precondition $\mathsf{E}_\Psi[Q]$, encode the verification condition into an SMT formula, and discharge it using an SMT solver.

In the remainder of this chapter, we will first introduce the syntactic pipeline for the base language without `havoc` and `assert` statements in Section 2.1, and then progressively extend this approach to nondeterminism in Section 2.2 and errors in Section 2.3. Finally, in Section 2.4 we explain how the resulting verification conditions are encoded for SMT solvers, which completes the syntactic backend for split-free programs.

```
C ::= skip
    | x := e
    | C ; C
    | if (b) {C} else {C}
    | assume b
```

Figure 2.2: Grammar of the base language. The complete formal grammar of `e` and `b` can be found in Appendix A.1. We also treat line breaks as semicolons.

2.1 Fundamental Approach: Error-free and Deterministic Programs

In this subsection, we introduce the syntactic pipeline for the simplest relevant fragment of the input language: Error-free, deterministic, and

split-free statements. Concretely, we consider programs built from the grammar in Figure 2.2. This allows us to introduce the verification pipeline in its purest form before extending it to more complex language constructs.

We use the following running example in Listing 2.1 throughout the section to illustrate the introduced concepts more lucidly.

Listing 2.1: Running example: A simple if-else snippet that assigns either `a` or `b` to `x`, based on the value of `c`. The postcondition in line 3 asserts the existence of a maximum for `x`. The precondition of the method is omitted for better readability, as it is not relevant to what we want to illustrate with this example.

```
1  method ifElse(a: Int, b: Int, c: Int) returns (x: Int)
2      requires ...
3      ensures  exists <_s1> :: (forall <_s2> :: _s1[x] >= _s2[x])
4  {
5      if (c > 0) {
6          x := a
7      } else {
8          x := b
9      }
10 }
```

2.1.1 Characterizer

Our overall verification goal is the following: Given a user-defined hyper-precondition P, a program statement C, and a user-defined hyper-postcondition Q, we want to determine whether the Hyper Hoare Logic triple $\{P\}\ C\ \{Q\}$ is valid.

To do so in a syntactic way, the general well-established approach is to compute a weakest precondition $P_w = \text{WP}(C, Q)$ using the syntactic proof rules of the logic, and then checking whether the entailment $P \models P_w$ holds, where P is a user-defined precondition. We adopt this strategy in our setting as well.

[8]: Dijkstra (1975), 'Guarded commands, nondeterminacy and formal derivation of programs'

Usually, the weakest precondition is defined compositionally. For instance, in classic Hoare Logic we have the smooth equality [8]

$$\text{WP}(\texttt{if}\ (b)\ \{\ C_1\ \}\ \texttt{else}\ \{\ C_2\ \}, Q) = (b \wedge \text{WP}(C_1, Q)) \vee (\neg b \wedge \text{WP}(C_2, Q))$$

where the function *WP* maps a command C and a postcondition Q to the weakest precondition. This disjunction works because the postcondition Q talks about *one* final state at a time.

However, in our HHL setting, a postcondition Q often relates *multiple* final states, e.g., it might compare a state that reached the then-branch with a state that reached the else-branch. Computing weakest preconditions separately for each path cannot express the required cross-path comparisons between different executions.

Our approach to overcome this issue is to enumerate all program paths first by computing a so-called *characterizer* of the program. The first version[2] of a characterizer is defined as follows:

2: We will gradually extend this definition throughout the chapter, when we consider more complex language constructs.

Definition 2.1.1 (Characterizer V1) *A* characterizer Ψ *of a program* C *in our base language is a list of pairs* (pc, θ)*, where each element of the list covers one path of the program. The characterizer itself then covers exactly all paths of* C*. The components* pc *and* θ *are defined by:*

- *The path condition* pc *is a program boolean expression, which holds on an initial state if and only if this initial state can take this path.*

- *The substitution θ maps each variable x of the program to an expression e, meaning that the variable x in the final state will have the value of the expression e in the initial state.*

Formally, the following should hold for a characterizer[3]:

$$\langle C, \sigma\rangle \to \sigma' \implies \mathrm{pc}(\sigma) \implies \sigma'(x) = e(\sigma),$$

where the notation $e(\sigma)$ denotes the semantic evaluation of the expression e in state σ.

3: In plain words: Whenever running a program C from an initial state σ leads to a state σ', and the path condition pc holds in the initial state σ, then in the final state σ' the value of the variable x is exactly the value of the expression e evaluated in the initial state.

Example 2.1.1 As an illustration, consider our running example from Listing 2.1 again. The corresponding characterizer for this program is:

$$\big[(c > 0, (x \mapsto a)), (\neg(c > 0), (x \mapsto b))\big]$$

That is, we have two paths, one that is being explored if $c > 0$ holds, in which x gets updated to a, and the other one is taken if $\neg(c > 0)$ holds, where x gets updated to b.

```
4    if (c > 0) {
5        x := a
6    } else {
7        x := b
8    }
```

2.1.2 Obtaining the Weakest Precondition

Now that we have introduced the notion of a characterizer, we can turn to the second step of the syntactic pipeline: computing the **weakest precondition (WP)** for a given postcondition.

As already explained before, our weakest precondition must account for every control-flow path that any of those executions might follow in the program. We leverage the characterizer's path-wise decomposition: it converts a statement into a finite[4] list of path conditions and corresponding substitutions, which form the basis of our WP construction.

4: As the input program is finitely long and doesn't contain any loops or method calls (in particular, no recursion), the number of paths in this program is also finite.

Using these ingredients, the weakest precondition for a state-quantified hyper-assertion is obtained by expanding over the characterizer paths and applying the construction recursively to nested state quantifiers inside Q. We define the transformation $\mathsf{E}_\Psi[\cdot]$ on state-quantified hyper-assertions by recursion on the *outermost state quantifier* as follows.

Definition 2.1.2 (Recursive path expansion for state quantifiers) *Let $\Psi = \{(b_1, \theta_1), \ldots, (b_n, \theta_n)\}$ be the set of path pairs returned by the characterizer, where each b_i is the path condition, and each θ_i is the corresponding substitution for this path. We define a transformation $\mathsf{E}_\Psi[\cdot]$ on hyper-assertions with an* outermost state quantifier *by:*

$$\mathsf{E}_\Psi\big[\forall\langle\sigma\rangle.\, Q\big] \triangleq \forall\langle\sigma\rangle.\, \bigwedge_{i=1}^{n}\Big(b_i(\sigma) \Rightarrow \mathsf{E}_\Psi\big[\, Q[\theta_i]_\sigma \,\big]\Big),$$

$$\mathsf{E}_\Psi\big[\exists\langle\sigma\rangle.\, Q\big] \triangleq \exists\langle\sigma\rangle.\, \bigvee_{i=1}^{n}\Big(b_i(\sigma) \wedge \mathsf{E}_\Psi\big[\, Q[\theta_i]_\sigma \,\big]\Big),$$

where we have used the following notations:

- *$Q[\theta_i]_\sigma$ denotes the syntactic instantiation of Q according to θ_i in the state variable σ (i.e., occurrences of $\sigma(x)$ are rewritten using the expression that θ_i assigns to x, also as a state lookup of σ).*
- *$b(\sigma)$ denotes the formula obtained by replacing every program variable x in b with a state lookup $\sigma(x)$.*

The transformation is applied recursively: if the body $Q[\theta_i]_\sigma$ contains further state quantifiers, $\mathsf{E}_\Psi[\cdot]$ expands them again using the same rule.

For a universal state quantifier, we obtain a conjunction over all characterizer paths, discharging each case conditionally: for every path, we prove the property under the hypothesis that the current state follows that path, i.e., guarded by an implication from the path condition. Dually, for an existential state quantifier, we obtain a disjunction over all paths, and witness a path by conjoining its path condition. All information required for this expansion is provided directly by the characterizer.

For readability, we only present here the clauses of $\mathsf{E}_\Psi[\cdot]$ that handle state quantifiers, since they are the only nontrivial part of the construction and the only place where path expansion and recursion interact. The remaining cases (Boolean connectives and first-order quantifiers) follow the standard, purely structural recursion on the syntax of hyper-assertions and do not introduce any additional ideas. The complete definition is given in Appendix A.2.

Remark 2.1.1 (Complexity of recursive path expansion) Let Ψ be the characterizer for a program and write $m = |\Psi|$. Moreover, let k be the *state-quantifier depth* of the hyper-assertion Q, i.e., the maximum number of nested occurrences of $\forall\langle\cdot\rangle$ and $\exists\langle\cdot\rangle$ in Q.

By Definition 2.1.2, the transformation $\mathsf{E}_\Psi[Q]$ expands each state quantifier by branching over all m paths (forming a conjunction or disjunction) and then recursing on the instantiated body. Hence, along a quantifier chain of length k, the number of path cases multiplies by a factor of m at each level. In particular, the expanded formula contains $O(m^k)$ distinct path/quantifier combinations.

Example 2.1.2 We continue with our running example (Listing 2.1). In Example 2.1.1 we have have derived the characterizer[5] for the program, which we will now use to construct the weakest precondition:

The program has the following postcondition, asserting the existence of a maximum for x:

$$\exists\langle\sigma\rangle.\ \forall\langle\sigma'\rangle.\ \sigma(x) \geq \sigma'(x)$$

We now compute the weakest precondition by applying the recursive state-quantifier expansion from Definition 2.1.2 step by step:

1. Expanding the outer existential state quantifier[6]:

$$\begin{aligned}\mathsf{E}_\Psi[Q] &= \mathsf{E}_\Psi\left[\exists\langle\sigma\rangle.\ \forall\langle\sigma'\rangle.\ \sigma(x) \geq \sigma'(x)\right] \\ &= \exists\langle\sigma\rangle.\ \Big((\sigma(c) > 0)\ \wedge\ \mathsf{E}_\Psi\left[\forall\langle\sigma'\rangle.\ \sigma(a) \geq \sigma'(x)\right]\ \vee \\ &\qquad \neg(\sigma(c) > 0)\ \wedge\ \mathsf{E}_\Psi\left[\forall\langle\sigma'\rangle.\ \sigma(b) \geq \sigma'(x)\right]\Big)\end{aligned}$$

2. Recursion: Expanding the inner universal state quantifiers:

$$\begin{aligned}&\mathsf{E}_\Psi\left[\forall\langle\sigma'\rangle.\ \sigma(a) \geq \sigma'(x)\right] = \\ &\qquad \forall\langle\sigma'\rangle.\ \Big((\sigma'(c) > 0) \Rightarrow (\sigma(a) \geq \sigma'(a)) \wedge \\ &\qquad\qquad \neg(\sigma'(c) > 0) \Rightarrow (\sigma(a) \geq \sigma'(b))\Big),\end{aligned}$$

5: Restated here for convenience:

$$\left[(c > 0, (x \mapsto a)), (\neg(c > 0), (x \mapsto b))\right]$$

6: Note, how the $\sigma(x)$ are already substituted, whereas the $\sigma'(x)$ aren't. This is exactly what $Q[\theta_i]_\sigma$ gives us. The remaining substitutions are handled in the recursion step.

The second case works analogously and is therefore omitted.

3. Assembling the final WP. Plugging Step 2 into Step 1 yields:

$$\begin{aligned}\mathsf{E}_\Psi[Q] = \exists\langle\sigma\rangle.\ \big(\sigma(c) \wedge \big(\forall\langle\sigma'\rangle.\ \big(\sigma'(c) \Rightarrow \sigma(a) \geq \sigma'(a)\big) \wedge \\ \big(\neg\sigma'(c) \Rightarrow \sigma(a) \geq \sigma'(b)\big)\big)\big) \\ \vee \big(\neg\sigma(c) \wedge \big(\forall\langle\sigma'\rangle.\ \big(\sigma'(c) \Rightarrow \sigma(b) \geq \sigma'(a)\big) \wedge \\ \big(\neg\sigma'(c) \Rightarrow \sigma(b) \geq \sigma'(b)\big)\big)\big)\end{aligned}$$

This weakest precondition ranges over all program paths and combines them so that every possible interleaving of paths taken by two[7] executions is considered.

This completes the core of the syntactic pipeline, and shows how we can syntactically derive a weakest precondition $P_w = \mathsf{E}_\Psi[Q]$ for a hyper-postcondition Q and a statement C in our base language. The only step left is to check whether the entailment $P \models P_w$ holds. If this is the case, our hyper-triple $\{P\}\ C\ \{Q\}$ holds, and the program can be considered correct with respect to its user-defined specification.

We concretely show in Section 2.4 how we model this entailment of hyper-assertions, such that it can be interpreted and solved by an SMT solver. Before that, the next sections extend the pipeline to additional language features, which require refining the concepts just introduced.

7: In this concrete example, we reason about pairwise combinations, because the hyper-postcondition has two state quantifiers.

2.2 Supporting Nondeterminism

The next step is to extend our approach to support nondeterministic assignments, written as `havoc` in our input language. The grammar of this extended input language is shown in Figure 2.3. Intuitively, `havoc x` assigns an arbitrary value to the variable[8] `x`, which lets us model unbounded nondeterminism. Together with the `assume` keyword, we can also model bounded nondeterminism [5].

```
C ::= skip
    | x := e
    | havoc x
    | C ; C
    | if (b) {C} else {C}
    | assume b
```

Figure 2.3: Grammar of the input language, extended with nondeterministic assignments.

Unlike deterministic assignments, this means that even a single initial state can lead to many possible final states, depending on which value is effectively chosen for `x`. Consequently, our previous model, where we have concrete substitutions mapping variables to expressions in the initial state, is no longer sufficient. Instead, we must treat `havoc` as an unconstrained choice that the weakest precondition has to quantify over.

Naïvely, one could think of `havoc x` as just assigning some fresh "method parameter", i.e., treating `x` like an extra input. However, this is unsound. The subtle problem is that this idea models one choice that is fixed in a concrete state but unknown, whereas `havoc` should represent all possible choices the program may make in any concrete state.

We can quickly see why this naïve approach is unsound in the following example program:

```
method naiveHavoc() returns (z: Int)
    requires forall <_s1>, <_s2> :: _s1 == _s2
    ensures  forall <_s1>, <_s2> :: _s1[z] == _s2[z]
{
    havoc z
}
```

Listing 2.2: A small code snippet showing why simply treating `havoc` as an assignment to a fresh parameter is unsound.

8: `x` has to be either a previously declared variable, or a return parameter; just like for a deterministic assignment.

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

The precondition says that all initial states are identical. However, if we encoded `havoc x` as `z := p` for a fresh parameter `p`, then any two executions starting from the same initial state would necessarily assign the same value[9] to `z`. The postcondition would then appear to hold.

9: Because they are the same state, they also read the same value for `p`.

But under the actual meaning of `havoc`, even if all initial states are equal, two executions may pick different values for `z`. Hence, the postcondition in this example doesn't hold in general. This shows that supporting nondeterminism requires us to extend our existing pipeline.

Remark 2.2.1 (Eliminating havoc hints) In the semantic automation approach [5], reasoning about nondeterminism relied on explicit user-annotated hints, which gave an "idea" to the backend solver what would be a reasonable value to instantiate the variable with. Each `havoc` had to be accompanied by such an annotation.

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

Our syntactic approach doesn't use hints anymore[10]. This makes nondeterminism lightweight for users and keeps specifications focused on the intended hyperproperty rather than on proof engineering.

10: In fact, our implementation ignores hints given in the input program completely.

That said, hints can still be beneficial in *witness-finding* situations – most notably for existentially quantified executions. In such cases, ranging over all nondeterministic outcomes can make the resulting SMT problems harder, and providing a witness (or constraining the search space) can improve robustness. Nevertheless, our evaluation[11] indicates that fully automatic solving performs well for the overwhelming majority of programs we considered.

11: See Section Section 4.2 on page 46

2.2.1 Extending the Characterizer

When constructing the weakest precondition, we need more information about the nondeterministic assignments in our program. The most convenient way to obtain this information is by slightly extending our current notion of a characterizer and collecting it in the same pass. Whenever we encounter a `havoc x` for a variable x in the code, we perform the following steps:

1. We define a fresh special-purpose symbol h_i and update the substitution map θ with $x \mapsto h_i$.
2. We append this new symbol h_i to a list $\mathcal{H}$ where we track all havoc symbols introduced while traversing the program.

This brings us to the following updated definition of a characterizer:

Definition 2.2.1 (Characterizer V2) *Let C be a statement in our input language as given in Figure 2.3 that may contain nondeterministic assignments. A* characterizer *of C is a pair $(\Psi, \mathcal{H})$, where Ψ is a characterizer as defined in Definition 2.1.1 which can now also track havoc symbols and $\mathcal{H}$ is a finite set of fresh* havoc symbols *common to all paths.*

The new definition extends the initial version from Definition 2.1.1 to track nondeterministic updates, while keeping its role unchanged. It still provides a summary of program behavior that serves as the input to the WP construction. Importantly, this generalized notion is backwards compatible: on deterministic programs, the set $\mathcal{H}$ is empty and the characterizer is equivalent to its initial definition.

2.2.2 Weakest Preconditions with Nondeterminism

When computing the weakest precondition, the key change is that nondeterministic choices must be indexed by executions. A `havoc` does not introduce one global unknown value, but a fresh unknown value for each quantified state in the hyper-assertion.

Again, we are given a hyper-triple $\{P\}\ C\ \{Q\}$ and a characterizer $(\Psi, \mathcal{H})$ for C. Before applying the recursive path expansion $\mathsf{E}_\Psi[\cdot]$ from Definition 2.1.2, we make the nondeterministic choices introduced by `havoc` explicit by extending every state quantifier with fresh havoc variables local to that state.

Definition 2.2.2 (State quantifiers with local havoc variables) *Let $\mathcal{H} = \{h_1, \ldots, h_k\}$ be the set of havoc variables occurring in C, given by the characterizer. For each state variable σ, we introduce a fresh tuple of symbols $\mathcal{H}_\sigma = \{h_1^\sigma, \ldots, h_k^\sigma\}$, representing the nondeterministic choices made in executions instantiated by σ. We define a rewriting $\mathsf{H}[\cdot]$ on state quantifiers by*

$$\mathsf{H}\big[\forall\langle\sigma\rangle.\, Q\big] \triangleq \forall\langle\sigma\rangle.\, \forall(\mathcal{H}_\sigma).\, \mathsf{H}[Q] \qquad \textit{and}$$
$$\mathsf{H}\big[\exists\langle\sigma\rangle.\, Q\big] \triangleq \exists\langle\sigma\rangle.\, \exists(\mathcal{H}_\sigma).\, \mathsf{H}[Q],$$

where $\mathsf{H}[Q]$ applies the same rewriting recursively to any further state quantifiers occurring in Q. All other constructs of hyper-assertions are handled by straightforward structural recursion (omitted).

We then compute the weakest precondition by applying the *same* recursive path expansion as before, but to the havoc-extended postcondition:

$$P_w = \mathsf{E}_\Psi\big[\mathsf{H}[Q]\big].$$

The only difference is that substitutions θ provided by the characterizer may now mention havoc symbols. Concretely, when expanding a state quantifier over the current state variable σ, we allow each path-specific instantiation $Q[\theta]_\sigma$ to reference the corresponding local havoc symbols $h_1^\sigma, \ldots, h_k^\sigma$, reflecting that the nondeterministic choices are made independently for each quantified state.

As a concrete illustration of how this extension of the WP pipeline works, consider the following example:

```
method havocExample() returns (x: Int, y: Int)
    requires ...
    ensures  forall <_s1> :: (exists <_s2> ::
                _s2[x] == _s1[x] + 1 && _s2[y] == _s1[y] - 1)
{
    havoc x
    havoc y
}
```

Listing 2.3: A code snippet with two `havoc` statements to illustrate how the additional havoc symbols are included in the WP construction. Again, the precondition of the method is omitted for better readability.

Example 2.2.1 Consider the program in Listing 2.3, which simply consists of two `havoc` statements. The postcondition

$$Q = \forall\langle\sigma_1\rangle.\, \exists\langle\sigma\rangle.\, \sigma_2(x) = \sigma_1(x) + 1 \ \wedge\ \sigma_2(y) = \sigma_1(y) - 1$$

asserts that for every possible final state, there is another possible final state where `x` is increased by 1 and `y` is decreased by 1. We want to

compute the WP for this program.

The characterizer gives us the following result:

$$(\Psi, \mathscr{H}) = \Big(\big[(\top, (x \mapsto h_1, y \mapsto h_2))\big], \{h_1, h_2\}\Big)$$

We first apply $\mathsf{H}[Q]$ by instantiating the havoc symbols for every state: $h_1^{\sigma_1}, h_2^{\sigma_1}, h_1^{\sigma_2}, h_2^{\sigma_2}$, and then inserting them at the respective locations with the right quantifiers:

$$\mathsf{H}[Q] = \forall\langle\sigma_1\rangle.\ \forall h_1^{\sigma_1}, h_2^{\sigma_1}.\ \exists\langle\sigma\rangle.\ \exists h_1^{\sigma_2}, h_2^{\sigma_2}.\\ \sigma_2(x) = \sigma_1(x) + 1\ \wedge\ \sigma_2(y) = \sigma_1(y) - 1$$

Finally, we apply $\mathsf{E}_\Psi[\cdot]$ on $\mathsf{H}[Q]$, which yields the following weakest precondition:

$$P_w = \mathsf{E}_\Psi\big[\mathsf{H}[Q]\big] = \forall\langle\sigma_1\rangle.\ \forall h_1^{\sigma_1}, h_2^{\sigma_1}.\ \exists\langle\sigma\rangle.\ \exists h_1^{\sigma_2}, h_2^{\sigma_2}.\\ \big(h_1^{\sigma_2} = h_1^{\sigma_1} + 1\ \wedge\ h_2^{\sigma_2} = h_2^{\sigma_1} - 1\big)$$

As we only have one path in the program, path expansion does not affect the structure of the WP.

2.3 Supporting Runtime Errors

```
C ::= skip
    | x := e
    | havoc x
    | C ; C
    | if (b) {C} else {C}
    | assume b
    | assert b
```

Figure 2.4: Grammar of the input language, extended with the `assert` statement.

We now turn to the final extension needed to complete the split-free weakest precondition pipeline: reasoning about **runtime errors**. We therefore extend our input language again to support assertions and show the updated grammar in Figure 2.4.

Everything we have covered until now implicitly assumes that an input program never runs into an error during its execution. This assumption matches the original introduction of Hyper Hoare Logic [4]. Hypra, however, extends the semantics of HHL with explicit *error states* to model runtime failures [5]. This gives rise to a range of interesting and nontrivial hyperproperties reasoning about the presence or absence of runtime errors in a program[12].

Remark 2.3.1 (Notation for error states) In our input language, runtime errors are introduced exclusively through the `assert` keyword: an `assert b` statement checks the Boolean condition b at runtime and, if b does not hold in the current state, the execution transitions into an explicit *error state*, instead of a final state as usual. Correspondingly, hyper-assertions can include these error states if they should reason about error-related properties.

We extend state quantification to also range over error states, writing

- $\langle\sigma\rangle$, or `<_s>` in the source code, for "σ (resp. `s`) is a normal reachable state", and
- $\langle\sigma\rangle_{er}$, or `<<_s>>` in the source code, for "σ (resp. `s`) is a reachable error state."

To illustrate how error states are handled in our syntactic pipeline, we use the small code snippet in Listing 2.4. We will return to this program throughout the next subsections to connect the formal constructions to a concrete example.

[4]: Dardinier et al. (2024), 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

12: One interesting example is *failure-sensitive non-interference*. This is the property that observing a runtime error does not leak secret information.

```
method errorExample(x: Int) returns (e: Int)
    requires ...
    ensures  forall <<_s>> :: (_s[e] == 0) || (_s[e] == 1)
{
    e := 0
    assert x != 0
    e := 1
    assert x != 1
    e := 2
}
```

Listing 2.4: Running example for error-state handling with two potentially failing assertions, based on the value of `x`. Again, the precondition of the method is omitted for better readability

2.3.1 The Complete Characterizer

Conceptually, the challenge is that errors abort the execution early and produce a terminal program state at the point of failure. So far, this was not an issue because our characterizer only needed to describe the unique normal return point at the end of the program. With error states, however, a program may terminate at multiple points – either normally at the end or prematurely at a failing assertion. To reason about such executions, we therefore have to extend our notion of a characterizer one last time.

Definition 2.3.1 (Characterizer V3: final version) *Let C be a statement in our input language as given in Figure 2.4 that may contain nondeterministic assignments and assertions. A* characterizer *of C is a triple $(\Psi, \mathcal{H}, \mathcal{A})$, where $(\Psi, \mathcal{H})$ is a characterizer as in Definition 2.2.1, and $\mathcal{A}$ is a map assigning to each syntactic `assert b` statement[13] α in C a list Ψ_α of path pairs as defined in Definition 2.1.1 that characterizes the execution prefix up to α.*

13: Occurrences of assertions are distinguished by their position in C, even if they have the same condition.

Similarly to before, if C contains no `assert` statement, then $\mathcal{A}$ is empty, and the characterizer coincides with the version from Definition 2.2.1.

Example 2.3.1 We now return to the running example from Listing 2.4 to illustrate how the characterizer is constructed in the presence of assertions. The resulting characterizer for this program is: $(\Psi, \mathcal{H}, \mathcal{A}) =$

$$\Big(\big[\big((x \neq 1 \wedge x \neq 0),\ (e \mapsto 2)\big)\big], \emptyset, \big(\alpha_1 \mapsto \Psi_{\alpha_1}, \alpha_1 \mapsto \Psi_{\alpha_1}\big)\Big)$$

where α_1 is `assert x != 0` and α_2 is `assert x != 1`. We have:

$$\Psi_{\alpha_1} = \big[\big(\top,\ (e \mapsto 0)\big)\big] \quad \text{and} \quad \Psi_{\alpha_2} = \big[\big(x \neq 0,\ (e \mapsto 1)\big)\big].$$

This tells us two key facts:

1. There is only one normal path to the end of the program, and on that path we result in the substitution $(e \mapsto 2)$.
2. There are two assertions in the program, each with exactly one program path leading to it. We get the path conditions and the corresponding substitutions right before the assertions.

It is also worth noting how assertions affect path conditions: once an assertion is encountered, the fact that it holds is conjoined to the path condition. In this sense, `assert b` behaves like `assume b` for normal execution, but additionally yields error states.

2.3.2 Weakest Preconditions with Runtime Errors

With the final characterizer in place, we can now lift weakest preconditions to programs that may terminate either normally or at a failing assertion. The overall shape of the WP remains as before, but we must distinguish quantifiers over normal states from quantifiers over error states in Q.

Normal states are handled exactly as in Section 2.2, by expanding over the path pairs in Ψ and instantiating Q using the corresponding substitutions. For error paths, however, the relevant program state is not the final state at the end of C, but the state at the failing assertion[14]. This is precisely why the characterizer in Definition 2.3.1 additionally provides, for each assertion α, a snapshot Ψ_α of path conditions and substitutions *at the program point of* α. These snapshots contain all the information needed to reason about error states.

14: If the program contains multiple assertions, we a priori do not know which assertion is responsible for an error.

Definition 2.3.2 (Recursive path expansion for error-state quantifiers) *Let C be a loop-free program with assertions $A = \{\alpha_1, \ldots, \alpha_k\}$. For every $\alpha \in A$, let $\Psi_\alpha \in \mathcal{A}$ be the list of path pairs (b, θ) returned by the characterizer at α, where b is the path condition and θ is the substitution representing the execution prefix up to α. We extend the transformation $\mathsf{E}_\Psi[\cdot]$ to support* error-state quantifiers *by:*

$$\mathsf{E}_\Psi\big[\forall\langle\sigma\rangle_{er}.\, Q\big] \triangleq \forall\langle\sigma\rangle.\ \bigwedge_{\alpha\in A}\ \bigwedge_{(b,\theta)\in\Psi_\alpha} \Big(b(\sigma)\ \wedge\ \neg \mathit{cond}(\alpha)[\theta](\sigma)\ \Rightarrow\ \mathsf{E}_\Psi\big[\,Q[\theta]_\sigma\,\big]\Big),$$

$$\mathsf{E}_\Psi\big[\exists\langle\sigma\rangle_{er}.\, Q\big] \triangleq \exists\langle\sigma\rangle.\ \bigvee_{\alpha\in A}\ \bigvee_{(b,\theta)\in\Psi_\alpha} \Big(b(\sigma)\ \wedge\ \neg \mathit{cond}(\alpha)[\theta](\sigma)\ \wedge\ \mathsf{E}_\Psi\big[\,Q[\theta]_\sigma\,\big]\Big),$$

where $\mathit{cond}(\alpha)[\theta](\sigma)$ denotes the assertion condition of α instantiated according to θ and then interpreted in the state σ (i.e., program variables are read as state lookups in σ after applying θ).

As before, the transformation proceeds recursively: if the instantiated body contains further (error- or normal) state quantifiers, the corresponding expansion rule is applied again (cf. Definition 2.1.2).

Intuitively, the universal case requires safety against *all* assertion failures: for every initial state and every feasible prefix reaching an assertion, if that assertion fails, then the recursively expanded postcondition must hold. Dually, the existential case witnesses *some* failing run: there exists an initial state and a feasible prefix reaching an assertion that fails, such that the recursively expanded postcondition holds.

Example 2.3.2 We now return once more to the code snippet in Listing 2.4, this time to construct the weakest precondition from the characterizer $(\Psi, \mathcal{H}, \mathcal{A})$ derived in Example 2.3.1.

The method has the following postcondition[15]:

$$Q = \forall\langle\sigma\rangle_{er}.\ \sigma(e) = 0 \vee \sigma(e) = 1$$

Let's construct the weakest precondition using the transformation from Definition 2.3.2:

1. Since $\mathcal{A}$ gives us two assertions, α_1 and α_2, with each one path reaching it, we get two conjuncts:

$$\mathsf{E}_\Psi[Q] = \forall\langle\sigma\rangle.\ \Big(\big(b_1(\sigma)\wedge\neg \mathit{cond}(\alpha_1)[\theta_1](\sigma) \Rightarrow \mathsf{E}_\Psi\big[\,R[\theta_1]_\sigma\,\big]\big) \wedge \big(b_2(\sigma) \wedge \neg \mathit{cond}(\alpha_2)[\theta_2](\sigma) \Rightarrow \mathsf{E}_\Psi\big[\,R[\theta_2]_\sigma\,\big]\big)\Big).$$

15: Intuitively, this hyper-assertion requires that every possible error state that the program can reach has e equal to either 0 or 1.

where $R = \sigma(e) = 0 \vee \sigma(e) = 1$.

2. As R doesn't have any further (error- or normal) state quantifiers inside, the recursive call is trivial. We just need to instantiate the terms accordingly:

$$\mathsf{E}_\Psi[Q] = \forall\langle\sigma\rangle.\ \Big(\big(\top \wedge \neg(\sigma(x) \neq 0) \Rightarrow 0 = 0 \vee 0 = 1\big) \wedge \big(x \neq 0 \wedge \neg(\sigma(x) \neq 1) \Rightarrow 1 = 0 \vee 1 = 1\big)\Big).$$

We can see that the computed weakest precondition holds by itself, as both implications have a valid consequent. Therefore, this triple holds for any precondition.

With this extension, the split-free WP pipeline is complete: it uniformly handles deterministic code, nondeterministic assignments, and assertion-based error states. We can now construct a weakest precondition for any hyper-postcondition. In the next step, we will show how these WP obligations are encoded into SMT, which will allow us to finally verify our first programs in practice.

2.4 Encoding Verification Conditions in SMT

Our overall goal is to have full automation for the verification of hyper-triples. The remaining task to achieve this goal is to discharge the verification conditions constructed in the previous sections automatically. The correctness of a hyper-triple $\{P\}\ C\ \{Q\}$ reduces to checking whether the user-provided precondition entails our computed weakest precondition P_w, i.e., whether $P \models P_w$ holds.

This is done using an SMT solver. *Satisfiability Modulo Theories (SMT)* [9] is the problem of deciding whether a logical formula is satisfiable when interpreted over a background theory such as linear arithmetic, arrays, bit-vectors, uninterpreted functions, or combinations of these.

[9]: Nieuwenhuis et al. (2006), 'Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)'

Using SMT is a good fit for our verifier. Our pipeline produces verification conditions that are pure logical constraints over program states. Modern SMT solvers are designed exactly for deciding satisfiability of such formulas, so translating our constraints lets us reuse highly optimized and sophisticated solvers and leverage decades of progress in efficient, theory-aware satisfiability solving.

2.4.1 From Validity to Satisfiability

In general, an entailment $P \models P_w$ means: "In every model where P holds, P_w also holds." Equivalently, the implication $P \Rightarrow P_w$ is valid, i.e., it is true in all models. SMT solvers, however, are built and optimized to decide satisfiability (whether a given formula has a model), rather than validity (whether a formula holds in all models), which is the form of the question considered here.

To bridge this gap, we use a standard trick and make use of the fact that $P \Rightarrow P_w$ is valid, if and only if its negation is unsatisfiable. So, in practice, we query the solver with the negated formula $\neg(P \Rightarrow P_w)$. If the solver reports **unsat**, then there is no model where this negation holds. In consequence, our actual formula is valid and hence $P \models P_w$ is proven.

If the solver reports **sat**, it gives us a counterexample showing a model that satisfies $\neg(P \Rightarrow P_w)$. Hence, our actual formula cannot be valid, as its negation is true for at least one witness, and we must conclude that $P \models P_w$ does not hold.

One subtlety of this reduction is that SMT procedures are not complete in general. For many theories, validity is generally undecidable, so solvers implement *sound but incomplete* decision procedures: if they report back **unsat**, the formula is really unsatisfiable, but there may be formulas that are unsatisfiable (and hence entailments that are in fact valid) for which the solver cannot show this. In such cases, it may return **unknown** or simply time out[16].

16: We describe how our implementation handles these cases in Section **??**, especially when having multiple solver modes in place.

This incompleteness is not a problem for our approach, because it affects only *precision*, not *soundness*. We only ever accept a triple when the solver reports **unsat**. Assuming a sound translation to SMT and a sound SMT solver, our entailment check is sound as well: whenever it reports success, the reported correctness property holds.

2.4.2 General Encoding Strategy

Domains

Our SMT encoding closely follows the structure introduced in the original HHL paper [4]. We fix a single first-order sort[17] Int for all values (both program and logical), and represent *states* by a dedicated sort

$$\mathsf{State} = \mathsf{Array}(\mathsf{Int}, \mathsf{Int}),$$

i.e., a total map from *variable indices* (integers that identify program or logical variables) to their values. This implements the abstract notion of states in Definition 2 of the HHL paper.

[4]: Dardinier et al. (2024), 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'

17: A sort is a *type* that classifies the kinds of values or objects an SMT formula can refer to.

The *current set of normal states* that a hyper-assertion ranges over is concretized as an uninterpreted unary predicate

$$S : \mathsf{State} \to \mathsf{Bool},$$

and, following Hypra's extension of HHL with error states, the *current set of error states*[18] is represented as a second predicate

$$S_{err} : \mathsf{State} \to \mathsf{Bool}.$$

18: One might wonder, why we even care about error states at this point, as they neither occur in the precondition nor in the WP. We will soon see why we need an SMT encoding of error states when we introduce splits in Chapter 3.

Encoding of Hyper-Assertions

Given a syntactic hyper-assertion φ in the sense of Definition 9 of the HHL paper, our backend defines a translation $[\![\varphi]\!]$ into a first-order SMT formula over sorts Int and State, plus the predicates S and S_{err}.

State variables. Each state variable σ is represented by a first-order variable $\sigma : \mathsf{State}$, regardless of whether we treat it as a normal state or an error state.

Quantification over states. The HHL-style notations $\forall\langle\sigma\rangle$ and $\exists\langle\sigma\rangle$ are syntactic sugar for a quantified state-variable guarded by a state-existence atom:

$$\begin{aligned}\forall\langle\sigma\rangle.\ \varphi \quad &\equiv \quad \forall\sigma.\ \big(\langle\sigma\rangle \Rightarrow \varphi\big)\\ \exists\langle\sigma\rangle.\ \varphi \quad &\equiv \quad \exists\sigma.\ \big(\langle\sigma\rangle \wedge \varphi\big),\end{aligned}$$

where $\langle\sigma\rangle$ is a *state-existence atom*. It means "σ is one of the current normal states". In the SMT encoding, these atoms are interpreted as membership in the abstract sets of states:

$$[\![\langle\sigma\rangle]\!] = S(\sigma).$$

The same holds analogously for quantifiers over error states, using an *error-state-existence atom* which is interpreted in the SMT encoding as membership in the abstract error-state set S_{err}.

Remark 2.4.1 It is crucial to keep in mind that quantification over states is conceptually separate from state-existence. At the core logic level, we only have ordinary first-order quantifiers over state-typed variables:

$$\forall\sigma.\ \varphi \quad \text{and} \quad \exists\sigma.\ \varphi,$$

where σ is state-typed. The state-exists atoms $\langle\sigma\rangle$ and $\langle\sigma\rangle_{er}$ then restrict this variable to those states that belong to the current set, via the predicates S and S_{err}.

This separation of concepts is crucial for the encoding of state quantifiers, because it changes the set of states these quantifiers range over. Our encoding makes this dependence explicit via S and S_{err}:

$$\begin{aligned}[\![\forall\langle\sigma\rangle.\ \varphi]\!] &= \forall\sigma : \mathsf{State}.\ \big(S(\sigma) \Rightarrow [\![\varphi]\!]\big)\\ [\![\exists\langle\sigma\rangle.\ \varphi]\!] &= \exists\sigma : \mathsf{State}.\ \big(S(\sigma) \wedge [\![\varphi]\!]\big)\\ [\![\forall\langle\sigma\rangle_{er}.\ \varphi]\!] &= \forall\sigma : \mathsf{State}.\ \big(S_{err}(\sigma) \Rightarrow [\![\varphi]\!]\big)\\ [\![\exists\langle\sigma\rangle_{er}.\ \varphi]\!] &= \exists\sigma : \mathsf{State}.\ \big(S_{err}(\sigma) \wedge [\![\varphi]\!]\big)\end{aligned}$$

Reading a variable in a state. An atomic term $\sigma(x)$ denotes "the value of program variable x in state σ." As states (via the State sort) are represented as an array, we encode $\sigma(x)$ as an array lookup

$$[\![\sigma(x)]\!] = \mathsf{select}(\sigma, v_x),$$

where v_x : Int is a distinguished integer constant[19] for the variable name x. Logical variables are treated uniformly.

19: v_x is the integer index used to access x's value in the state array, so $\mathsf{select}(\sigma, v_x)$ retrieves exactly the slot corresponding to x.

First-order variables. A (logical) variable n (e.g., coming from a $\forall n$ or $\exists n$ assertion) is encoded as a standard SMT integer variable. We keep logical variables in the same store as program variables. That lets us treat them uniformly and results in a simpler encoding in practice.

Boolean structure. Conjunction, disjunction, implication, arithmetic comparisons, etc., are mapped homomorphically and in an intuitive,

structure-preserving way:

$$[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \wedge [\![\varphi_2]\!], \quad [\![\varphi_1 \leq \varphi_2]\!] = [\![\varphi_1]\!] \leq [\![\varphi_2]\!], \quad \text{etc.}$$

Summary

Taken together, these clauses give us a systematic translation from HHL hyper-assertions into concrete first-order SMT formulas. As an example of this translation, we give the encoding for the verification conditions generated for Listing 2.1 in Appendix A.3.

With that, we have completed the development of a fully syntactic verification pipeline for split-free programs and connected it to an automated backend. This gives us a solid and elegant core, which we can build on in the remaining thesis when we deal with more complex language constructs.

Handling Loops and Calls 3

Until now, we have established a core verification approach for split-free programs. This chapter now shifts focus and explains how we lift that core to the full input language by handling the remaining language constructs that are *not* split-free – most importantly **method calls and loops**. Both constructs break the "single-shot" characterizer/WP view, because we have to deal with additional specifications in the form of hyperproperties.

A method call is a modular boundary: to verify a caller, we rely on the callee's hyper-specification to summarize its behavior. In fact, in our setting, a method can even be treated as completely uninterpreted. Its behavior is then summarized solely by its pre- and postcondition, without ever requiring access to its implementation.

Likewise, a loop represents unbounded iteration, so any finite path enumeration by the characterizer is incomplete. To reason about all iterations at once, we need a hyper-invariant that summarizes the properties that hold before the loop and are preserved by every iteration.

Splitting the program into multiple triples is, therefore, a mechanism by which we can incorporate additional hyper-specifications into the proof. The result is a finite set of split-free obligations that can be discharged by our existing verification pipeline. Figure 3.1 summarizes this extended workflow. In the remainder of this chapter, we introduce the splitting schemas that realize this reduction.

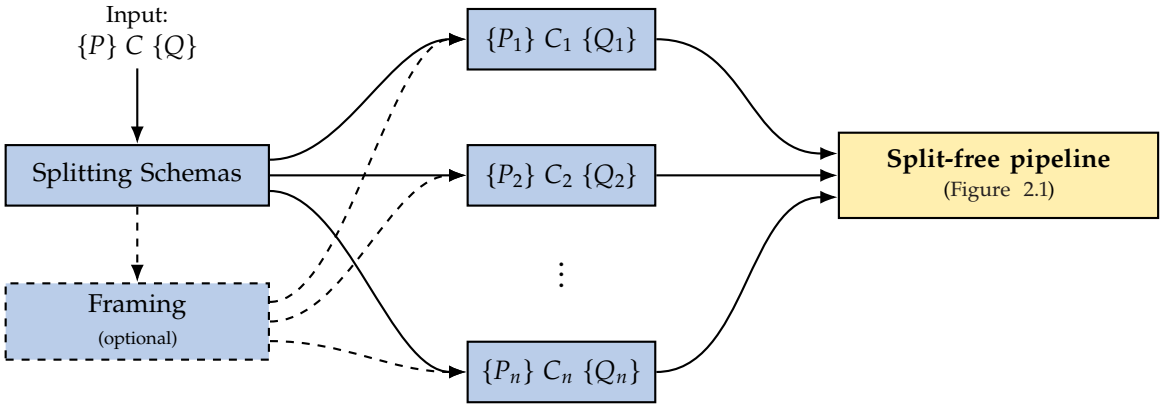


Figure 3.1: Extended verification pipeline: the splitting schemas produce multiple split-free verification triples. Framing can optionally strengthen them before each triple is discharged via the existing split-free pipeline.

3.1 The General Approach

In this section, we will introduce an algorithm that takes a triple $\{P\}\ C\ \{Q\}$ and produces a finite list of smaller split-free triples whose validity implies the validity of the original triple. In the case where C is split-free, it returns a singleton list containing the original triple.

The resulting list of triples is then checked by our existing split-free backend. This transformation into split-free triples is purely syntactic

as well: it recursively inspects the outermost construct of C and applies the corresponding splitting rule until there is nothing left to decompose. We will now dive into how this works and how the syntactic HHL rules are applied. For that, we will introduce decomposition schemas of the form

$$(P, C, Q) \rightsquigarrow \left[(P_1, C_1, Q_1), \ldots, (P_n, C_n, Q_n)\right]$$

for every complex language construct of our input language, possibly introducing side conditions for certain rules. Correctness of splitting is then captured by a single meta-property: if all generated sub-triples are valid, then the original triple is valid. This is based on the correctness of the HHL rules.

Remark 3.1.1 (Prefix-suffix convention) We will use the following structural convention throughout: every decomposition schema produces a list of triples that is ordered along the control-flow of the original program. In particular, the first triple always represents the *prefix* up to the first split point, and the last triple always represents the *suffix* from the last split point to the end of the program.

This convention will later allow us to refer to the postcondition of the first triple and the precondition of the last triple as the "interface" conditions at a split point.

In the next sections, we instantiate this general pattern for the two remaining split-triggering constructs of our language: loops and method calls. To keep the presentation focused, Section 3.2 and Section 3.3 first consider the common case where splits occur only along the main program path, i.e., without additional splits arising inside conditional branches. Section 3.5 then drops this restriction and explains how the procedure is extended to nested splits that do not lie on the main program path. Finally, Section 3.6 introduces framing, which ensures that relevant assumptions are propagated across the generated obligations in a controlled way.

```
C ::= skip
    | x := e
    | havoc x
    | C ; C
    | if (b) {C} else {C}
    | assume b
    | assert b
    | while [R] (b) I {C}

R ::= syncRule
    | syncTotRule
    | forAllExistsRule
    | existsRule
```

Figure 3.2: Grammar of the input language, now including `while` loops. The complete grammar can be found in Appendix A.1.

3.2 While-Loops

We begin by considering `while`-loops, as they are a canonical source for non-split-free control flow. We extend our input language and present the new grammar in Figure 3.2. In this section, we restrict attention to the setting in which loops appear only on the *main program path*. That means, we exclude nested loops[1] and loops inside conditional branches for now. These cases will be handled later on, when we talk about nested splits in Section 3.5.

Concretely, we cover the HHL rules for loops, as given in [10], and explain how each rule is realized and generates a finite set of verification obligations. Since different rules are appropriate for different loop shapes and specifications, we additionally describe how we integrate Hypra's existing rule-selection mechanism into the syntactic pipeline. Given a loop and its annotations, the selector chooses the applicable rule.

3.2.1 Splitting Schemas for While-Loops

We now make the connection between the logical loop-rules and our implementation. The underlying proof principles originate from the

1: We postpone nested loops as well because the WhileAuto-$\forall^*\exists^*$ rule generates obligations whose loop bodies occur inside conditionals. This leads to nested splits as well.

[10]: Dardinier (2025), 'Formal Foundations for Automated Deductive Verifiers'

HHL paper [4], which introduces several loop rules to reason about hyperproperties. These rules have been refactored to be more suitable for automation in the Hypra paper [5], and we use the latest version of the rules from [10], as shown in Figure 3.3.

We view each logical rule as a *splitting schema*: instead of applying the rule directly as proof rules in a derivation tree, we use it to *recursively* decompose a triple of the form shown in Listing 3.1 into a finite set of sub-triples whose validity implies the original one. Each generated sub-triple is then treated as a new splitting candidate and is further decomposed whenever applicable, until no splitting rules remain, and we can discharge them by the split-free backend.

In the implementation, four available loop rules can be used in annotations by the user: `syncRule`, `syncTotRule`, `forAllExistsRule`, and `existsRule`. Each of these names corresponds to one of the loop rules in Figure 3.3 and invokes the respective schema in the syntactic pipeline.

[4]: Dardinier et al. (2024), 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

Listing 3.1: A program skeleton with a **while**-loop and a *prefix* and a *suffix* that requires structural splitting.

```
method loopSplit()
    requires P
    ensures  Q
{
    // loop prefix
    while R (b)
         invariant I
        [decreases e]
    {
        // loop body
    }
    // loop suffix
}
```

WhileSync rule

For the `syncRule`, the pipeline treats the loop as a three-part decomposition around a user-provided loop invariant I, and generates exactly three verification triples – one for the code before the loop, one for the loop body, and one for the code after the loop.

Concretely, given a program of the form as in Listing 3.1 with `R = syncRule` , we first check the side condition $I \models \mathit{low}(b)$[2]. The rule is applicable only if this entailment holds. If this is the case, we generate the following triples:

1. Prefix triple (establishing the invariant): We verify that executing the loop prefix from the precondition P establishes our invariant I:

$$\{P\}\ \texttt{loop prefix}\ \{I\}$$

2. Body triple (preserving the invariant): We verify that a synchronized loop iteration preserves I, assuming that the invariant holds before the execution and that the guard is true across all states:

$$\{I \wedge \Box b\}\ \texttt{loop body}\ \{I\}$$

3. Suffix triple (loop exit): The loop may either terminate, then the invariant must still hold, and the guard must be false for all states, or the loop may diverge and not terminate, then there are no reachable states[3]. We ensure that executing the suffix from this point establishes the desired postcondition Q:

$$\{(I \vee \Box\bot) \wedge \Box(\neg b)\}\ \texttt{loop suffix}\ \{Q\}$$

Intuitively, these three obligations correspond to the standard invariant reasoning pattern: establishing I before the loop, showing that the loop body maintains I with every iteration, and finally combining I with the negated guard to continue with the remainder of the program.

2: Intuitively, this ensures that the loop guard has the same truth value in all currently considered executions, which enforces synchronized control flow (all executions either continue or exit together). The term is formally defined in the caption of Figure 3.3.

3: This is expressed with the $\Box\bot$ case in the rule.

WhileSyncTot rule

The `syncTotRule` follows the same three-triple decomposition as before, but strengthens the body obligation to additionally establish *termination*

4: This has to be specified via the **decreases** `e` clause in the input.

$$\textsc{WhileSync}\quad \frac{I \models \mathit{low}(b) \quad \models [I \wedge \Box b]\ C\ [I]}{\models [I]\ \textbf{while}\,(b)\,\{C\}\ \left[(I \vee \Box\bot) \wedge \Box(\neg b)\right]}$$

$$\textsc{WhileSyncTot}\quad \frac{I \models \mathit{low}(b) \quad \models_{\Downarrow}\ [I \wedge \Box(b \wedge e = t)]\ C\ [I \wedge \Box(e \prec t)] \quad t \notin \mathit{fv}(I) \cup \mathit{mod}(C)}{\models_{\Downarrow}\ [I]\ \textbf{while}\,(b)\,\{C\}\ [I \wedge \Box(\neg b)]}$$

$$\textsc{WhileAuto-}\forall^*\exists^*\quad \frac{\models [I]\ \textbf{if}\,(b)\,\{C\}\ [I] \quad \text{no } \forall\langle\cdot\rangle \text{ after any } \exists \text{ in } I}{\models [I]\ \textbf{while}\,(b)\,\{C\}\ \left[\Theta_{\neg b}(I) \wedge \Box(\neg b)\right]}$$

$$\textsc{WhileAuto-}\exists\quad \frac{\begin{array}{c}\forall v.\ \models \left[(\exists\langle\sigma\rangle.\, P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)) \wedge R\right]\ \textbf{if}\,(b)\,\{C\}\ \left[(\exists\langle\sigma\rangle.\, P_\sigma \wedge e(\sigma) \prec v) \wedge R\right] \\ \forall\sigma.\ \models [P_\sigma \wedge R]\ \textbf{while}\,(b)\,\{C\}\ [P_\sigma \wedge R]\end{array}}{\models \left[(\exists\langle\sigma\rangle.\, P_\sigma) \wedge R\right]\ \textbf{while}\,(b)\,\{C\}\ \left[(\exists\langle\sigma\rangle.\, P_\sigma) \wedge R \wedge \Box(\neg b)\right]}$$

Figure 3.3: The rules applied by Hypra to reason about while loops. In the rules WhileSyncTot and WhileAuto-$\exists$, the order $\prec$ must be *well-founded*. Moreover, $\mathit{low}(b) \triangleq \forall\langle\sigma\rangle\,\forall\langle\sigma'\rangle.\, b(\sigma) = b(\sigma')$ and $\Box b \triangleq \forall\langle\sigma\rangle.\, b(\sigma)$. Finally, $\models_{\Downarrow} [P]\ C\ [Q]$ corresponds to a terminating hyper-triple. Reproduced from [10], Fig. 6.7.

(total correctness) via a user-provided loop variant e[4]. As before, we consider the program structure in Listing 3.1 and require the same side condition $I \models \mathit{low}(b)$ to ensure the rule is applicable. In addition, we check that termination is handled for all nested loops in the body by checking whether they contain a `decreases` clause[5]. We generate the following triples:

5: Although we have excluded nested loops in this section, they will be addressed later on, where this check will become relevant.

1. Prefix triple (establishing the invariant):

$$\{P\}\ \texttt{loop prefix}\ \{I\}$$

2. Body triple (preserving the invariant and decreasing): We introduce a fresh logical variable t that represents the value of the loop variant e at the beginning of the body. We verify that the loop body preserves our invariant while also decreasing. We additionally require $t \geq 0$ after the body, ensuring that $\prec$ is well-founded:

$$\{I \wedge \Box(b \wedge e = t)\}\ \texttt{loop body}\ \{I \wedge \Box(0 \leq t < e)\}$$

3. Suffix triple (loop exit): On normal loop exit, we use the standard loop summary from before without the non-terminating case to verify the remaining code:

$$\{I \wedge \Box(\neg b)\}\ \texttt{loop suffix}\ \{Q\}$$

WhileAuto-$\forall^*\exists^*$ rule

For this and the next rule, we deal with the more general case where different executions might exit the loop at different iterations. The `forAllExistsRule` is an instance of such a *non-synchronized loop rule*. It is applicable when the loop invariant I has a $\forall^*\exists^*$ shape, i.e., there is no $\forall\langle\cdot\rangle$ after any $\exists$ in I.

Again, we consider programs of the form as shown in Listing 3.1, this

time with `R = forAllExistsRule`. Our pipeline yields the following decomposition:

1. Prefix triple (establishing the invariant):

$$\{P\}\ \texttt{loop prefix}\ \{I\}$$

2. Body triple (conditional invariant preservation): As we don't know which executions are still running the loop, and which have already exited, we now prove the preservation of I for a conditional statement. This allows the invariant I to refer to all executions, i.e., executions that are still running the loop, and executions that have already exited the loop [5]:

$$\{I\}\ \texttt{if (b) \{ loop body \}}\ \{I\}$$

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

3. Suffix triple (loop exit): The fact that different states may now exit the loop at different times, of course, also affects our proof for the loop suffix. Intuitively, existentially quantified state witnesses in I must remain admissible at loop exit:

$$\{\Theta_{\neg b}(I) \land \Box(\neg b)\}\ \texttt{loop suffix}\ \{Q\},$$

where we obtain the transformation $\Theta_{\neg b}(I)$, by recursively replacing all instances of $\exists\langle\sigma\rangle.\ P$ with $\exists\sigma.\ P \land (\neg b(\sigma) \Rightarrow \langle\sigma\rangle)$[6]. That is, it is ensured that the existentially-quantified states in I exist, but they only belong to the set of states after the loop if they satisfy $\neg b(\sigma)$, and hence have exited the loop.

6: Recall the discussion in Remark 2.4.1 where we distinguish quantification over states from state-existence.

WhileAuto-∃ rule

Finally, the `existsRule` is the most specialized of the four loop rules: it targets invariants with a top-level existential state quantifier and is also a *non-synchronized loop rule*. Given a program as in Listing 3.1 with `R = existsRule`, a loop invariant of the form $I = (\exists\langle\sigma\rangle.\ P_\sigma) \land R$[7] and a loop variant e, the pipeline proceeds as follows:

7: P_σ is a hyper-assertion which can mention σ as a "free" variable.

1. Prefix triple (establishing the invariant):

$$\{P\}\ \texttt{loop prefix}\ \{I\}$$

2. Body triple (preserving the invariant and decreasing): Similar to before, we have a triple to check whether I is preserved by a single loop iteration. Again, we introduce a fresh logical variable v. However, now we fix a "witness state" for which we prove termination in the loop[8]:

$$\{(\exists\langle\sigma\rangle.\ P_\sigma \land b(\sigma) \land v = e(\sigma)) \land R\}$$
$$\texttt{if (b) \{ loop body \}}$$
$$\{(\exists\langle\sigma\rangle.\ P_\sigma \land 0 \leq e(\sigma) < v) \land R\}$$

8: This is only an intuitive picture: in the formal proof obligations, the existentially quantified states in the pre- and postcondition may come from different executions.

3. Recursive triple (discharging the non-existential core): We strip I of its top-level existential quantifier[9] and generate a *recursive* loop obligation without it:

$$\{P_\sigma \land R\}\ \texttt{while (b) \{ loop body \}}\ \{P_\sigma \land R\}$$

9: Intuitively, this is sound because we have proven in the "body triple" that this particular state σ has exited the loop.

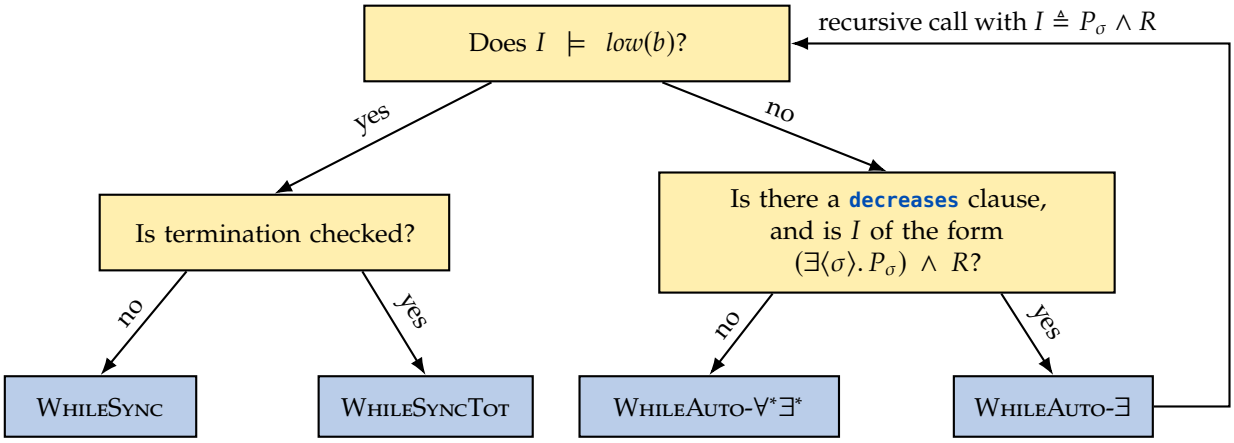


Figure 3.4: Loop-rule selection flowchart. Reproduced from [10], Fig. 6.9.

This sub-triple is handled by re-invoking the loop machinery on the same loop, but with one existential layer removed from the invariant[10].

4. Suffix triple (loop exit): Finally, once the loop has terminated, we continue with the remaining code under the loop-exit assumption.

$$\{I \wedge \square(\neg b)\}\ \texttt{loop suffix}\ \{Q\}$$

10: Note that, a priori, it is not clear which loop rule should be applied to this derived triple: removing the existential layer changes the shape of the invariant and may therefore alter the set of applicable loop rules. We rely on the automatic loop-rule selection mechanism (introduced in Subsection 3.2.2) to choose an appropriate rule for the resulting obligation.

Once generated, the obligations we get from these four loop rules fit the same downstream workflow as the rest of our pipeline: every resulting sub-triple is either split-free already or can be split up further. The loop rules serve as the bridge between unbounded iteration and the finite, syntactic verification conditions that our backend from Chapter 2 can already solve.

3.2.2 Loop Rule Selection

We have seen four different loop rules, each with different assumptions and a different scope. In practice, a verifier must decide which of the available loop rules to apply when encountering a loop in the program. Choosing an unsuitable rule either fails immediately or generates unnecessarily strong obligations.

The user can optionally annotate a loop in the source code with the rule that should be applied. However, requiring such annotations is not very convenient in practice: users should not have to study the technical preconditions of each rule just to verify a loop. Therefore, if no rule is specified by the user, our verifier has the option to automatically select a suitable rule.

We implement the same rule-selection algorithm as in the original Hypra, illustrated in Figure 3.4. Concretely, the selector inspects the loop condition and invariant and chooses an appropriate rule to apply. First, we check whether $I \models low(b)$ holds[11]. If so, we can apply one of the two synchronized loop rules, depending on whether a loop variant is provided by the user. As these two synchronized rules have weaker premises and stronger conclusions than the non-synchronized loop rules, they should always be preferred if they are applicable.

11: This is done using the SMT encoding from Section 2.4.

If $I \not\models low(b)$, we apply one of the two non-synchronized loop rules, depending on the shape of the loop invariant I and whether termination

is checked. If I is of the form $\forall^+\exists^*$, only WhileAuto-$\forall^*\exists^*$ is applicable, and similarly, we can only apply WhileAuto-$\exists$ if I is of the form $\exists^+\forall^*$. In the case when I has the shape $\exists^+$, both rules would be applicable. We then prefer WhileAuto-$\exists$, as it is more powerful.

One subtlety that stands out when looking at the diagram in Figure 3.4 is that we "check termination" for the WhileSyncTot rule, whereas for the WhileAuto-$\exists$ rule we only check whether the loop contains a decreases annotation. The reason is that WhileSyncTot is used to reason about the total correctness of the program. This is only meaningful if executing the loop body itself is guaranteed to terminate. Consequently, when selecting WhileSyncTot, we require termination to be checked for all sub-loops that may occur in the body. In contrast, WhileAuto-$\exists$ uses the variant to prove termination (and hence existence after the loop) for a particular "witness state". For rule selection, it is thus sufficient to require that the current loop provides a `decreases` clause, while nested termination does not need to be a prerequisite for applying WhileAuto-$\exists$.

3.2.3 Example

We conclude this section with a worked example that illustrates the loop-handling pipeline end to end. Starting from a single loop-containing triple, we show how the selector chooses an appropriate rule, and how this chosen rule decomposes the program into a set of split-free verification triples.

Consider the following code snippet C_{fib} in Listing 3.2. We want to prove *monotonicity* for the standard iterative Fibonacci loop: for any two runs, if one run starts with a bigger input n, the output *res* of this run won't be smaller than the output of the other run. Supplementary to the code of C_{fib} in Listing 3.2 we have the following specifications for the program:

$$
\begin{aligned}
P &\triangleq \forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(n) \geq \sigma_2(n) \\
Q &\triangleq \forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(a) \geq \sigma_2(a) \\
I &\triangleq \forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \\
&\qquad \big(\sigma_1(n) - \sigma_1(i) \geq \sigma_2(n) - \sigma_2(i) \land \\
&\qquad \sigma_1(a) \geq \sigma_2(a) \ \land\ \sigma_1(b) \geq \sigma_2(b)\big)
\end{aligned}
$$

Example 3.2.1 (Loop rule selection) As the loop in our input program contains no loop rule annotation, the verifier needs to select one of the four loop rules to use. This is done using the algorithm described in Subsection 3.2.2.

We can quickly see that the entailment $I \models \mathit{low}(b)$ doesn't hold, since I allows the two states it quantifies over to perform a different number of executions. Hence, we cannot use a synchronized loop rule to prove our program correct.

Instead, the selection procedure considers the automatic rules. As the loop doesn't carry a `decreases` clause and isn't of the required form for the rule WhileAuto-$\exists$, the algorithm falls back to the WhileAuto-$\forall^*\exists^*$ rule.

With the loop rule fixed, we can now focus on how we actually deconstruct our original triple into multiple smaller ones:

Listing 3.2: Iterative Fibonacci implementation with a relational pre- and post-condition expressing that the result is monotone in the input.

```
1  method fibonacci(n: Int, t:
     Int) returns (res: Int)
2      requires P
3      ensures  Q
4  {
5          var a: Int
6          var b: Int
7          var tmp: Int
8          var i: Int
9
10         a := 0
11         b := 1
12         i := 0
13
14         while (i < n)
15             invariant I
16         {
17             tmp := b
18             b := a + b
19             a := tmp
20             i := i + 1
21         }
22
23         res := a
24 }
```

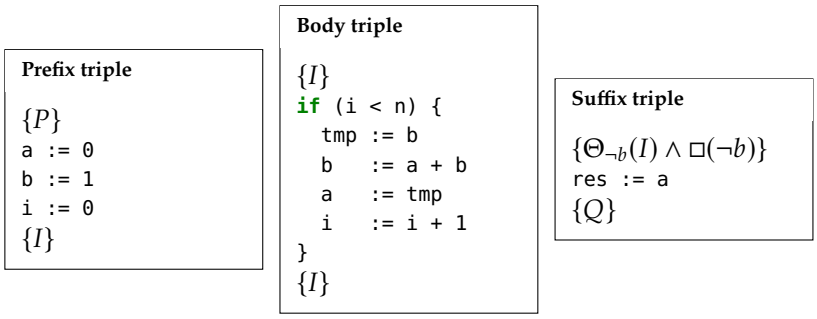


Figure 3.5: Splitting schema of the WhileAuto-$\forall^*\exists^*$ rule instantiated for the Fibonacci loop in Listing 3.2.

Example 3.2.2 (Splitting the triple) Having decided on the WhileAuto-$\forall^*\exists^*$ rule, the verifier now instantiates the rule's splitting schema for our Fibonacci method. The schema yields three verification triples, as shown in Figure 3.5.

It remains to derive the transformed invariant $\Theta_{\neg b}(I) \wedge \square(\neg b)$ that appears as a precondition in the suffix triple, as explained in Subsection 3.2.1: As I doesn't contain any existential state quantifier, $\Theta_{\neg b}$ behaves like the identity function on the invariant: $\Theta_{\neg b}(I) = I$. We just have to add the $\square(\neg b)$ conjunct:

$$\Theta_{\neg b}(I) \wedge \square(\neg b) \;=\; I \wedge \forall\langle\sigma\rangle.\, \neg(\sigma(i) < \sigma(n))$$

All three resulting triples are split-free and can therefore be passed directly to our split-free backend, which will verify them.

```
C ::= skip
    | x := e
    | havoc x
    | C ; C
    | if (b) {C} else {C}
    | assume b
    | assert b
    | while [R] (b) I {C}
    | m(a,b...)
    | x,y... := m(a,b...)
```

Figure 3.6: Grammar of the input language, now supporting method calls.

3.3 Method Calls

We now turn our attention to method calls, the second major language construct that requires splitting. A method call introduces a modular reasoning boundary: the effect of the callee is not derived by computing a weakest precondition over its body, but is summarized by its relational specification. In particular, we can use a method's behavior in proofs, even if it is treated as uninterpreted or its source code is not available.

To support method calls, we extend the grammar of our input language with two additional statements, as shown in Figure 3.6:

- `m(a_1, ..., a_n)` is a *procedure call statement*, which invokes a method `m` with n parameters and no return value.
- `x_1, ..., x_r := m(a_1, ..., a_n)` is a *multi-assignment call*, invoking a method `m` with n parameters and r return values. The return values are assigned to exactly m variables

The number of variables on the left-hand side must match the method's number of return values exactly. Otherwise, the program is ill-formed.

Conceptually, we split the surrounding program at the call site and use the callee's specification as the "glue" between the two resulting triples. Accordingly, our pipeline handles a call by decomposing the surrounding triple into obligations that

- establish the callee's required precondition at the call site, and
- use the callee's postcondition as an intermediate assertion to continue verifying the remainder of the program.

Listing 3.3: A program skeleton with a method call and a *prefix* and a *suffix* that requires structural splitting.

```
1  method callSplit()
2      requires P
3      ensures  Q
4  {
5      // call prefix
6      x1, x2 := m(a1, a2)
7      // call suffix
8  }
9
10 method m(b1: Int, b2: Int)
       returns (r1: Int, r2: Int)
11     requires P_m
12     ensures  Q_m
13 {
14     // method body (optional)
15 }
```

Listing 3.3 shows the general shape of a program fragment containing a method call. For concreteness (and spacing reasons), the call uses two actual arguments and two result variables, but the construction is completely generic. For a triple of the form shown in the `callSplit` method, we generate two triples:

1. Before-call triple (establishing the callee's precondition): We verify the prefix before the call under the original precondition, but we aim to establish the callee's precondition at the call site:

$$\{P\}\ \texttt{call prefix}\ \{P_m[\vec{b} \mapsto \vec{a}]\},$$

   where $\vec{b} \mapsto \vec{a}$ denotes substituting the method's formal parameters $\vec{b} = (b_1, \ldots, b_n)$ by the actual arguments used at the call site $\vec{a} = (a_1, \ldots, a_n)$.
2. After-call triple (continuing from the callee's postcondition): For the suffix, we assume that the call has established the method's hyper-postcondition[12] and use it as the new precondition for the remaining code:

$$\{Q_m[\vec{b} \mapsto \vec{a},\ \vec{r} \mapsto \vec{x}]\}\ \texttt{call suffix}\ \{Q\}.$$

   Here, we substitute both the formal parameters by the actual arguments like before, as well as the formal result variables $\vec{r} = (r_1, \ldots, r_m)$ by the caller-side result variables $\vec{x} = (x_1, \ldots, x_m)$ (for calls that return values). This ensures that the specification talks about the caller's program variables after the call, rather than the callee's formal names.

12: This use of Q_m is only sound if the callee indeed satisfies its specification. Depending on the setting, this can either be established by verifying the callee separately (when its body is available) or must be taken as a trusted assumption (e.g., for external or uninterpreted methods).

In other words, a method call is handled purely through specification instantiation: the call itself disappears from the verification problem and is replaced by the two obligations "show the callee may be called here" and "assume the callee's guarantee from here on." This is exactly what makes method calls a splitting construct in our setting.

Once generated, the two obligations produced by our method-call splitting schema fit the same downstream workflow as the rest of our pipeline: the before-call and after-call triples are either split-free already and can be discharged directly by our split-free backend, or, if they still contain loops, further calls, or other split-triggering constructs, can be decomposed further by recursively applying the corresponding splitting schemas.

3.4 Error Propagation Across Triples

Splitting the program into multiple triples not only decomposes normal behavior, but it also changes the way we have to reason about error states. Whenever we cut a program into pieces, we implicitly allow errors to occur in any of these fragments. If we then simply verified each triple in isolation, we would lose the information about where errors can occur in the program. We will show in the following small example that this issue can even make the overall approach unsound.

```
1 method errorPropagation(i: Int)
2     requires forall <_s> :: true          // P
3     ensures  forall <<_s>> :: _s[i] == 0  // Q
4 {
5     assert false
```

Listing 3.4: Counterexample showing that isolated checking of split triples is unsound when reasoning about error states.

```
6       foo()
7       assert i != 0
8   }
9
10  method foo()
11      requires forall <_s> :: true
12      ensures  forall <_s> :: true
13  { }
```

Example 3.4.1 (Unsoundness example) We illustrate the need for error propagation with a counterexample. Consider the program shown in Listing 3.4. We want to know whether the triple $\{P\}\ C\ \{Q\}$ of the method `errorPropagation` holds, where C denotes the method's body. The postcondition Q tries to assert that the value of i is 0 in all error-states.

One can easily see that this is not the case, and the triple is invalid. The `assert false` statement makes all initial states result in an error-state, no matter of their value for i. In particular, the precondition P doesn't impose any restrictions on i, so its value in an error state might be arbitrary.

But, what happens if we apply the splitting schema for method calls from Section 3.3 to the triple $\{P\}\ C\ \{Q\}$? We get two new triples:

1. $\{\forall\langle\sigma\rangle.\ \top\}$ `assert false` $\{\forall\langle\sigma\rangle.\ \top\}$
2. $\{\forall\langle\sigma\rangle.\ \top\}$ `assert i != 0` $\{\forall\langle\sigma\rangle_{er}.\ \sigma(i) = 0\}$

We compute the WP for each postcondition[13] and get the following entailments we need to check:

1. $\forall\langle\sigma\rangle.\ \top \quad\models\quad \forall\langle\sigma\rangle.\ \bot \Rightarrow \top$[14]
2. $\forall\langle\sigma\rangle.\ \top \quad\models\quad \forall\langle\sigma\rangle.\ \neg(\sigma(i) \neq 0) \Rightarrow \sigma(i) = 0$

Each of these entailments holds on its own, as the RHS trivially evaluates to $\top$ in both cases. Hence, this yields an apparent proof of $\{P\}\ C\ \{Q\}$ and our verifier would accept the triple, although we have seen that it is invalid.

The key observation from this example is that an early sub-triple may reach an error state due to a failing assertion, but this information is not reflected in the assumptions of later triples. This behavior is a direct consequence of the design choice that preconditions must not reason about error states[15]. We implicitly assume that the set of error states is empty at the beginning of a triple. So, even if a preceding sub-triple has already established that certain initial states inevitably lead to an error, this information is not carried over to subsequent triples. To avoid this, we need a systematic way of propagating error information across triples.

We handle this issue by keeping the original $\mathsf{E}_\Psi[\cdot]$ transformer unchanged for normal state quantifiers, but extend it for error-states, for which we additionally introduce a "carry" component. The purpose of this component is to ensure that earlier triples deal with the error-state clauses required by the postcondition. This propagates the error-relevant part of the postcondition backwards.

13: Using the framework discussed in Chapter 2.

14: We get this result by applying the $\mathsf{E}_\Psi[\cdot]$ transformation for normal state quantifiers, as defined in Definition 2.1.2. The $\bot$ comes from the assertion, which is added to the path condition.

15: This design choice is inherited from the original Hypra infrastructure on which our implementation is based.

Definition 3.4.1 (WP transformer with error propagation) *Let C be a loop-free program with assertions $A = \{\alpha_1, \ldots, \alpha_k\}$. For every $\alpha \in A$, let $\Psi_\alpha \in \mathcal{A}$ be the list of path pairs (b, θ) returned by the characterizer at α, where b is the path condition and θ is the substitution representing the execution prefix up to α.*

Formally, we introduce a variant $\mathsf{E}^{\uparrow}_{\Psi}[\cdot]$ that coincides with $\mathsf{E}_{\Psi}[\cdot]$ on all cases except error-state quantifiers, which are rewritten as:

$$\mathsf{E}^{\uparrow}_{\Psi}\big[\forall\langle\sigma\rangle_{er}.\, Q\big] \triangleq \left(\forall\langle\sigma\rangle.\, \bigwedge_{\alpha\in A}\ \bigwedge_{(b,\theta)\in\Psi_\alpha} \Big(b(\sigma) \wedge \neg[\alpha]_\theta(\sigma) \Rightarrow \mathsf{E}^{\uparrow}_{\Psi}\big[\, Q[\theta]_\sigma \,\big]\Big)\right) \wedge \forall\langle\sigma\rangle_{er}.\, Q,$$

$$\mathsf{E}^{\uparrow}_{\Psi}\big[\exists\langle\sigma\rangle_{er}.\, Q\big] \triangleq \left(\exists\langle\sigma\rangle.\, \bigvee_{\alpha\in A}\ \bigvee_{(b,\theta)\in\Psi_\alpha} \Big(b(\sigma) \wedge \neg[\alpha]_\theta(\sigma) \wedge \mathsf{E}^{\uparrow}_{\Psi}\big[\, Q[\theta]_\sigma \,\big]\Big)\right) \vee \exists\langle\sigma\rangle_{er}.\, Q,$$

where $[\alpha]_\theta(\sigma)$ denotes the assertion condition of α evaluated at the assertion point under the substitution θ (as provided by the characterizer), with program variables interpreted as lookups in the current state σ.

The first part of the extended $\mathsf{E}^{\uparrow}_{\Psi}[\cdot]$ tracks assertion failures arising within the current triple, as we know it from Section 2.3. We don't apply any substitution to the newly added "carry" conjunct/disjunct, because it is an annotation for errors happening in earlier triples.

Note how the WP resulting from the $\mathsf{E}^{\uparrow}_{\Psi}[\cdot]$ transformation now explicitly mentions error states. This is exactly where the SMT encoding for error-states from Section 2.4 becomes essential and explains why we have introduced it earlier[16].

16: Actually, when talking about method calls, an error-state could also be part of a method's postcondition that, after splitting, acts as the precondition for the subsequent triple.

In the overall splitting workflow, we apply this extended weakest-precondition construction to **every generated triple except the first one**. The first triple has no predecessor and therefore doesn't require the propagated error component.

Example 3.4.2 (Sound error propagation) Returning to the counterexample in Listing 3.4 from the beginning of this chapter, we can now illustrate how our refined transformation is no longer able to prove the triple. Again, we end up with the following split of the `errorPropagation` method:

1. $\{\forall\langle\sigma\rangle.\ \top\}$ `assert false` $\{\forall\langle\sigma\rangle.\ \top\}$
2. $\{\forall\langle\sigma\rangle.\ \top\}$ `assert i != 0` $\{\forall\langle\sigma\rangle_{er}.\ \sigma(i) = 0\}$

In the initial WP computation in Example 3.4.1, the error component was not propagated. Basically, everything error-related in the first triple was not taken into account in the final postcondition. This time, we compute the WP of the second postcondition using the new $\mathsf{E}^{\uparrow}_{\Psi}[\cdot]$ transformation, which yields the following entailments:

1. $\forall\langle\sigma\rangle.\ \top \quad\models\quad \forall\langle\sigma\rangle.\ \bot \Rightarrow \top$ *(same as before)*
2. $\forall\langle\sigma\rangle.\ \top \quad\models\quad \big(\forall\langle\sigma\rangle.\ \neg(\sigma(i) \neq 0) \Rightarrow \sigma(i) = 0\big) \wedge \forall\langle\sigma\rangle_{er}.\sigma(i)$

The second entailment now doesn't hold anymore, as nothing in the premise constrains the values of $\sigma(i)$ for error states. As a result, our verifier correctly rejects the triple.

While this example explains why the extended transformation is necessary for soundness, it does not yet demonstrate how it is useful in practice. We therefore introduce yet another illustrative example:

```
1 method errorLoop()
2     requires exists <_s> :: true    // P
3     ensures  exists <<_s>> :: true  // Q
```

Listing 3.5: Illustrative example used to demonstrate error propagation across split triples. The postcondition requires that there exists a reachable error state.

```
4  {
5          var i: Int
6          i := 0
7
8          while (i < 10)
9              invariant (exists <<_s>> :: true) ||
10                 (exists <_s> :: _s[i] == 0)
11             invariant forall <_s1>, <_s2> :: _s1[i] == _s2[i]
12             decreases 10 - i
13         {
14             assert false
15             i := i + 1
16         }
17 }
```

Intuitively, we can already see why propagation matters in Listing 3.5: the failing assert occurs inside the loop body and therefore in an earlier split obligation, not in the final suffix triple. Without propagating the error condition backwards, our verification engine would immediately reject the program, because there is no error occurring in the last triple. However, with propagation enabled, the existence of an error state established by any preceding triple can also be taken into account.

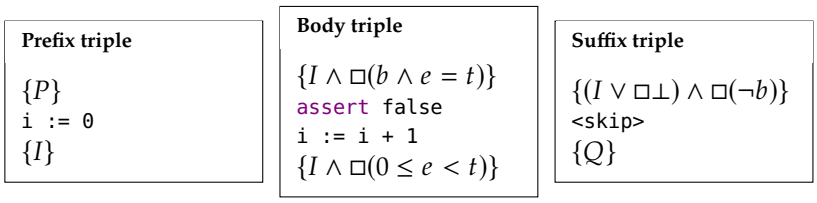


Figure 3.7: Splitting schema of the WhileSyncTot rule instantiated for Listing 3.5.

Example 3.4.3 (Error propagation) Consider the program shown in Listing 3.5. We want to know whether the triple $\{P\}\ C\ \{Q\}$ of the method `errorLoop` holds, where C denotes the method's body.

We begin by selecting an appropriate loop rule. Inspecting the loop guard, invariant, and variant yields the WhileSyncTot rule as the most suitable candidate by applying the algorithm from Subsection 3.2.2. Applying its splitting schema gives us the three verification triples shown in Figure 3.7, where b stands for the loop guard, I for the two `invariant` clauses, and e for the `decreases` clause.

For the *suffix triple* we compute the weakest precondition $\big(\exists\langle\sigma\rangle.\bot\big) \vee \big(\exists\langle\sigma\rangle_{er}.\top\big)$, as there is no assertion occurring in that part. As the loop invariant I also contains the clause $\exists\langle\sigma\rangle_{er}.\top$, the entailment holds, and we conceptually hand over the responsibility of finding the error-state witness to the previous triple.

For the *body triple*, the last conjunct of the obligations holds trivially because of the second invariant clause and the loop variant, which allow us to use the WhileSyncTot rule. We focus on the first part, which is relevant to our illustration. Applying $\mathsf{E}^{\uparrow}_{\Psi}[I]$ yields the following WP:

$$\big(\exists\langle\sigma\rangle.\ \top \wedge \neg\bot \wedge \top\big) \vee \big(\exists\langle\sigma\rangle_{er}.\ \top\big) \vee \big(\exists\langle\sigma\rangle.\top \wedge \sigma(i) + 1 = 0\big) \wedge \ldots,$$

where we omit the latter part for simplicity. It can be easily seen that this triple also holds because of the propagation of the error-relevant part of the invariant.

The *prefix triple* uses the normal $\mathsf{E}_\Psi[\cdot]$ propagation and trivially holds because of the $\exists\langle\sigma\rangle.\ \sigma(i) = 0$ conjunct in the invariant.

One final remark is in order: The loop rules we use were not designed or optimized to support error states. As a result, while the combination of splitting and error propagation is expected to remain sound and to work in principle, the required invariants can become noticeably more cumbersome when errors are involved.

3.5 Handling Nested Splits

So far, we have described splitting under a simplifying assumption: split-triggering constructs like loops and method calls occur only along the main program path, so that a single application of a splitting rule produces a clean, linear list of obligations.

In realistic programs, however, loops and method calls frequently appear *inside* conditionals[17]. Handling such nested splits is therefore essential for the completeness of our approach.

17: The most prominent example being recursive functions, which are often guarded by conditionals that determine whether the recursion continues or terminates.

This is a non-trivial goal because it forces the two central mechanisms of our verifier to interact: path reasoning and structural decomposition. On the one hand, splitting breaks a program into multiple triples that are verified independently. On the other hand, our weakest-precondition construction relies on a characterizer precisely to account for the fact that different quantified executions may follow different control-flow paths. We need a way to combine these two competing concepts and reflect path information at the level of hyper-assertions.

3.5.1 Motivating Example

To motivate the need for special treatment of nested splits, we begin with a small example in which our previous "linear" splitting intuition fails. The method `nestedComposition` in Listing 3.6 calls one of the two procedures `m1` or `m2` depending on a branch condition. Our goal is to construct a postcondition Q_{block} for the conditional statement that faithfully captures all possible outcomes of both branches.

```
method nestedComposition(x: Int, b: Int) returns (y: Int)
    requires forall <_s1>, <_s2> :: _s1[x] == _s2[x]
    ensures  forall <_s1>, <_s2> :: _s1[y] == _s2[y]
{
    if (b > 0) {
        y := m1(x)
    } else {
        y := m2(x)
    }
}

method m1(x: Int) returns (y: Int)
    requires forall <_s1>, <_s2> :: _s1[x] == _s2[x]
    ensures forall <_s1>, <_s2> :: _s1[y] == _s2[y]
{
    y := x + 1
}

method m2(x: Int) returns (y: Int)
    requires forall <_s1>, <_s2> :: _s1[x] == _s2[x]
```

Listing 3.6: Example used to demonstrate the need to account for mixed-branch executions when constructing postconditions for conditionals. The triple of the method `nestedComposition` does not hold.

```
21      ensures forall <_s1>, <_s2> :: _s1[y] == _s2[y]
22  {
23      y := x + 2
24  }
```

Example 3.5.1 (Unsound naïve strategy) We first show what happens if we apply our splitting construction to `nestedComposition` in the most straightforward way. That is, we treat the method calls inside the conditional exactly as we did for call sites on the main path and then recompose the two branches by guarding them with the branch condition[18]. For our example in Listing 3.6, the resulting naïve postcondition of the branch would be: $Q_{\text{block}} =$

$$\big(\forall\langle\sigma_1\rangle.\ \sigma_1(b) > 0 \Rightarrow \big(\forall\langle\sigma_2\rangle.\ \sigma_2(b) > 0 \Rightarrow \sigma_1(y) = \sigma_2(y)\big)\big) \wedge$$
$$\big(\forall\langle\sigma_1\rangle.\ \neg(\sigma_1(b) > 0) \Rightarrow \big(\forall\langle\sigma_2\rangle.\ \neg(\sigma_2(b) > 0) \Rightarrow \sigma_1(y) = \sigma_2(y)\big)\big)$$

At first glance, this looks exactly like what we want: the precondition enforces that all quantified executions agree on the input x, and both `m1` and `m2` preserve this agreement from x to their respective return values. With Q_{block} in place, we can seemingly prove the program, as it trivially entails the method postcondition.

18: This is conceptually similar to the characterizer approach in Section 2.1.

However, this naïve combination of the per-branch postconditions is fundamentally blind to the situation where different executions take different control-flow paths in the program, which is a core feature of Hyper Hoare Logic. As a result, the naïvely derived postcondition Q_{block} in Example 3.5.1 is unsound. Indeed, we must reject the program, because after the conditional, y may have different values in different executions, clearly violating the postcondition of `nestedComposition`.

To see what a sound treatment must do, we first show how a sound Q_{block} should look for the example in Listing 3.6.

Example 3.5.2 (Sound strategy via pigeonhole principle) We attempt the construction of Q_{block} again, now trying to avoid the implicit synchronization assumption from before. The guiding requirement is that Q_{block} must remain valid even when different executions take different branches. In particular, it must also cover the mixed case where one execution comes from `m1` and the other one from `m2`.

This time, we will do exactly that by making use of the pigeonhole principle. Concretely, in this example, we will quantify over an additional execution and promise a disjunction of equalities: Either σ_1 agrees with σ_2, or σ_1 agrees with σ_3, or σ_2 agrees with σ_3[19]. Formally, we get the following Q_{block}:

$$\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle, \langle\sigma_3\rangle.\ (\sigma_1(y) = \sigma_2(y)) \vee (\sigma_1(y) = \sigma_3(y)) \vee (\sigma_2(y) = \sigma_3(y))$$

This is deliberately weaker than the previous version from Example 3.5.1, and it basically tells us that among any three executions, at least two must agree on y. Regardless of how the three executions split across the branch, at least one of the three disjuncts in Q_{block} is guaranteed to hold. In the mixed case, if two executions take different branches, the third execution acts as a "witness" that necessarily shares a branch with one of the two. Q_{block} captures this fact without committing to which pair agrees, by leaving it as a disjunction.

19: This is exactly the pigeonhole argument: we consider the three executions $\sigma_1, \sigma_2, \sigma_3$, and we "place" them into two pigeonholes according to which of the two distinguished executions they agree with (namely the two quantified executions σ_1 or σ_2 from the branch-local postconditions).

```
method nestedSplit()
    requires P
    ensures  Q
{
    // if-else prefix
    if (b) {
        C_t^pre
        C_t^split // [P_t, Q_t]
        C_t^suf
    } else {
        C_e^pre
        C_e^split // [P_e, Q_e]
        C_e^suf
    }
    // if-else suffix
}
```

```
method nestedSplit()
    requires P
    ensures  Q
{
    // if-else prefix
    if (b) {
        C_t^pre
    } else {
        C_e^pre
    }
    if (b) {
        C_t^split // [P_t, Q_t]
    } else {
        C_e^split // [P_e, Q_e]
    }
    if (b) {
        C_t^suf
    } else {
        C_e^suf
    }
    // if-else suffix
}
```

Figure 3.8: A schematic illustration of how isolating the first split within a conditional works. The left program shows an if-else statement where both branches may contain splitting constructs. The right program shows the corresponding refactoring in which each branch is decomposed into prefix, first split, and suffix parts that are merged back into the surrounding context, as explained in Subsection 3.5.2.

In the remainder of the section, we generalize this idea and develop it as a systematic construction. We present a restricted, but practically very relevant variant that works for $\forall^*$-hyperproperties, i.e., hyper-assertions whose state quantification is purely universal. Whether and how it can be extended to a more general setting is an interesting question for future work.

The left code skeleton in Figure 3.8 shows the general shape of a program fragment containing a nested split. Conceptually, we lift the split points from the then- and else-branches onto the main verification path. When doing that, we need to consider all possible interleavings of executions across the branches.

3.5.2 Isolating the First Split in Each Branch

To keep the presentation concrete and to avoid being overwhelmed by multiple split points at once, we proceed in the smallest non-trivial step: we isolate the first splitting construct in each branch and treat everything around it as context.

Conceptually, we rewrite the program so that all split-free work that happens *before* the first split points in each branch is moved into a single prefix, and everything that happens *after* them is moved into a single suffix. This does not change the program's behavior, but merely makes the proof obligation explicit: the only remaining "unknown" statements inside the conditional are exactly the first split points we want to handle.

Let C_t be the body of the then-branch, and C_e be the body of the else-branch. We syntactically decompose them into

$$C_t \rightsquigarrow C_t^{\text{pre}};\ C_t^{\text{split}};\ C_t^{\text{suf}} \quad \text{and} \quad C_e \rightsquigarrow C_e^{\text{pre}};\ C_e^{\text{split}};\ C_e^{\text{suf}},$$

where C_t^{pre} / C_e^{pre} are split-free prefixes, C_t^{split} / C_e^{split} are the first statements in each branch that require splitting, and C_t^{suf} / C_e^{suf} are the

remaining suffixes which might contain other splitting statements. If one of the branches contains no split point, then C_t^{split} (resp. C_e^{split}) and the corresponding suffix is absent.

Using this decomposition, we construct a new program context as follows. We extend the surrounding `if-else prefix` part with the two branch prefixes, so that the conditional executes only up to (but not including) the first split point in each branch. Dually, we prepend the two branch suffixes into the surrounding `if-else suffix` so that everything that happens after the first split points will be handled recursively. The transformation is illustrated in Figure 3.8[20].

A crucial side condition for this decomposition to work is that the branch condition b must be stable[21]. Evaluating it before entering the branch must yield the same result as evaluating it directly after the branch. This is important because only then can we be sure that the same branch is selected in the new if-else blocks in the prefix and the suffix.

20: We show how the isolation procedure works in practice in the end-to-end example in Section 3.7, specifically in Listing 3.9.

21: This is easy to enforce in practice by first translating the program into SSA (static single-assignment) form.

3.5.3 Generating Verification Obligations

With the surrounding prefix and suffix factored out, we can now concentrate purely on the first split point in each branch. In particular, we dispatch to the appropriate schema for that split (the loop rules, the call rule, or – if the split itself is another conditional – the if-else rule applied recursively). Conceptually, each such schema yields[22]:

- a required precondition P_t (resp. P_e) that must hold at the split point such that the rule is applicable,
- a provided postcondition Q_t (resp. Q_e) summarizing what holds immediately after that split point, and
- any additional verification triples mandated by the applied schema (e.g., triples for loop bodies).

22: Strictly speaking, an application of a splitting schema yields a sequence of verification triples: by construction, the first triple is the *prefix triple* and the last triple is the *suffix triple* (cf. Remark 3.1.1). We therefore obtain P_t (resp. P_e) as the postcondition of the first triple, and Q_t (resp. Q_e) as the precondition of the last triple.

From these, we build a combined if-else-block precondition that ensures that the precondition in the respective branch holds in the executions that actually take that branch. This is done by guarding each branch requirement with the corresponding path condition. Assume the precondition P_t from the then-block is of the form

$$P_t = \forall\langle\sigma_1\rangle, \ldots, \langle\sigma_s\rangle.\ \xi_t(\sigma_1, \ldots, \sigma_s),$$

and the precondition P_e from the else-block is of the form

$$P_e = \forall\langle\sigma_1\rangle, \ldots, \langle\sigma_t\rangle.\ \xi_e(\sigma_1, \ldots, \sigma_t),$$

with $s, t \geq 1$ and ξ_t and ξ_e not containing any quantifications over states. Then, we define the combined if-else block precondition as

$$\begin{aligned} P_{\text{block}} \ \equiv\ & \Big(\forall\langle\sigma_1\rangle, \ldots, \langle\sigma_s\rangle.\ \big(b(\sigma_1) \wedge \cdots \wedge b(\sigma_s)\big) \Rightarrow \xi_t(\sigma_1, \ldots, \sigma_s)\Big) \\ \wedge\ & \Big(\forall\langle\sigma_1\rangle, \ldots, \langle\sigma_t\rangle.\ \big(\neg b(\sigma_1) \wedge \cdots \wedge \neg b(\sigma_t)\big) \Rightarrow \xi_e(\sigma_1, \ldots, \sigma_t)\Big). \end{aligned}$$

We will see how this construction works in practice when we work through an end-to-end example in Section 3.7.

Coming up with a postcondition is a more elaborate task because, unlike the precondition, it cannot assume a fixed control-flow choice. We must account for the fact that every execution may take either branch, and

hence all combinations of executions coming from Q_t and Q_e are possible. The overall postcondition, therefore, has to hold for every such mixture.

Our general construction for arbitrary $\forall^*$-postconditions follows the same idea as in the motivating example in Example 3.5.2. We quantify over enough executions so that, regardless of how they split between then- and else-branch, there are always sufficiently many executions from at least one branch to instantiate the corresponding branch-local postcondition.

Concretely, if the then-branch postcondition quantifies over n executions and the else-branch postcondition quantifies over m executions, then by taking $k = n + m - 1$ executions overall, we ensure that either at least n executions took the then-branch or at least m executions took the else-branch.

We now formalize how we systematically construct a postcondition for an if-else block that remains valid even when executions split across branches. Assume the postcondition Q_t from the then-block is of the form

$$Q_t = \forall \langle \sigma_1 \rangle, \ldots, \langle \sigma_n \rangle.\ \varphi_t(\sigma_1, \ldots, \sigma_n),$$

and the postcondition Q_e from the else-block is of the form

$$Q_e = \forall \langle \sigma_1 \rangle, \ldots, \langle \sigma_m \rangle.\ \varphi_e(\sigma_1, \ldots, \sigma_m),$$

with $n, m \geq 1$ and φ_t and φ_e not containing any quantifications over states. Then a sound postcondition Q_{block} for the whole if-else block covering all interleavings looks like this[23]:

$$Q_{\text{block}} = \forall \langle \sigma_1 \rangle, \ldots, \langle \sigma_k \rangle. \left(\bigvee_{\substack{T \subseteq [k] \\ |T| = n}} \varphi_t\big((\sigma_i)_{i \in T}\big) \ \vee \bigvee_{\substack{E \subseteq [k] \\ |E| = m}} \varphi_e\big((\sigma_i)_{i \in E}\big) \right)$$

where $k = n + m - 1$ and $[k] = \{1, \ldots, k\}$. The tuples $(\sigma_i)_{i \in U},\ U \in \{T, E\}$ are taken in increasing-index order, although any fixed canonical order would be fine.

23: Optionally, one can incorporate the branch condition into Q_{block} by guarding φ_t with the then-guard and φ_e with the negated guard, evaluated in the appropriate. This would yield an even stronger postcondition.

The disjunction reflects that we do not know a priori which executions took which branch. Instead, we merely assert that there exists a sufficiently large same-branch subset among the k executions to which one of the branch postconditions applies. Given an arbitrary pattern, we look at the set of indices that ended up in the then-branch and the set that ended up in the else-branch. Because we chose $k = n + m - 1$, one of these two sets must be large enough to instantiate the corresponding branch postcondition.

Proof. We want to show that Q_{block} is sound and covers all interleavings. We fix $k = n + m - 1$ arbitrary states $\sigma_1, \ldots, \sigma_k$. These represent the states of the whole if-else-statement postcondition. We also define the integers a and b as follows

$$a := |\{i \in [k] \mid \text{execution of } \sigma_i \text{ takes the then-branch}\}|$$
$$b := |\{i \in [k] \mid \text{execution of } \sigma_i \text{ takes the else-branch}\}|$$

by counting the number of states taking each branch. As every state takes exactly either the then-branch or the else-branch, we have: $a + b = k = n + m - 1$. By the pigeonhole principle, we have that either $a \geq n$ or $b \geq m$. We proceed with a case distinction:

Case $a \geq n$. We choose a subset $T \subseteq [k]$ with $|T| = n$, such that all states in the set $\{\sigma_i \mid i \in T\}$ take the then-branch. At least one such T must exist, because of how a is defined and the assumption $a \geq n$. Because of the assumption that Q_t holds and the fact that T is defined to only contain states taking the then-branch, $\varphi_t\left((\sigma_i)_{i\in T}\right)$ also holds. As Q_{block} contains this corresponding φ_t-disjunct by construction, Q_{block} holds.

Case $b \geq m$. Analogously to the first case we choose a subset $E \subseteq [k]$ with $|E| = m$, such that all states in the set $\{\sigma_i \mid i \in E\}$ take the else-branch. Again, at least one such E must exist. As Q_e holds per assumption and E only contains only states taking the else-branch, $\varphi_e\left((\sigma_i)_{i\in E}\right)$ also holds. Therefore, the whole disjunction holds, and with that also Q_{block}.

We have shown that Q_{block} holds in each of the above cases, and as the cases are exhaustive, we have shown that Q_{block} is a sound postcondition for the if-else block. □

Remark 3.5.1 (Tightness of the construction) If we reduce the number of universal quantifiers to $k' = n + m - 2$, the postcondition Q'_{block} could fail. We show this with a concrete counterexample:

As above, we fix k' states $\sigma_1, \ldots, \sigma_{k'}$, and pick $a' = n-1$ and $b' = m-1$, with a' and b' being the number of states that execute the then- or the else-branch, respectively. As $a' + b' = n + m - 2 = k'$, this is a valid choice. However, there is no n-subset with all states that took the then-branch and no m-subset with all states that took the else-branch, so every φ_t and φ_e disjunct is false and the whole Q'_{block} is falsified.

This shows that $k = n + m - 1$ is necessary for the postcondition to be useful.

If only one branch provides a universal postcondition, because there is no splitting construct in the other branch, we just keep it and guard it with the appropriate branch condition, analogously to how we did it for P_{block}.

3.5.4 Resulting Splitting Schema

With P_{block} and Q_{block} at hand, we now have everything needed to turn this decomposition into a concrete splitting schema. Concretely, given a program of the form as on the left-hand side in Figure 3.8, the handler returns a list of triples of the form:

1. Before-if triple:
$$\{P\}\ \texttt{newPrefix}\ \{P_{\text{block}}\},$$
where `newPrefix` is the program on the right-hand side in Figure 3.8, up to line 10.
2. Branch-specific obligations: For both branches, we propagate the triples emitted by the splitting schema handling C_t^{split} and C_e^{split}, e.g., the body triple for a loop, or no extra triple for a method call beyond what is already summarized by P_{block} and Q_{block}.
3. After-if triple:
$$\{Q_{\text{block}}\}\ \texttt{newSuffix}\ \{Q\},$$
where `newSuffix` is the program on the right-hand side in Figure 3.8, starting from line 16.

3.6 Automatic Framing

So far, our splitting rules reduce a verification problem to a collection of smaller triples with intermediate hyper-assertions provided by method specifications, loop invariants, or other obligations from verification rules. In practice, however, these intermediate assertions are often not sufficient on their own: when a program is cut into pieces, later triples may still rely on facts that were established earlier, even though these facts are not explicitly mentioned in their preconditions[24]. This is the purpose of **framing**: carrying along relevant assumptions across split boundaries, such that each triple has access to the information it needs for the verification to succeed.

24: Sometimes this context can be carried along manually, for instance by strengthening loop invariants, but doing so is typically cumbersome for the user. In other places, most notably in the specifications of called methods, it may not be possible to express the needed assumptions at all.

This section introduces the concept of **automatic framing** as the final component of our syntactic pipeline. We implement two lightweight syntactic heuristics that infer and propagate useful frame conditions without requiring additional user annotations. We start with a small motivating example that shows why framing matters.

```
method framing(x: Int) returns (y: Int)
    requires forall <_s1>, <_s2> :: _s1[x] == _s2[x]
    requires forall <_s> :: _s[x] >= 0
    ensures  forall <_s1>, <_s2> :: _s1[y] == _s2[y]
{
    y := 0
    while (y < x)
        invariant forall <_s> :: _s[y] <= _s[x]
    {
        y := y + 1
    }
}
```

Listing 3.7: Example that requires automatic framing to be successfully verified.

Example 3.6.1 Consider the code snippet in Listing 3.7. We will not go into the detailed verification process of the program, as this has been illustrated repeatedly in earlier examples.

However, this example nicely demonstrates why framing is crucial. The loop invariant $\forall\langle\sigma\rangle.\ \sigma(y) \leq \sigma(x)$ by itself is sufficient to relate y and x during the loop, but it does not by itself capture the facts that were available at method entry, most notably the fact that all states agree on x. In all splitting schemas introduced in Subsection 3.2.1, only the information provided by the loop invariant is preserved in subsequent triples.

In particular, the verifier needs access to the fact that x is stable across all states, even after the loop, as x is not modified. Only then, together with the loop invariant and the negated loop guard, can the postcondition be proven.

We implement automatic framing in two phases that enrich the preconditions of the generated split triples, whereas program statements and postconditions of the triples stay unchanged.

3.6.1 Stable-Precondition Propagation

We treat suitable preconditions as method-wide invariants and propagate them forward across subsequent triples as long as they cannot have been invalidated. Concretely, we traverse all triples $\{P_i\}\ C_i\ \{Q_i\}$ in program

order and keep a *set of active assumptions* Γ, which are considered stable up to the current triple. We then strengthen the precondition of the current triple by adding these assumptions:

$$P_i' \triangleq P_i \cup \Gamma.$$

After processing the triple, we update Γ for the next triple by adding new candidates from P_i and removing any assumption that may have been invalidated by C_i. To do so, we use two conservative criteria:

- **No existential state quantification.** If an assumption contains an existential quantifier over states, it may rely on the existence of states *after* some code fragment previously encountered. However, their existence is not guaranteed across a non-terminating statement, so we do not propagate such assumptions[25].
- **Not mentioning modified variables.** If an assumption in Γ or P_i refers to a program variable that is modified by the current statement, we conservatively treat it as potentially invalidated and stop propagating it.

25: This restriction is deliberately conservative. In Subsection 5.2.4, we outline how the heuristic could be relaxed.

Example 3.6.2 We come back to the example from Listing 3.7 and show how it becomes provable once *stable-precondition propagation* is in place. Recall that in Example 3.6.1 we had the issue of not having enough information about x after the loop invariant to prove the postcondition.

This time, our propagation addresses this missing link: as x is not modified in the program, the missing information can be propagated, and we get the following precondition for the final triple:

$$\begin{aligned} &\forall\langle\sigma\rangle.\ \sigma(y) \le \sigma(x) \\ \land\ &\forall\langle\sigma\rangle.\ \neg(\sigma(y) < \sigma(x)) \\ \land\ &\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x) \\ \land\ &\forall\langle\sigma\rangle.\ \sigma(x) \ge 0, \end{aligned}$$

where the first two conjuncts come from the WhileAuto-$\forall^*\exists^*$ rule and let us conclude that $y = x$ in all states. The latter two conjuncts are propagated from the precondition and ensure that all quantified states still agree on x, which is exactly the missing ingredient to conclude the postcondition.

3.6.2 Alias-Based Framing

The second heuristic addresses a different idea: if earlier code has established that a program variable x is equal to some expression e (possibly under a path condition), then later triples that reason about x or e should be allowed to assume that relationship[26].

26: We will see in Section 3.7 why this second heuristic is important in practice, when we go through an end-to-end example.

Formally, as we traverse the generated triples in program order, we maintain an *active alias store* Δ. Each entry in Δ records a variable together with a guarded right-hand side:

$$x \mapsto (pc, e),$$

intended to mean: on executions satisfying the path condition pc, the value of x corresponds to the value of e at the current program point. For each triple (P_i, C_i, Q_i), we then proceed in two steps:

1. **Select relevant aliases.** We determine which previously recorded aliases are relevant for the current triple, namely those whose left-hand side x occurs in P_i or Q_i. Let $\Delta_i \subseteq \Delta$ be this filtered set.
2. **Add them as frame assumptions.** For every $(x \mapsto (pc, e)) \in \Delta_i$, we append a universally quantified frame condition to the precondition stating that in any reachable state, if pc holds, then $x = e$ holds in that state:

$$P_i' \triangleq P_i \cup \left\{ \forall \langle \sigma \rangle.\, pc(\sigma) \Rightarrow \sigma(x) = e(\sigma) \;\middle|\; (x \mapsto (pc, e)) \in \Delta_i \right\}.$$

This makes the earlier assignment information available to the solver exactly where it might matter[27]. After framing triple i, we update Δ by extracting the new aliases introduced by its statement S_i using the characterizer. To avoid unsound inferences, we do not record aliases for variables that may be assigned in ways that are not simple syntactic equalities, most notably variables assigned nondeterministically or through the result of method calls.

27: Which we don't know a priori, since our analysis is purely syntactic.

3.7 Case Study: Recursive Fibonacci

We conclude this chapter with a larger end-to-end example: the recursive Fibonacci implementation, for which we want to prove monotonicity. The source code is given in Listing 3.8. We will exercise the full machinery developed throughout the chapter. Its proof crucially relies on *method calls* (Section 3.3) to reason modularly about recursive invocations, *nested splitting* (Section 3.5) because the recursive calls are guarded by conditionals, and *automatic framing* (Section 3.6) to retain facts across recursive calls. As such, Fibonacci serves as a compact "stress test" and, at the same time, a showcase of how the individual ingredients fit together.

```
method fibonacci(n: Int, t: Int) returns (r: Int)
    requires forall <_s1>, <_s2> :: _s1[t] == 1 && _s2[t] == 2 ==> _s1[n] >= _s2[n]
    ensures  forall <_s1>, <_s2> :: _s1[t] == 1 && _s2[t] == 2 ==> _s1[r] >= _s2[r]
{
    var tmp1: Int
    var tmp2: Int
    var res1: Int
    var res2: Int

    tmp1 := n - 1
    tmp2 := n - 2

    if (n <= 1) {
        r := n
    } else {
        res1 := fibonacci(tmp1, t)
        res2 := fibonacci(tmp2, t)

        r := res1 + res2
    }
    assume res1 + res2 >= 1
}
```

Listing 3.8: Recursive Fibonacci. The specification states that whenever two different initial states start with $\sigma(n) \geq \sigma'(n)$, then the corresponding final states satisfy $\sigma(r) \geq \sigma'(r)$, i.e., Fibonacci is monotone in its input. The parameter t serves purely as a logical tag to distinguish executions.

The final `assume` is an additional proof assumption needed to discharge the last split-free triple, as we will see later, but it doesn't change the semantics of the program[28].

28: This reveals a current limitation of our approach. We explain later in the example why this assumption is required. Eliminating such extra assumptions is left as future work.

In the following, we walk through the verification pipeline instantiated for this example step by step. Rather than explicitly constructing weakest preconditions as we have done for multiple examples in Chapter 2, we focus on the extended pipeline developed in this chapter, and give intuition on why each resulting split-free triple is valid.

Our first goal is therefore to decompose the program into a list of split-free verification triples, each of which can then be discharged by the split-free pipeline from Chapter 2. Inspecting the body, the first construct that forces a split is the recursive call on line 16. Since this call occurs inside the else-branch of the conditional, we must apply our nested splitting procedure.

To do so, we first isolate this method call as described in Subsection 3.5.2. The resulting body of the function is shown in Listing 3.9. Now, we can finally apply the splitting schema for nested splits. We are dealing with a method call, so we instantiate the method-call schema from Section 3.3 as the split step inside the nested-splitting procedure.

Listing 3.9: Method body of recursive Fibonacci in Listing 3.8, after applying the decomposition from Subsection 3.5.2.

```
1  // --[start of triple 1.1]--
2  var tmp1: Int
3  var tmp2: Int
4  var res1: Int
5  var res2: Int
6
7  tmp1 := n - 1
8  tmp2 := n - 2
9
10 if (n <= 1) {
11    r := n
12 } else {
13    <skip>
14 }
15 // --[end of triple 1.1]--
16 if (n <= 1) {
17    <skip>
18 } else {
19    res1 := fibonacci(tmp1, t)
20 }
21 // --[start of triple 1.2]--
22 if (n <= 1) {
23    <skip>
24 } else {
25    res2 := fibonacci(tmp2, t)
26    r := res1 + res2
27 }
28 assume res1 + res2 >= 1
29 // --[end of triple 1.2]--
```

Since there is only a method call in the else-branch (line 19 in Listing 3.9), nested splitting becomes particularly simple: we keep the call's resulting pre-and postcondition obligations, and guard them with the corresponding branch condition $\neg(n \leq 1)$, instantiated for each state. This results in two triples, whose code fragments are highlighted in Listing 3.9. The pre- and postconditions for these triples are:

- **Triple 1.1:**

$$P_1 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\sigma_1(t) = 1 \land \sigma_2(t) = 2\big) \Rightarrow \sigma_1(n) \geq \sigma_2(n)$$
$$Q_1 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(tmp1) \geq \sigma_2(tmp1)\big)\big)$$

- **Triple 1.2:**

$$\text{Pre: } \big(\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(res1) \geq \sigma_2(res1)\big)\big)\big)$$
$$\text{Post: } \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\sigma_1(t) = 1 \land \sigma_2(t) = 2\big) \Rightarrow \sigma_1(r) \geq \sigma_2(r)$$

Inspecting these two resulting triples, we observe that the first one is already split-free. The second triple, however, still contains the remaining recursive call (see line 25 in Listing 3.9) and is therefore not yet in our required form. Consequently, we must apply the splitting pipeline once more. As before, the call appears only under the else-branch of a conditional, so we invoke the nested-splitting procedure again to isolate the call and generate the corresponding proof obligations.

Triple 2.1

$\{P_1\}$
```
tmp1 := n - 1
tmp2 := n - 2
if (n <= 1) {
   r := n
} else {
   <skip>
}
```
$\{Q_1\}$

Triple 2.2

$\{P_2\}$
```
<skip>
```
$\{Q_2\}$

Triple 2.3

$\{P_3\}$
```
if (n <= 1) {
   <skip>
} else {
   r := res1+res2
}
assume res1 + res2
    >= 1
```
$\{Q_3\}$

Figure 3.9: Final split-free decomposition of `fibonacci` after the second nested split. Variable declarations are omitted for readability.

The resulting split is shown in Figure 3.9. Here, Triple 2.1 coincides with Triple 1.1, which was already split-free, while Triples 2.2 and 2.3 arise from the second decomposition step applied to Triple 1.2. The corresponding pre- and postconditions of each triple are given below. The only changes originating from the second iteration can be seen in Q_2 and P_3, which incorporate the obligations from the second call:

- **Triple 2.1:** *(same as Triple 1.1)*

$$P_1 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\sigma_1(t) = 1 \land \sigma_2(t) = 2\big) \Rightarrow \sigma_1(n) \geq \sigma_2(n)$$
$$Q_1 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(\mathit{tmp1}) \geq \sigma_2(\mathit{tmp1})\big)\big)$$

- **Triple 2.2:**

$$P_2 = \big(\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(\mathit{res1}) \geq \sigma_2(\mathit{res1})\big)\big)\big)$$
$$Q_2 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(\mathit{tmp2}) \geq \sigma_2(\mathit{tmp2})\big)\big)$$

- **Triple 2.3:**

$$P_3 = \big(\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n) \leq 1) \land \neg(\sigma_2(n) \leq 1)\big) \Rightarrow \big((\sigma_1(t) = 1 \land \sigma_2(t) = 2) \Rightarrow \sigma_1(\mathit{res2}) \geq \sigma_2(\mathit{res2})\big)\big)\big)$$
$$Q_3 = \forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\sigma_1(t) = 1 \land \sigma_2(t) = 2\big) \Rightarrow \sigma_1(r) \geq \sigma_2(r)$$

At this point, it may look as if we were done, since all resulting obligations are now split-free. Triple 2.1 is indeed easy to prove: the characterizer supplies the substitution `tmp1 = n-1`, and after computing the weakest precondition with this substitution, the remaining entailment is straightforward.

However, in this form, we still cannot discharge Triples 2.2 and 2.3: the obligations do not provide a sufficiently strong link between the intermediate results and their respective inputs. Most notably, we cannot relate `res1` to `tmp2`, nor `res2` to the final return value `r`. As a consequence, the monotonicity argument cannot be completed from these triples alone.

The missing connections above are not caused by an incorrect split, but by information loss: properties established in earlier triples are not automatically available in later triples. This is where automatic framing comes in. By applying the heuristics from Section 3.6, we can strengthen P_2 and P_3 with the following conjuncts:

- P_2 gets additionally:

$$\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \big(\sigma_1(t) = 1 \land \sigma_2(t) = 2\big) \Rightarrow \sigma_1(n) \geq \sigma_2(n),$$
$$\forall\langle\sigma\rangle.\ \sigma(\mathit{tmp2}) = \sigma(n) - 2,$$

  where the last assertion is a result of *alias-based framing*, and the first one is propagated from the method precondition.

- P_3 gets additionally:

$$\begin{aligned}
&\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \big(\big(\neg(\sigma_1(n)\le 1)\wedge\neg(\sigma_2(n)\le 1)\big)\Rightarrow\\
&\qquad\big((\sigma_1(t)=1\wedge\sigma_2(t)=2)\Rightarrow\sigma_1(\mathit{res1})\ge\sigma_2(\mathit{res1})\big)\big),\\
&\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \big(\sigma_1(t)=1\wedge\sigma_2(t)=2\big)\Rightarrow\sigma_1(n)\ge\sigma_2(n),\\
&\forall\langle\sigma\rangle.\ \sigma(n)\le 1\Rightarrow\sigma(r)=\sigma(n),
\end{aligned}$$

  where the last assertion again arises from *alias-based framing*, while the first one is propagated from P_1, and the second one from P_2.

With framing in place, all remaining triples become provable: the framed obligations now carry exactly the "linking" information that was previously lost across the splits. Triple 2.2 is immediate: alias-based framing preserves the defining relationship between `tmp2` and `n` across the split, which enables us to prove it successfully. As the triple doesn't contain any statements, the weakest precondition coincides with Q_2.

Reasoning correctness for Triple 2.3 requires a bit more effort. With the additional `assume` statement, we get the following WP:

$$\begin{aligned}
\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ &\Big((G_1\wedge S_1)\Rightarrow\Big((G_2\wedge S_2)\Rightarrow(\tau\Rightarrow\sigma_1(r)\ge\sigma_2(r))\Big)\quad\text{(BB)}\\
&\wedge(G_1\wedge S_1)\Rightarrow\Big((G_2\wedge\neg S_2)\Rightarrow(\tau\Rightarrow\sigma_1(r)\ge R_2)\Big)\quad\text{(BR)}\\
&\wedge(G_1\wedge\neg S_1)\Rightarrow\Big((G_2\wedge S_2)\Rightarrow(\tau\Rightarrow R_1\ge\sigma_2(r))\Big)\quad\text{(RB)}\\
&\wedge(G_1\wedge\neg S_1)\Rightarrow\Big((G_2\wedge\neg S_2)\Rightarrow(\tau\Rightarrow R_1\ge R_2)\Big)\Big),\quad\text{(RR)}
\end{aligned}$$

where we have conveniently used the following abbreviations:

$$\begin{aligned}
R_i &\triangleq \sigma_i(\mathit{res1})+\sigma_i(\mathit{res2}), \qquad (i\in\{1,2\})\\
G_i &\triangleq (R_i\ge 1),\\
S_i &\triangleq (\sigma_i(n)\le 1),\\
\tau &\triangleq (\sigma_1(t)=1\wedge\sigma_2(t)=2).
\end{aligned}$$

We now show for each of the four conjuncts in the WP, why it is entailed by the framed P_3:

(BB): Both σ_1 and σ_2 are in the base case, so we can make use of the $r=n$ identity, which we get from alias-based framing, and the entailment is straightforward.

(BR): σ_1 is in the base case, σ_2 in the recursive case. The propagated precondition asserting monotonicity of n tells us $\sigma_1(n)\ge\sigma_2(n)$, but at the same time $\sigma_2(n)>1$ (recursive case) and $\sigma_1(n)\le 1$ (base case) – a contradiction.

(RB): This is the tricky case, where our additional assumption comes in. Through alias-based framing, we can make use of the identity $\sigma_2(r)=\sigma_2(n)$, and as σ_2 is in the base case, we can conclude $\sigma_2(r)\le 1$. With our extra assumption[29], we get $R_i\ge 1\ge\sigma_2(r)$, which makes the entailment immediate.

(RR): From the framed assertions, we get monotonicity for *res1* and *res2*, from which the entailment directly follows.

As all generated split-free triples hold, the original method is verified. This Fibonacci case study ties the chapter together: it shows, in a single proof, how nested splitting, modular method-call reasoning, and automatic framing interact to reduce a nontrivial recursive program to a collection of obligations that our backend can discharge routinely.

29: Now, we see why the extra assumption is needed. Notice that P_3 never relates *res1* + *res2* (for $n>1$) to r. It only gives us any information about *res1* and *res2*, when both states satisfy $n>1$. An idea would be to strengthen P_3 by extending framing to also record some base-case equalities for the two variables (e.g. *res1* = 1 and *res2* = 0), but this still does not provide the missing *cross-branch* relationship needed here.

Implementation and Evaluation 4

In the previous chapters, we have developed the conceptual and logical foundations of our syntactic verification pipeline. With these building blocks in place, we now shift focus to the concrete implementation.

This chapter describes the implementation of the approach and presents an empirical evaluation. We first outline the architecture and discuss how each component is realized in code. We then turn to a detailed evaluation, with a particular emphasis on comparing the new syntactic pipeline against Hypra's existing semantic backend. Overall, the results show that our implementation achieves substantial speedups while maintaining a high verification success rate, indicating that it is a practical alternative to Hypra's current approach.

4.1 Implementation

The syntactic verifier is implemented as an alternative backend of Hypra rather than a separate tool. We integrate it directly into the existing architecture. Since Hypra is implemented in Scala 2.13.10 [11], we implement our backend in the same language to ensure seamless integration. It operates on the same input language, reuses the existing parsing, symbol-checking, and type-checking infrastructure, and produces the same kind of overall verification verdict. When running Hypra, the syntactic backend is activated via the command-line option `--syntactic`.

[11]: Odersky et al. (2006), *An Overview of the Scala Programming Language*

From an engineering perspective, this design choice has two advantages. First, it minimizes duplication: by reusing Hypra's existing input syntax, parser, and AST, the syntactic backend can focus entirely on the new verification pipeline rather than re-implementing front-end functionality. Second, it makes the comparison in the evaluation meaningful and fair: any differences in behaviour or performance can be attributed to the backends themselves, rather than to differences in pre-processing or intermediate representations. The remainder of this section outlines the main components of the syntactic backend and how they interact within Hypra's architecture.

4.1.1 Module Structure

We now give a structural overview of the syntactic backend and how its components interact with each other. Figure 4.1 provides a high-level flow-chart of our implementation. It is organized in modules that correspond directly to the steps in the pipeline introduced in Chapter 2 and Chapter 3. Since these chapters already introduced the underlying logic in great detail, we do not repeat the conceptual material here. Instead, this section focuses on the implementation-specific aspects.

The flow starts from an input program, which is already pre-processed by Hypra's existing infrastructure. The program is parsed, type-checked, and symbol-checked, and is then provided to us as an `HHLProgram`, Hypra's AST-level representation. The `SyntacticEngine` module is the main entry

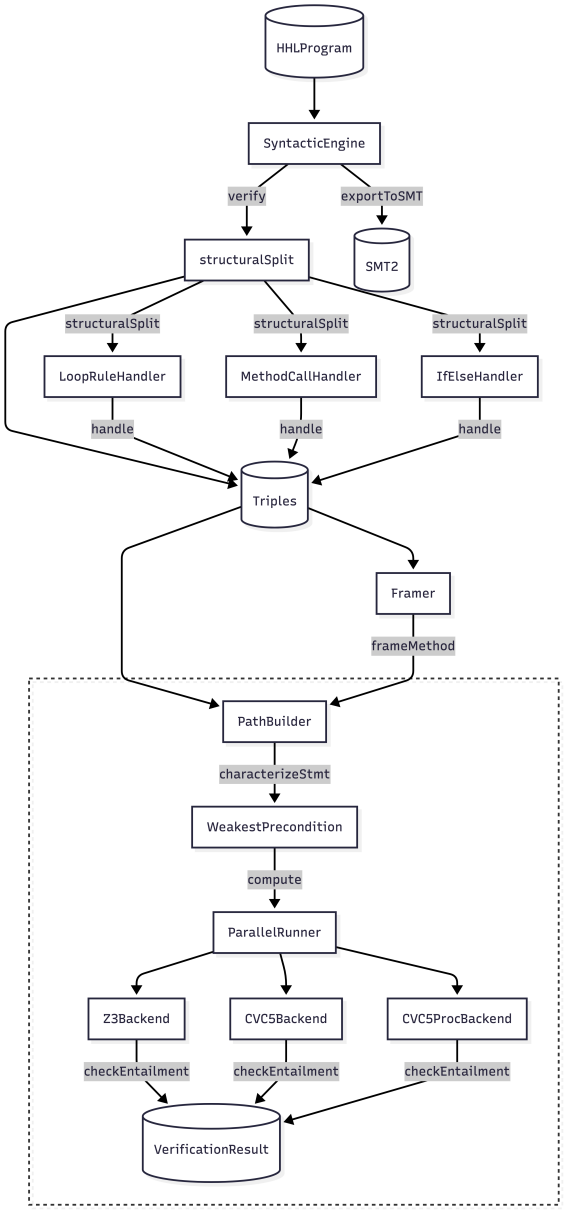


Figure 4.1: High-level flow-chart of the syntactic backend in Hypra (enabled via the `--syntactic` option). The diagram summarizes the end-to-end pipeline, from receiving a `HHLProgram` (AST representation), through obligation generation by structural splitting and WP construction, to discharging the resulting obligations to one (or multiple) SMT solvers. It gives a high-level overview of how the different modules of the implementation interact with each other. Every node in the chart corresponds to a module, and edge labels denote method calls of the preceding module.

and orchestration point for verification. It first performs structural splitting by delegating split points to specialized handlers for each language construct: a dedicated `LoopRuleHandler` for each of the four loop rules, the `MethodCallHandler` for calls, and the `IfElseHandler` for splits inside conditionals. These handlers collectively generate a list of split-free verification triples. Before discharging these obligations, the `Framer` optionally enriches the triples with additional framing assumptions. This can be deactivated via the `--noframe` command-line option.

The lower part of the flow-chart, shown in the dashed rectangle, represents the split-free pipeline from Chapter 2. Accordingly, for each of the triples, the `PathBuilder` module computes a characterizer, which is then used to construct the weakest precondition in `WeakestPrecondition`. Finally, the `ParallelRunner` checks each entailment by using one (or multiple) SMT backends. In addition, the engine can export the generated SMT-LIB representation for debugging purposes. This feature is activated by specifying an output path via the `--output` command-line option.

4.1.2 SMT Solver Integration

In Section 2.4 we presented the general encoding strategy and explain how verification conditions can be reduced to SMT satisfiability checks. We now turn to the implementation perspective and describe how these entailment queries are actually constructed, dispatched, and evaluated in the tool. The following discussion is specific to our implementation and addresses aspects that were intentionally abstracted away in the conceptual part of the thesis.

At this point in the pipeline, all high-level reasoning has already been compiled down to a sequence of entailments of the form $P \models WP$. The remaining task is to translate these entailments into solver queries and to select an appropriate SMT backend.

`ParallelRunner` is the implementation's central dispatch point for resolving entailment checks. The SMT backend used to discharge entailments can be optionally selected by the user via the command-line option `--smtmode`, choosing one of `z3`, `cvc5`, `cvc5-proc`, or `both`. At runtime, this choice overrides the default to determine how each entailment check is resolved.

Supported SMT backends.

We use Microsoft's *Z3 Theorem Prover* [12] as our primary SMT backend. Z3 is a mature, highly optimized SMT solver and provides strong, well-engineered support for the theories that arise in our verification conditions – most importantly, integers, arrays, and uninterpreted functions. In this mode, all entailments are discharged directly via the in-process `Z3Backend`, which utilizes the Z3 Java API[1].

As an alternative backend solver, the implementation also supports *cvc5* [13]. Adding a second SMT solver is motivated by the benefits of solver diversity. Since SMT solving is generally incomplete and validity is undecidable for many relevant theories, different solvers can behave quite differently on the same verification conditions. In `cvc5` mode, entailments are discharged via the in-process `CVC5Backend` using the cvc5 Java API[2]. However, due to practical limitations[3] of the native-library integration, we consider this mode discontinued in practice and prefer the external mode instead (see below).

The `cvc5-proc` mode is the practical alternative to the in-process version. It reuses the existing SMT encoding provided by the Z3 Java API and invokes cvc5 as an external process via `CVC5ProcBackend`. This avoids the fragility of the in-process integration while still allowing us to benefit from cvc5's different solving behavior. In practice, `cvc5-proc` should be preferred over the legacy `cvc5` mode.

Parallel race mode. A noteworthy feature is the parallel race mode, which can be enabled via the `both` option and is also the default configuration. In this mode, each entailment is dispatched to two solvers in parallel, and the first decisive[4] result is taken. By default, `Z3Backend` and `CVC5ProcBackend` are used for the race. We only fall back to the second solver within the remaining time budget in case the first solver returns **unknown**. This provides a pragmatic robustness/performance trade-off: many obligations are discharged quickly by at least one solver, while the second backend acts as a safety net on instances where the first solver struggles.

[12]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

1: `https://z3prover.github.io/api/html/namespacecom_1_1microsoft_1_1z3.html`

[13]: Barbosa et al. (2022), 'cvc5: A Versatile and Industrial-Strength SMT Solver'

2: `https://cvc5.github.io/docs/cvc5-1.1.2/api/java/java.html`

3: In-process `cvc5` support relies on translating our AST into cvc5's internal term representation via the Java API. In our initial experiments, this encoding step was disproportionately slow in performance and cumbersome to implement.

4: *Decisive* here means that the solver reports either **sat** (counterexample exists) or **unsat** (obligation proved). In contrast, **unknown** is inconclusive and shows that the solver cannot solve the query: while we treat it conservatively as **sat** if nothing better is available, the other solver may still return a decisive **unsat** and actually discharge the obligation.

4.2 Results and Evaluation

In this section, we evaluate the syntactic backend empirically and compare it against Hypra's existing semantic approach. Our goal is twofold. First, we assess how the new mode performs as a drop-in alternative. We compare which benchmarks can be verified by each backend and what the corresponding runtimes are. Second, we want to gain insight into our pipeline's internal cost and measure the time spent in the individual phases in order to identify the dominant bottlenecks.

Our experiments are based on a benchmark set from Hypra's original evaluation [5], which allows us to directly contrast the semantic and syntactic backends on an established collection.

[5]: Dardinier et al. (2024), 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'

4.2.1 Experimental Setup

All experiments are conducted on an *Apple MacBook Pro* (2020) equipped with a *1.4 GHz Quad-Core Intel Core i5* CPU and 8 GB of memory. We run the JVM with a thread stack size of 32 MB (`-Xss32M`). To reduce noise due to runtime variability, we execute each benchmark suite five times[5] and report aggregate results over these runs.

For the main comparison between Hypra's existing semantic backend[6] and our syntactic backend, we mainly consider the default solver configuration with parallel race mode enabled, running Z3 together with cvc5 in external-process mode (`cvc5-proc`), as it is our most stable mode. The timeout for all SMT solver modes is set to 10 seconds. Unless stated otherwise, automatic framing is enabled in this comparison, as it is part of the intended end-to-end pipeline.

For the detailed analysis of the syntactic pipeline, we first break down the runtime across its individual stages for the `z3` and `cvc5-proc` solver modes. Additionally, we then examine the impact and overhead of automatic framing by running the same pipeline both with framing enabled and disabled, which allows us to isolate its effect on solving success as well as its runtime overhead.

5: Each execution runs the entire suite in a fresh JVM process to produce a more realistic, steady-state performance profile. This is particularly important for the semantic configuration, whose runtimes showed to be highly sensitive to JVM warmup and showed high variance across runs when the same benchmark was repeated within a single long-lived JVM.

6: For the semantic backend, we use the Hypra implementation as described in [7].

4.2.2 Overall Comparison: Semantic vs. Syntactic

We begin with an end-to-end comparison between Hypra's existing semantic backend and our new syntactic approach. Since we feature several solver configurations, we focus on its most robust setup, the parallel race mode (`--smtmode both`), which runs Z3 and cvc5 (external-process mode) in parallel. This is the configuration we expect users to run in practice.

For comparability with prior work, we use the benchmark suite from Hypra's original evaluation. However, we remove test cases that rely on collections or datatypes other than integers, as these features are currently outside the scope of our implementation. The resulting test suite consists of 80 tests.

The results show that the syntactic backend achieves a substantial performance improvement while maintaining a high success rate. The semantic backend fails on one benchmark[7] and the syntactic backend in race mode produces the expected result on 76 out of the 80 benchmarks. The concrete failing test cases for the syntactic implementation are:

7: The `exists/orhle/sandbox.hhl` benchmark is part of the original Hypra benchmark suite, which the Hypra paper reports as fully verifiable. Nevertheless, when running our experiments with the updated implementation we use throughout (from [7]), this benchmark fails.

Table 4.1: Overall results on the Hypra benchmark suite (five repetitions, averaged per test). For each mode, we report the number of benchmarks (#tests), the count of failed tests, and pass rate over the full suite. The runtime statistics are computed only on the restricted subset solvable by both the semantic implementation and the syntactic backend in race mode: total runtime (s), mean, standard deviation σ, and median per-test runtime (s), and the 95th-percentile per-test runtime (P95, s), i.e., 95% of tests complete within this time.

Mode	#tests	Failed	PassRate	Total (s)	Mean (s)	σ (s)	Median (s)	P95 (s)
semantic	80	1	98.75%	206.58	2.75	7.80	0.86	9.07
syntactic-both	80	4	95.00%	55.53	0.74	3.13	0.07	3.58
syntactic-cvc5-proc	80	6	92.50%	93.61	1.25	4.85	0.09	4.85
syntactic-z3	80	9	88.75%	36.69	0.49	2.10	0.04	1.67

- `exists-forall/orhle/gni/nondet_leak.hhl`
- `forall-exists/hypa/counter_diff.hhl`
- `forall-exists/hypa/counter_sum.hhl`
- `forall/pcsat/half_square_ni.hhl`

All four are non-deterministic test cases. Their failure is explained by a deliberate restriction in our current implementation: as discussed in Remark 2.2.1, we eliminated the use of *hints* for nondeterministic assignments. The SMT solvers struggle with some proof goals to instantiate the nondeterministic variables if no hints are given. Importantly, these results show that our hint-free mechanism is already quite strong – successfully handling the remaining 19 nondeterministic benchmarks in the suite. Yet, they also show that for some proof goals, additional guidance in the form of hints or equivalent mechanisms remains necessary.

From this point on, we only compare runtimes on the largest common subset of benchmarks that both the semantic version and the race mode configuration can successfully verify. Consequently, we restrict the runtime evaluation to 75 files[8]. On this restricted subset, the semantic version has an aggregated runtime of 206.58 s (mean: 2.75 s/test, median: 0.86 s/test), whereas the syntactic implementation in race mode completes in 55.53 s (mean: 0.74 s/test, median: 0.07 s/test), making it almost four times as fast as the semantic version. Table 4.1 highlights the results for all solver modes.

8: We have seen that the semantic implementation cannot verify one test, and the syntactic race mode fails on four different tests.

In terms of raw runtime, `z3` is the fastest configuration, which is consistent with the fact that both encoding and solving happen within the same in-process Z3 API, avoiding any process-level overhead. By comparison, the `cvc5-proc` mode first creates the encoding via the Z3 API, and then spawns cvc5 as a system call, which adds noticeable process overhead.

This also explains why the runtime of the parallel race mode tends to lie between the `z3` and `cvc5-proc` runtimes in general. The race mode inherits the overheads of running both backends (thread creation, coordination, and two solver invocations). Yet, it benefits whenever the faster backend already returns a definitive **sat**/**unsat** answer.

At the same time, the results in Table 4.1 show the benefit of solver diversity: race mode improves robustness and yields the highest pass rate among the syntactic configurations. The remaining four failures listed above are precisely the benchmarks that neither Z3 nor cvc5 can resolve in our setting, so running both in parallel cannot recover them.

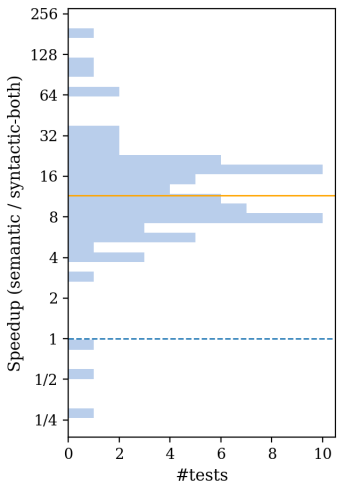


Figure 4.2: Logarithmic distribution of per-benchmark speedups of the syntactic backend over the current semantic implementation. The dashed line at 1 marks equal runtime, the orange line marks the median of the speedup.

To quantify the per-test performance difference, we compute a speedup value for each benchmark individually. Concretely, for each test case t we first average its runtime over the five repetitions for each backend, obtaining $\overline{T}_{\text{sem}}(t)$ and $\overline{T}_{\text{syn}}(t)$, and then define the per-test speedup as

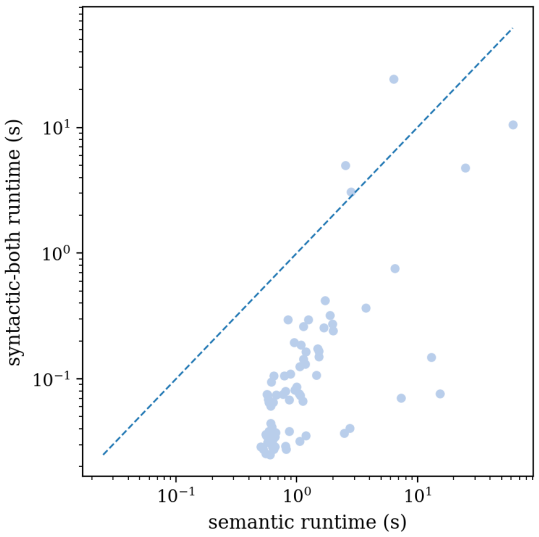


Figure 4.3: Semantic vs. syntactic runtime per benchmark (log-log scale). Each point represents one test case and its average runtime over all repetitions. The diagonal $y = x$ indicates equal runtime. Points below the diagonal correspond to benchmarks where the syntactic pipeline is faster, while points above indicate cases where the semantic implementation is faster.

Table 4.2: Test cases from the benchmark where verification with our syntactic pipeline in race mode is slower than with the existing semantic implementation. Runtimes averaged over repetitions.

Test case	Type	Paths	Semantic (s)	Syntactic (s)	Speedup
`forall/descartes/defaultcookie.hhl`	∀∀∀	62	6.31	24.38	0.26×
`forall/descartes/datapoint_false.hhl`	∀∀∀	30	2.53	4.98	0.51×
`forall/descartes/solution_false.hhl`	∀∀∀	18	2.83	3.06	0.92×

the ratio

$$\text{speedup}(t) = \frac{\overline{T}_{\text{sem}}(t)}{\overline{T}_{\text{syn}}(t)} .$$

We then summarize these per-test ratios using aggregate statistics, and visualize their distribution with a histogram in Figure 4.2. On this benchmark suite, the syntactic implementation achieves a geometric mean[9] speedup of about 11×, and the 95th-percentile speedup is 67×. This indicates that, in the vast majority of cases, the syntactic pipeline discharges verification conditions significantly faster than the semantic backend.

9: We use the geometric mean for speedups because a speedup is a multiplicative ratio between runtimes.

Figure 4.3 complements these ratio-based summaries by plotting the absolute runtimes of both implementations against each other. Consistent with the aggregate speedups, 72 out of 75 benchmarks (96%) are verified faster with our syntactic version, showing that we improve runtime on nearly the entire benchmark suite, with only a small number of outliers where the semantic pipeline performs better.

Concretely, the three benchmarks that experience a slowdown when verified with our implementation are listed in Table 4.2. They all come from the `descartes` suite, and verify successfully. A look at their source code gives us a natural explanation for their long runtimes. All three programs exhibit an unusually large number of control-flow paths together with deep hyper-quantification. For example, the `compare2` method in the `defaultcookie` benchmark has 62 paths, and its postcondition contains three nested state quantifiers. This combination is particularly challenging for the syntactic pipeline, because our weakest-precondition construction expands each quantified state by branching over all characterizer paths and then recursing into the instantiated body. As already discussed in Remark 2.1.1, this leads to an exponential blow-up. Hence, these bench-

Table 4.3: Test cases from the benchmark where verification with our syntactic pipeline in race mode achieves the largest speedups over the existing semantic implementation. Runtimes averaged over repetitions.

Test case	Type	Paths	Sem. (s)	Syn. (s)	Speedup
`exists/descartes/match_false_exists.hhl`	∃∃∃	4	15.36	0.08	200.39×
`exists-forall/orhle/param-usage/three_used.hhl`	∃∃∃∀∀∀	1	7.34	0.07	104.38×
`forall-exists/orhle/param-usage/three_used_false.hhl`	∀∀∀∃∃∃	1	13.08	0.15	88.21×
`exists/descartes/node_false_exists.hhl`	∃∃∃	4	2.73	0.04	67.13×
`exists/descartes/fileitem_false_exists.hhl`	∃∃∃	3	2.46	0.04	66.73×

marks quickly have large verification conditions and correspondingly longer runtimes, even though they remain provable.

We also want to have a look at the opposite case, namely those benchmarks where our syntactic pipeline yields the largest performance improvements. Concretely, Table 4.3 lists the five test cases with the highest speedups, again averaged over repetitions. In contrast to Table 4.2, these programs are characterized by only a few control-flow paths[10], so our weakest-precondition construction stays compact and avoids the path-induced blow-up.

10: Notably, they still reason about complex hyperproperties with up to six nested state quantifiers.

4.2.3 Evaluation of Syntactic Backend Stages

The overall comparison has shown that the syntactic backend is substantially faster than the existing semantic implementation and requires fewer hints. We now turn to a more fine-grained evaluation of the individual stages of the syntactic split-free pipeline itself. We instrument each major component of the core-pipeline (characterizer, WP construction, SMT encoding, SMT solving) and measure its contribution to the total runtime. This provides a more detailed picture of the implementation's behavior and highlights concrete bottlenecks and optimization opportunities.

Remark 4.2.1 We slightly adapt the benchmark suite for this stage-wise evaluation once more. We additionally exclude the benchmarks on which at least one solver hits the global timeout. For these tests, the majority of the runtime is spent waiting for the solver, so including them would heavily bias the per-stage breakdown without providing additional insight into the internal costs of the syntactic pipeline. The additionally excluded tests are:

- `forall/descartes/solution_true.hhl`
- `exists/descartes/datapoint_false_exists.hhl`

For the stage-wise breakdown, we compute aggregated time shares per mode (`z3` and `cvc5-proc`) as follows. For each benchmark, we first measure the average time spent in each of the four main stages per split-free triple and average these per-stage times again over the five repetitions. For a given mode, we then sum these averaged times over all benchmarks, obtaining four global totals (one per stage). Finally, we normalize by dividing each stage's total by the sum of all four totals, so that the resulting values represent the fraction of overall runtime attributable to each stage for that mode.

The results in Table 4.4 and Figure 4.4 show that, for both solvers, the vast majority of time is spent in the SMT-related parts of the pipeline, while Characterizer and WP construction account only for a small fraction of the overall runtime. For Z3, the largest share is spent in SMT encoding

Stage	cvc5 (%)	z3 (%)
Char.	0.02	0.05
WP	4.98	8.17
Encoding	60.39	82.23
Solving	34.61	9.52

Table 4.4: Relative runtime spent in each backend stage, aggregated over all benchmarks, for the `cvc5-proc` and `z3` modes.

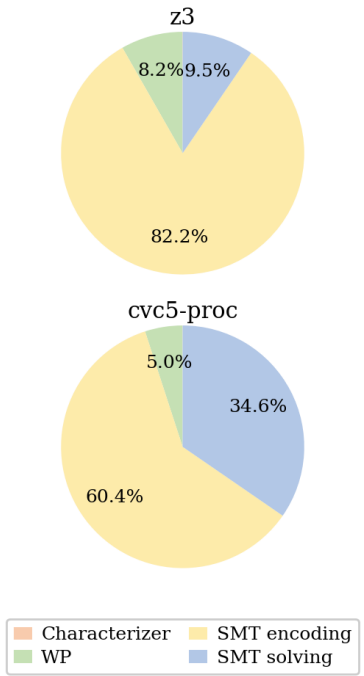


Figure 4.4: Relative runtime spent in each stage of the syntactic backend for the `z3` (top) and `cvc5-proc` (bottom) modes. The percentages correspond to the results reported in Table 4.4.

(about 82%), while actual solving contributes only about 10%. This is consistent with running both encoding and solving in the same in-process library, where a substantial amount of work happens on building the SMT formula, and the solver is run on an already optimized encoding.

In contrast, the `cvc5-proc` mode necessarily introduces an additional boundary between encoding and solving: we still construct the SMT encoding via the in-process Z3 API, but then we serialize it to SMT-LIB and invoke cvc5 as a separate process. This external handoff results in additional overhead from process creation, and also shifts substantial work into the measured solving phase. cvc5 must first run its own internal preprocessing before it can actually search for a model or proof. As a result, solving takes a much larger share in `cvc5-proc` (about 35%), while encoding takes a smaller share (about 60%) compared to Z3.

That we spend such a small share of time in the characterizer is also unsurprising. The characterizer operates purely on the program text: it explores the control-flow structure and computes path conditions and substitutions, but it doesn't care about the hyperproperties in the pre- and postconditions. Since these are typically the most complex part of the benchmarks, the expensive reasoning work arises only in later stages.

Example

To make this decomposition more concrete, we now break down the runtime on a representative example: the `compare2` method from the file `defaultcookie.hhl`, run in `z3` mode. This file is the slowest benchmark for our syntactic backend, and it is particularly illustrative because it combines an unusually large number of control-flow paths (62) with a deeply quantified hyper-postcondition of the form ∀∀∀. That is exactly the kind of structure that stresses our syntactic backend.

Stage	Time
Characterizer	3.372 ms
Weakest Pre.	1.201 s
SMT Encoding	19.037 s
SMT Solving	22.133 ms

Table 4.5: Runtime spent in each backend stage for the `compare2` method from `defaultcookie.hhl` using the `z3` mode, when run once.

Table 4.5 shows the results. Even for such a high amount of paths, the characterizer cost is negligible (22 ms), and WP construction accounts for only 1.2 seconds. The runtime is instead almost entirely spent in SMT encoding (19 seconds), while Z3's actual solving time is negligible (22 ms). Although it is an extreme example, it clearly highlights once more where the bottleneck of the syntactic pipeline lies.

Overall, the breakdown clearly shows that we still have the most potential for improvement at the SMT boundary, as this is where most of the runtime is currently invested. Improvements in encoding efficiency, formula shape, and solver interaction will directly attack the dominant cost center.

4.2.4 Impact of Automatic Framing

For the final part of the evaluation, we investigate whether our automatic framing heuristics from Section 3.6 actually help in practice. To this end, we rerun the benchmark suite with the syntactic backend in race mode, now with framing disabled, to compare the resulting verification outcomes.

In the default configuration with framing enabled, the syntactic backend verifies 76/80 benchmarks, as we have seen in the previous subsection. Now, with framing disabled, this number drops to 72/80. The concrete test cases for which framing has an effect are:

Table 4.6: Overall results for the syntactic backend in parallel race mode with and without automatic framing (five repetitions, averaged per test). We report the number of benchmarks (#tests), the count of failed tests, and the pass rate. Runtime statistics are computed on the restricted subset solvable by the syntactic backend in race mode: total runtime (s), mean, standard deviation σ, median per-test runtime (s), and the 95th-percentile per-test runtime (P95, s).

Config	#tests	Failed	PassRate	Total (s)	Mean (s)	σ (s)	Median (s)	P95 (s)
with framing	80	4	95.00%	69.75	0.92	3.92	0.09	4.34
without framing	80	8	90.00%	66.03	0.87	3.68	0.09	3.92

- `forall/pcsat/double_square_ni.hhl`
- `forall/pcsat/squares_sum.hhl`
- `exists/orhle/blackjack_until21.hhl`
- `forall-exists/orhle/delimited-release/parity_fun.hhl`

That shows that our automatic framing implementation is directly responsible for turning four benchmarks from failure/unknown into success. This is particularly noteworthy given that the suite contains only eight tests in total, where at least one method is actually split into multiple triples. For all other cases, framing doesn't apply, as they only contain a single triple.

For the performance evaluation, we again exclude the four benchmarks that the syntactic race mode cannot solve, so that any possible solver timeouts do not dominate the comparison, leaving 76 tests. We run the benchmark again, five times in each configuration. The results are shown in Table 4.6. Overall, the runtime impact of framing is marginal, showing at most a slight performance degradation.

In summary, automatic framing turns out to be a small and lightweight but highly effective addition to the syntactic backend. Conceptually, it addresses exactly the gap introduced by splitting: important contextual facts that are obvious in the original program are not automatically visible in every generated triple.

Conclusion 5

To conclude the thesis, we briefly summarize the contributions we have made and outline directions for future work that could even further improve automation, performance, and expressiveness beyond what has been achieved in this thesis.

5.1 Summary of Contributions

In this thesis, we have developed, implemented, and evaluated a syntactic verification backend for Hyper Hoare Logic as an alternative to Hypra's existing semantic pipeline.

The first contribution is a weakest-precondition construction for **split-free programs**[1] tailored to HHL. This construction systematically handles deterministic and nondeterministic assignments, conditionals, and assertions, which require special treatment because they can lead to error-states. It is grounded in a path-based characterizer that makes all control-flow paths of the program explicit, such that the WP construction can systematically account for every possible execution.

1: As defined in Definition 2.0.1.

Building on this core, the second contribution extends the approach to **method call and loops**. We introduce splitting schemas that decompose language constructs that contain hyper-assertions themselves into multiple split-free verification triples, such that they can be handled by our core approach. We support error propagation across triples, and complement the resulting triples with automatic framing heuristics that recover crucial contextual information lost by splitting.

The third contribution is the **integration** of this developed calculus into Hypra as a fully functioning syntactic backend. The implementation reuses Hypra's existing front-end and AST, and implements the calculus with different SMT configurations that the user can choose from. Most notably, it supports a parallel race mode, which combines the strengths of multiple SMT solvers.

Finally, the fourth contribution is an extensive **empirical evaluation** and comparison with the existing semantic backend. It demonstrates that we have developed not just a theoretically elegant alternative, but also a practically convincing one. Across the slightly adapted benchmark suite, our syntactic backend consistently shows substantial performance gains over the existing implementation, often discharging verification conditions in a fraction of the time.

5.2 Future Work

While the syntactic backend developed in this thesis already provides a practical and efficient alternative to the existing semantic pipeline, several opportunities for further extension and refinement remain.

5.2.1 Formalization in a Proof Assistant

A natural next step is a complete formalization of the calculus in a proof assistant such as Isabelle/HOL [14]. In this thesis, the soundness arguments for the introduced concepts were given intuitively and on paper. Formalizing these arguments and proofs would increase confidence in the correctness of the approach and clarify subtle corner cases.

[14]: Nipkow et al. (2002), *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*

5.2.2 User-Guided Hints for Nondeterminism

A second direction is to reintroduce hints for nondeterministic assignments in a controlled, optional way. In the semantic Hypra backend, every nondeterministic assignment has to be accompanied by a user-provided hint. In this thesis, we deliberately choose a different path and don't make hints a prerequisite for `havoc` updates. This keeps the basic workflow simpler and avoids pushing additional annotation burden onto the user. However, it also explains the small set of non-deterministic benchmarks that our implementation currently cannot verify.

A natural extension would therefore be to reintroduce hints in a user-guided, optional form: by default, nondeterministic assignments would be handled without hints, preserving full automation for most programs, while users could selectively add hints where they see that the SMT solver struggles.

5.2.3 Parallelization of Split-free Triples

The syntactic backend already exposes a list of split-free triples as its central intermediate representation before computing the weakest precondition. By design, these obligations are independent and can in principle be discharged in parallel. An obvious idea is therefore to exploit this structure and dispatch split-free triples concurrently to the backend, potentially using a thread pool and batching of SMT queries.

How much this helps in practice remains an open question: in the benchmark suite used here, most tests (73 out of 81) consist of methods that yield only a single triple, so there is limited parallelism to exploit. Larger case studies or code bases with many independent methods and more splitting constructs are likely to benefit more and would be an interesting target for a future evaluation.

5.2.4 Less Conservative Framing

The automatic framing heuristics introduced in this thesis are conservative: they only propagate preconditions that do not mention modified variables and avoid assumptions with existential state quantification[2].

2: Intuitively, existential state quantifiers are delicate because they talk about the existence of *some* reachable state. For instance, after non-terminating code or calls with unknown behavior, that particular state may no longer be reachable or exist at all. So, blindly treating such assertions as invariants can easily become unsound.

However, there is room for a more fine-grained criterion. A concrete next step would be to propagate existential state assertions in all cases except when we encounter one of the following constructs on the path:

- an `assume` statement, which may arbitrarily filter the set of reachable states,
- a method call, whose termination and behaviour may depend on additional specifications, and

- a `while` loop without (nested) `decreases` clauses, where non-termination may prevent the existence of post-states altogether.

Also, rewriting the program into SSA form would reduce the impact of the "mentions modified variables" restriction, since assignments only introduce fresh variables. Hence, the variables referenced by an assertion at some point are never modified again afterwards.

This refinement would allow the framing phase to retain more informative invariants across splits. Formalizing and validating this relaxed propagation rule, both theoretically and experimentally, is a promising direction for future work.

5.2.5 Support for More Types and Collections

The current implementation is restricted to integer-typed programs. The semantic Hypra implementation has already been extended in subsequent work [7] to support more datatypes and collections (such as sets, maps, and sequences). A natural next step would be to lift these extensions to the syntactic setting as well. This would significantly broaden the applicability of the syntactic backend and allow a one-to-one comparison with the full Hypra benchmark suite, not just its integer fragment.

[7]: Winkler (2025), *Improving a Deductive Program Verifier for Hyperproperties*

5.2.6 Generalizing Nested Splits Beyond $\forall^*$-Properties

In its current form, the nested composition scheme from Section 3.5 is restricted to $\forall^*$-properties, i.e., properties that can only quantify *universally* over states. Right now, this is the only part of the pipeline that isn't fully generalized. This restriction is crucial for our existing postcondition construction, based on the pigeonhole principle.

A natural direction for future work is to lift this restriction and support arbitrary hyperproperties in nested splits. To do so, one would need a general construction of a combined postcondition for conditionals that remains sound for arbitrary hyperproperties and that correctly accounts for all possible interleavings of branches. Designing such a construction, proving its soundness, and integrating it into the syntactic pipeline is a challenging but promising step towards a fully general treatment of nested splitting.

A

Appendix

A.1 Formal Grammar of the Input Language

This appendix section collects the formal concrete syntax of the language fragment supported by the current syntactic verifier implementation. The specification is derived directly from the parser implementation. While the main chapters introduce individual constructs incrementally and provide informal "snapshots" of the extended syntax as they become relevant, the formal grammar given here serves as the reference for the complete supported input language. Based on the parser, all binary operator precedence is right recursive. Note that the grammar specifies only concrete syntax. Any further well-formedness constraints are enforced separately by typing and semantic checks.

Lexical Conventions (Simplified)

```
MethodName  ::= (Letter | "_") { Letter | Digit | "_" }
ProgVar     ::= Letter { Letter | Digit | "_" }
AssertVar   ::= "_" Letter { Letter | Digit | "_" }
Type        ::= "Int"
Number      ::= Digit { Digit }
Letter      ::= "A" | ... | "Z" | "a" | ... | "z"
Digit       ::= "0" | ... | "9"
```

Program structure

```
HHLProgram    ::= { Method }

Method        ::= "method" MethodName "(" [ MethodVarDecl { "," MethodVarDecl } ] ")"
                  [ "returns" "(" [ MethodVarDecl { "," MethodVarDecl } ] ")" ]
                  { Precondition }
                  { Postcondition }
                  "{" Stmts "}"

MethodVarDecl ::= ProgVar ":" Type
Precondition  ::= "requires" Expr
Postcondition ::= "ensures"  Expr
```

Statements

```
Stmts          ::= { Stmt }

Stmt           ::= VarDecl | MultiAssign | MethodCallStmt | Assign
                 | IfElse | While | Assume | Assert | Havoc

VarDecl        ::= "var" ProgVar ":" Type

MethodCallStmt ::= MethodCall
MethodCall     ::= MethodName "(" [ ProgVar { "," ProgVar } ] ")"
```

```
MultiAssign    ::= ProgVar { "," ProgVar } ":=" MethodCall
Assign         ::= ProgVar ":=" PlainExpr

Havoc          ::= "havoc" ProgVar
Assume         ::= "assume" ( NormalAssertion | PlainExpr )
Assert         ::= "assert" ( NormalAssertion | PlainExpr )

IfElse         ::= "if" "(" PlainExpr ")" "{" Stmts "}"
                   [ "else" "{" Stmts "}" ]

While          ::= "while" [ WhileRule ] "(" PlainExpr ")"
                   { LoopInv }
                   [ "decreases" AddExpr ]
                   "{" Stmts "}"

WhileRule      ::= "syncRule" | "forAllExistsRule" | "existsRule" | "syncTotRule"

LoopInv        ::= "invariant" Expr
```

Expressions

```
Expr               ::= Assertion | PlainExpr

Assertion          ::= HyperAssertionErr | HyperAssertion | NormalAssertion

NormalAssertion    ::= Quantifier NormalAssertVarDecl { "," NormalAssertVarDecl } "::" Expr
NormalAssertVarDecl ::= AssertVar ":" Type

HyperAssertion     ::= Quantifier "<" AssertVar ">" { "," "<" AssertVar ">" } "::" Expr
HyperAssertionErr  ::= Quantifier "<<" AssertVar ">>" { "," "<<" AssertVar ">>" } "::" Expr

Quantifier         ::= "forall" | "exists"

PlainExpr   ::= BoolExpr [ "==>" Expr ]
BoolExpr    ::= EqExpr [ ("&&" | "||") BoolExpr ]
EqExpr      ::= RelExpr [ ("==" | "!=") EqExpr ]
RelExpr     ::= AddExpr [ (">=" | "<=" | ">" | "<") RelExpr ]
AddExpr     ::= MulExpr [ ("+" | "-") AddExpr ]
MulExpr     ::= UnaryExpr [ ("*" | "/" | "%") MulExpr ]
UnaryExpr   ::= "!" UnaryExpr | "-" Number | Atom

StateLookup ::= AssertVar "[" PlainExpr "]"

Atom        ::= ProgVar | AssertVar | StateLookup | Number | "(" Expr ")"
```

A.2 Full Definition of E_Ψ

This appendix section collects the complete definition of the recursive path-expansion transformation $E_\Psi[\cdot]$ in one place. For brevity, we use a metavariable $\preceq$ to range over all binary operators that are treated by E_Ψ (e.g., $=$, $\neq$, $<$, $\leq$, $>$, $\geq$). The four clauses for state quantifiers constitute the only non-trivial recursion rules, whereas all other Boolean connectives and first-order quantifiers are propagated homomorphically.

$$
\begin{aligned}
\mathsf{E}_\Psi\big[\forall\langle\sigma\rangle.\, Q\big] &\triangleq \forall\langle\sigma\rangle.\, \bigwedge_{i=1}^{n}\Big(b_i(\sigma) \Rightarrow \mathsf{E}_\Psi\big[\, Q[\theta_i]_\sigma \,\big]\Big),\\
\mathsf{E}_\Psi\big[\exists\langle\sigma\rangle.\, Q\big] &\triangleq \exists\langle\sigma\rangle.\, \bigvee_{i=1}^{n}\Big(b_i(\sigma) \wedge \mathsf{E}_\Psi\big[\, Q[\theta_i]_\sigma \,\big]\Big),\\
\mathsf{E}_\Psi\big[\forall\langle\sigma\rangle_{er}.\, Q\big] &\triangleq \forall\langle\sigma\rangle.\, \bigwedge_{\alpha\in A}\ \bigwedge_{(b,\theta)\in\Psi_\alpha}\Big(b(\sigma) \wedge \neg \mathit{cond}(\alpha)[\theta](\sigma) \Rightarrow \mathsf{E}_\Psi\big[\, Q[\theta]_\sigma \,\big]\Big),\\
\mathsf{E}_\Psi\big[\exists\langle\sigma\rangle_{er}.\, Q\big] &\triangleq \exists\langle\sigma\rangle.\, \bigvee_{\alpha\in A}\ \bigvee_{(b,\theta)\in\Psi_\alpha}\Big(b(\sigma) \wedge \neg \mathit{cond}(\alpha)[\theta](\sigma) \wedge \mathsf{E}_\Psi\big[\, Q[\theta]_\sigma \,\big]\Big),\\
\mathsf{E}_\Psi\big[\forall x.\, Q\big] &\triangleq \forall x.\, \mathsf{E}_\Psi[Q],\\
\mathsf{E}_\Psi\big[\exists x.\, Q\big] &\triangleq \exists x.\, \mathsf{E}_\Psi[Q],\\
\mathsf{E}_\Psi[\neg Q] &\triangleq \neg\, \mathsf{E}_\Psi[Q],\\
\mathsf{E}_\Psi[Q_1 \wedge Q_2] &\triangleq \mathsf{E}_\Psi[Q_1] \wedge \mathsf{E}_\Psi[Q_2],\\
\mathsf{E}_\Psi[Q_1 \vee Q_2] &\triangleq \mathsf{E}_\Psi[Q_1] \vee \mathsf{E}_\Psi[Q_2],\\
\mathsf{E}_\Psi[Q_1 \Rightarrow Q_2] &\triangleq \mathsf{E}_\Psi[Q_1] \Rightarrow \mathsf{E}_\Psi[Q_2],\\
\mathsf{E}_\Psi[b] &\triangleq b,\\
\mathsf{E}_\Psi[e_1 \preceq e_2] &\triangleq e_1 \preceq e_2,
\end{aligned}
$$

where σ is a state variable and θ a substitution that maps program variables to expressions, as given by the characterizer. We use the following syntactic instantiation operators:

1. **State-interpretation of expressions.** For an expression e over program variables, let $e(\sigma)$ denote the expression obtained by replacing every program variable x in e by a state lookup $\sigma(x)$.
2. **Path instantiation of state lookups.** For a hyper-assertion Q that may contain state lookups $\sigma(x)$, define $Q[\theta]_\sigma$ by structural recursion, with the only non-homomorphic clause:

   $$(\sigma(x))[\theta]_\sigma \triangleq \theta(x)(\sigma).$$

   Intuitively, $Q[\theta]_\sigma$ rewrites each lookup $\sigma(x)$ to the value of x after executing the prefix described by θ, interpreted in the *initial* state σ.
3. **Assertion condition instantiation.** For an assertion α with assertion condition $\mathit{cond}(\alpha)$, define

   $$\mathit{cond}(\alpha)[\theta](\sigma)$$

   as the formula obtained by first instantiating program variables in $\mathit{cond}(\alpha)$ according to the substitution θ, and then interpreting any remaining program variables as state lookups in σ (i.e., by replacing each program variable x with $\sigma(x)$).

A.3 Complete SMT-LIB Encoding of the Running Example Listing 2.1

This appendix section presents the complete SMT encoding of the verification conditions obtained from the weakest precondition constructed for Listing 2.1. The encoding is performed exactly as presented in Section 2.4.

We define the following precondition for the method `ifElse` (which was omitted previously):

$$\exists\langle\sigma_1\rangle.\, \forall\langle\sigma_2\rangle.\, \sigma_1(a) \geq \sigma_2(a) \wedge \sigma_1(a) \geq \sigma_2(b) \wedge \sigma_1(b) \geq \sigma_2(a) \wedge \sigma_1(b) \geq \sigma_2(b),$$

and recall that the weakest precondition as derived in Example 2.1.2 is:

$$\exists\langle\sigma\rangle.\ \big(\sigma(c) \wedge \big(\forall\langle\sigma'\rangle.\ \big(\sigma'(c) \Rightarrow \sigma(a) \geq \sigma'(a)\big) \wedge \big(\neg\sigma'(c) \Rightarrow \sigma(a) \geq \sigma'(b)\big)\big)\big) \\ \vee \big(\neg\sigma(c) \wedge \big(\forall\langle\sigma'\rangle.\ \big(\sigma'(c) \Rightarrow \sigma(b) \geq \sigma'(a)\big) \wedge \big(\neg\sigma'(c) \Rightarrow \sigma(b) \geq \sigma'(b)\big)\big)\big).$$

The full SMT-LIB encoding of the resulting verification condition, i.e. the negated implication from the precondition to the weakest precondition, generated by our syntactic implementation, looks as follows:

```
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun a () Int)
(declare-fun S ((Array Int Int)) Bool)
(assert (let ((a!1 (=> (exists ((_s1 (Array Int Int)))
                 (! (let ((a!1 (forall ((_s2 (Array Int Int)))
                                 (! (=> (S _s2)
                                        (and (>= (select _s1 a) (select _s2 a))
                                             (>= (select _s1 a) (select _s2 b))
                                             (>= (select _s1 b) (select _s2 a))
                                             (>= (select _s1 b) (select _s2 b))))
                                    :weight 0))))
                      (and (S _s1) a!1))
                    :weight 0))
               (exists ((_s1 (Array Int Int)))
                 (! (let ((a!1 (and (and (> (select _s1 c) 0) true)
                                    (forall ((_s2 (Array Int Int)))
                                      (! (let ((a!1 (=> (and (> (select _s2 c)
                                                                0)
                                                             true)
                                                        (>= (select _s1 a)
                                                            (select _s2 a))))
                                               (a!2 (and (not (> (select _s2 c)
                                                                 0))
                                                         true)))
                                           (let ((a!3 (and a!1
                                                           (=> a!2
                                                               (>= (select _s1 a)
                                                                   (select _s2 b))))))
                                             (=> (S _s2) a!3)))
                                         :weight 0))))
                          (a!2 (and (not (> (select _s1 c) 0)) true)))
                      (let ((a!3 (and a!2
                                      (forall ((_s2 (Array Int Int)))
                                        (! (let ((a!1 (=> (and (> (select _s2 c)
                                                                  0)
                                                               true)
                                                          (>= (select _s1 b)
                                                              (select _s2 a))))
                                                 (a!2 (and (not (> (select _s2 c)
                                                                   0))
                                                           true)))
                                             (let ((a!3 (and a!1
                                                             (=> a!2
                                                                 (>= (select _s1 b)
                                                                     (select _s2 b))))))
                                               (=> (S _s2) a!3)))
                                           :weight 0)))))
                        (and (S _s1) (or a!1 a!3))))
                    :weight 0)))))
  (not (and a!1))))
(check-sat)
(exit)
```

Bibliography

The references listed below are presented in the order in which they are cited in the text.

[1] C. A. R. Hoare. 'An axiomatic basis for computer programming'. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on page 1).

[2] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 'A Sound Type System for Secure Flow Analysis'. In: *Journal of Computer Security* 4.2–3 (Apr. 1996), pp. 167–187. DOI: 10.3233/jcs-1996-42-304 (cited on page 1).

[3] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *Journal of Computer Security* 18.6 (Sept. 2010). Ed. by Andrei Sabelfeld, pp. 1157–1210. DOI: 10.3233/jcs-2009-0393 (cited on page 1).

[4] Thibault Dardinier and Peter Müller. 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (June 2024), pp. 1485–1509. DOI: 10.1145/3656437 (cited on pages 1, 12, 16, 21).

[5] Thibault Dardinier, Anqi Li, and Peter Müller. 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (Oct. 2024), pp. 1279–1308. DOI: 10.1145/3689756 (cited on pages 1, 9, 10, 12, 21, 23, 46).

[6] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer Berlin Heidelberg, Jan. 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2 (cited on page 1).

[7] Paul Winkler. *Improving a Deductive Program Verifier for Hyperproperties*. Bachelor's Thesis. Advisors: Anqi Li, Thibault Dardinier, and Peter Müller. Apr. 2025 (cited on pages 2, 46, 55).

[8] Edsger W. Dijkstra. 'Guarded commands, nondeterminacy and formal derivation of programs'. In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975 (cited on pages 3, 6).

[9] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 'Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)'. In: *Journal of the ACM* 53.6 (Nov. 2006), pp. 937–977. DOI: 10.1145/1217856.1217859 (cited on page 15).

[10] Thibault Dardinier. 'Formal Foundations for Automated Deductive Verifiers'. ETH Diss. No. 31492. Doctoral Thesis. Zurich, Switzerland: ETH Zurich, Oct. 2025. DOI: 10.3929/ethz-c-000784984 (cited on pages 20–22, 24).

[11] Martin Odersky et al. *An Overview of the Scala Programming Language*. Technical Report LAMP-REPORT-2006-001. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne (EPFL), Mar. 2006 (cited on page 43).

[12] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on page 45).

[13] Haniel Barbosa et al. 'cvc5: A Versatile and Industrial-Strength SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cited on page 45).

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002 (cited on page 54).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☐ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

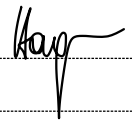
Automating Hyper Hoare Logic via Predicate Transformers

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Hagen	David Nikolaus

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 19 December 2025	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).