ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

An Automatic Program Verifier for Hyperproperties

Master's Thesis

Anqi Li

October 2023

Advisors: Thibault Dardinier, Prof. Dr. Peter Müller
Programming Methodology Group
Department of Computer Science, ETH Zürich

Abstract

Hyper Hoare Logic (HHL) is a powerful proof system that can prove and disprove arbitrary trace properties and hyperproperties (i.e. properties of multiple executions of the same program) for programs. It overcomes limitations of existing Hoare-style logics by reasoning over arbitrary sets of program states. For the time being, HHL remains as a novel theoretical achievement. Automation of HHL is required for it to be used to verify programs in practice.

This thesis aims at building a program verifier that automates HHL. In particular, we focus on automating the part of HHL that allows us to prove the absence and existence of certain (combinations of) executions of the same program. In this thesis, we propose an approach that automates HHL for a simple imperative programming language by developing encodings into Viper – a program verification tool-chain. Moreover, we have built a prototype which implements the proposed approach, and also evaluated it using the benchmarks of several state-of-the-art program verifiers. The evaluation shows that our prototype can prove and disprove various trace properties and hyperproperties within a reasonable time limit.

Acknowledgement

I would like to express my most sincere gratitude to my supervisor, Thibault Dardinier. I could not be more grateful to him for all the technical support that he has provided for me since the very beginning of my Master's thesis project. His enthusiasm towards this project and research in the field of computer science has always been extremely inspiring to me.

I would also like to thank Professor Peter Müller for giving me the opportunity to do my Master's thesis in his group. He and all the other members in his group have given me lots of valuable advice related to my thesis.

Lastly, I want to thank my parents, my partner Leo, and all the cats in my neighborhood for supporting me emotionally throughout this Master's thesis project.

Contents

Chapter 1

Introduction

In this introductory chapter, we present the motivation of building an automatic verifier for hyperproperties. In particular, we explain why we aim at developing such a verifier based on Hyper Hoare Logic [1]. After we make clear the motivation, we give an overview of this thesis to complete the introduction.

1.1 Motivation

Program verification is the process of formally proving that a program behaves according to its specification. One proof system widely used in program verification is Hoare Logic (HL) [2]. It can prove properties of individual program executions, namely trace properties, by using Hoare triples of the form $\{P\}C\{Q\}$, where C is a program command and P, Q are predicates describing the initial and final program states respectively. A Hoare triple $\{P\}C\{Q\}$ holds if and only if executing C in an initial state satisfying P can only result in final states satisfying Q.

However, classical HL cannot prove hyperproperties [3], i.e. properties of multiple program executions, such as determinism (if two executions of the same program agree on the input, then they also agree on the output). As a result, several extensions and adaptations of HL have been suggested to overcome this limitation. For instance, Sousa and Dillig [4] have proposed Cartesian Hoare Logic (CHL) that can reason over k executions of the same program, where k is some fixed integer ≥ 1. This is useful for verifying properties like associativity and monotonicity. Similar to classical HL, CHL can only prove the absence of bugs in computer programs because it overapproximates the possible program executions. To prove the existence of bugs in a program, O'Hearn [5] has proposed Incorrectness Logic (IL) which underapproximates the possible program executions [6]. Hoare-style logics combining overapproximation and underapproximation have also been pro-

posed, such as Outcome Logic (OL) [7] and RHLE [8]. Nevertheless, each of the existing logics has its own limitations regarding the program properties that they can prove (including trace properties and hyperproperties) and the type of reasoning that they can perform (including overapproximation and underapproximation). For example, RHLE can prove both the existence and absence of bugs while relating multiple executions of the same program, and it can also prove $\forall\exists$-hyperproperties (i.e. the existence of an execution that satisfies some condition with respect to all possible executions), but, like any other existing logics, it cannot prove $\exists\forall$-hyperproperties (i.e. the existence of executions with respect to all other executions) such as the existence of a minimum returned value. Consequently, verifying a program that involves various properties requires the application of multiple logics. This motivates the proposal of Hyper Hoare Logic (HHL) [1] – a logic that can prove and disprove all aforementioned program properties and more. It is also formally proved to be sound and complete.

For the time being, HHL is a purely theoretical achievement. Its theoretical details have been comprehensively presented in the paper authored by Dardinier and Müller [1]. This thesis project aims at building a verifier that can automatically reason with HHL to prove hyperproperties by developing an encoding into a program verification infrastructure called Viper [9].

1.2 Thesis Outline

The content of this thesis is organized as follows. In Chapter 2, we provide all background knowledge related to this thesis, especially the details of HHL. Afterwards, we describe the approach used to develop Viper encodings for a set of commonly used program commands, except for loops, in Chapter 3. This is followed by the presentation of verification primitives that allow users to provide verification guidance in Chapter 4. The approach to encoding loops is elaborated in Chapter 5. After all encodings are shown and explained, we elucidate how we implemented a prototype that can automatically generated these encodings in Chapter 6. The evaluation result of the prototype is given in Chapter 7. Finally, in Chapter 8, we draw a conclusion on our work and discuss its limitations and possible improvements that can be made in the future.

Chapter 2

Background

This chapter aims at equipping the readers with all background knowledge related to this thesis. To begin with, we give a detailed introduction to HHL, and then briefly describe the Viper verification tool-chain that we used to automate HHL.

As a side note, all content in Section 2.1 is adapted from the paper where HHL was originally introduced. For readers who are interested in learning about HHL more comprehensively, we strongly recommend reading the original paper written by Dardinier and Müller [1].

2.1 Hyper Hoare Logic

Hyper Hoare Logic, abbreviated as HHL, is a powerful proof system capable of proving and disproving arbitrary trace properties and hyperproperties. It is centered around the idea of reasoning about arbitrary sets of program states instead of a fixed number of program states. Analogous to Hoare triples, HHL uses hyper-triples of the form $[P]C[Q]$, where C is a program command and P, Q are predicates describing the initial and final sets of program states respectively. Before we formally define hyper-triples, we will introduce two concepts, i.e. hyper-assertions and a semantic function *sem*, that are dependent by the definition of hyper-triples.

Definition 2.1 ***Hyper-assertions.*** *A hyper-assertion is a predicate over sets of program states.*

Definition 2.2 ***Semantic function.*** *We define a semantic function named sem that takes a command C and a set of states S as inputs. It returns the set of states that can be reached by executing C in an initial state from S. Let σ_0 be a state in S and σ be a state in the set of states returned by sem. Moreover, $\langle C, \sigma_0 \rangle \rightarrow \sigma$ refers*

to the big-step semantics of the command C.

$$sem(C, S) \triangleq \{\sigma \mid \exists \sigma_0.\ \sigma_0 \in S \wedge \langle C, \sigma_0 \rangle \rightarrow \sigma\}$$

With hyper-assertions and the semantic function, we can now formally define hyper-triples.

Definition 2.3 ***Hyper-triples.*** *Let P and Q be two hyper-assertions, S be a set of initial states, and C be a program command. A hyper-triple $[P]C[Q]$ is valid if and only if executing C in any set of initial states satisfying P leads to a set of final states satisfying Q. That is to say:*

$$\models [P]C[Q] \triangleq (\forall S.\ P(S) \implies Q(sem(C, S)))$$

Similar to verifying programs with classical Hoare Logic, verifying some program C with respect to its precondition P and postcondition Q using HHL is equivalent to verifying the validity of the hyper-triple $[P]C[Q]$. Since P, Q are assertions that universally or existentially quantify over sets of states, HHL can prove or disprove any trace properties and hyperproperties.

In this thesis, we focus on automating only the part of HHL that can prove either the absence or the existence of certain program executions. To prove the absence of some executions, a program postcondition must universally quantify over the set of final program states, thus overapproximating the set of final states that are reachable from the set of initial states satisfying the precondition upon program termination. In contrast, to prove the existence of some executions, a program postcondition must existentially quantify over the set of final program states, thus underapproximating the set of final states that are reachable from the set of initial states satisfying the precondition upon program termination [5]. In the rest of this thesis, we refer to the part of HHL that performs the first kind of reasoning (i.e. overapproximation) as $\forall$-HHL and the part of HHL that performs the second kind of reasoning (i.e. underapproximation) as $\exists$-HHL. Their definitions are given below.

Definition 2.4 ***$\forall$-HHL.*** *$\forall$-HHL is the strict subset of HHL that involves only overapproximation reasoning over multiple executions of the same program.*

Definition 2.5 ***$\exists$-HHL.*** *$\exists$-HHL is the strict subset of HHL that involves only underapproximation reasoning over multiple executions of the same program.*

To help the readers better understand how $\forall$-HHL or $\exists$-HHL can be used to verify programs, we will elaborate on a concrete example in the following paragraphs.

Imagine that we would like to prove that Program 2.1 below satisfies strong non-interference (SNI). SNI is a hyperproperty defined as follows: for all

pairs of executions τ_1, τ_2, if they agree on the initial values of all low-sensitivity variables (i.e. *low_in*), they must also agree on the final values of all low-sensitivity variables (i.e. *low_out*). To verify SNI for the program, we must write a precondition and a postcondition based on the definition of SNI. Before doing so, let us introduce the following notations. Let S, S' be the set of initial program states and the set of final program states (i.e. the set of program states after C is executed in S) respectively. Moreover, let σ_1, σ_2 be two arbitrary states in S and σ_1', σ_2' be two arbitrary states in S'. Furthermore, to represent the value of some variable x in a state σ, let us use the notation $\sigma(x)$. By using these notations and also following the definition of SNI, we can easily derive that the precondition and postcondition should be $\forall \sigma_1, \sigma_2 \in S.\ \sigma_1(low\_in) = \sigma_2(low\_in)$ and $\forall \sigma_1', \sigma_2' \in S'.\ \sigma_1'(low\_out) = \sigma_2'(low\_out)$ respectively. To verify the program with respect to this specification, it is clear that we must reason with $\forall$-HHL.

```
1  method sni_good(low_in: Int, high_in: Int)
2      returns (low_out: Int)
3  {
4      if (low_in > 0) {
5          low_out := low_in + 1
6      } else {
7          low_out := low_in * (-1) + 2
8      }
9  }
```

Program 2.1: A program which satisfies SNI

In comparison, let us consider a program that violates SNI. Such a program can be easily obtained by updating line 4 of Program 2.1 to let the value of *low_out* depend on the value of *high_in*, as shown by Program 2.2. To prove the violation of SNI, we need to prove the existence of bad program executions. Therefore, the precondition and postcondition should be updated to $\exists \sigma_1, \sigma_2 \in S.\ \sigma_1(low\_in) = \sigma_2(low\_in) \wedge \sigma_1(high\_in) > 0 \wedge \sigma_2(high\_in) < 0$ and $\exists \sigma_1', \sigma_2' \in S'.\ \sigma_1'(low\_out) \neq \sigma_2'(low\_out)$ respectively. We can notice that the precondition poses constraints on the value of *high_in* in the two initial states. This is to guarantee that *any* set of initial states satisfying the precondition will lead to a set of final states satisfying the postcondition, which complies with the definition of hyper-triples. It is clear that $\exists$-HHL should be used to verify this program.

```
1  method sni_bad(low_in: Int, high_in: Int)
2      returns (low_out: Int)
3  {
4      if (high_in > 0) {
5          low_out := low_in + 1
6      } else {
7          low_out := low_in * (-1) + 2
8      }
```

```
9 }
```

Program 2.2: A program which violates SNI

2.2 Viper

Viper is a powerful program verification toolchain with an architecture on which new program verifiers can be easily built. As shown by Figure 2.1, it consists of an intermediate language, also called Viper, and two back-ends. The back-end that is based on symblic execution is named Silicon [10], while the back-end that relies on verification condition generation and generates Boogie [11] programs as output is called Carbon. Ultimately, both of them use the SMT solver Z3 [12].

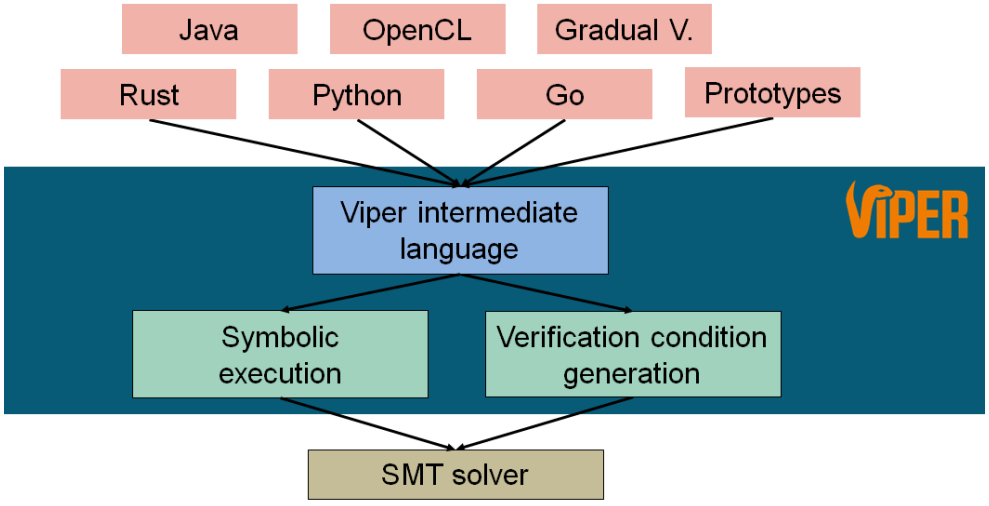

Figure 2.1: Viper architecture overview

The verifier that we aim at developing to automate HHL will generate encodings written in the Viper intermediate language to form a valid Viper program, which is then verified by the Silicon back-end of Viper. It is mentionable that, since valid Viper programs can be verified by both back-ends, we can easily integrate the Carbon back-end into our verifier in the future.

Chapter 3

Approach

This chapter presents our approach to automating HHL. On a high level, HHL is automated by tracking a set of program states from the beginning to the end of the program under verification. How the program states are tracked depends on whether we want to overapproximate or underapproximate the set of states.

In the following sections, we will present the language for which HHL is automated, and then elaborate on the main idea of the automation approach, followed by the exact Viper encodings of each command supported by our language.

3.1 Language Definition

We automate HHL for a simple imperative programming language shown below.

Definition 3.1 ***Programming Language.*** *Let us consider the following programming language, where C is a program command, x is a program variable, e is an expression, and b is a predicate describing program states.*

$$
\begin{aligned}
C ::= \ & \texttt{skip} \mid \texttt{havoc}\ x \mid \texttt{assume}\ b \mid \texttt{assert}\ b \\
\mid \ & x := e \ \textit{(Assignment)} \\
\mid \ & C; C \ \textit{(Sequential composition)} \\
\mid \ & C_1 + C_2 \ \textit{(Non-deterministic choice)} \\
\mid \ & C^* \ \textit{(Non-deterministic iteration)}
\end{aligned}
$$

In the language definition above, $C_1 + C_2$ non-deterministically executes either C_1 or C_2, while C^* non-deterministically executes for another iteration or stops. In practice, we will adapt these two statements to deterministic

conditional statements and while loops respectively as shown below:

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

The big-step semantics of all the commands above can be found in the paper [1] where HHL was first introduced.

3.2 General Approach

Given any program written in the language from Section 3.1, we develop encodings of the program into Viper to enable reasoning with HHL.

Recall that verifying some program C with HHL is equivalent to veriyfing the validity of the hyper-triple $\vDash [P]C[Q]$ (see Definition 2.3). To do so, given a set of program states satisfying some precondition, we incrementally update the set of states based on the sequence of commands in the program C, and verify whether the set of states in the end satisfies the postcondition. This indicates that, in order to automate HHL, we must automatically generate Viper encodings to keep track of the set of program states from the beginning to the end of the program under verification. In other words, given a set of initial states S which satisfies the precondition, we need to automatically compute $sem(C, S)$ (see Definition 2.2) – the set of states reachable by executing C in a state from S . Its definition implies that states in S and $sem(C, S)$ relate to each other in the following way: every state $\sigma' \in sem(C, S)$ results from executing C in some initial state $\sigma \in S$ as shown by Formula 3.1, and executing C in any initial state $\sigma \in S$ results in a state $\sigma' \in sem(C, S)$ as shown by Formula 3.2. Once the value of $sem(C, S)$ is expressed with Viper encodings, Viper can automatically check if it satisfies the postcondition to complete the verification.

$$\forall \sigma'.\ \sigma' \in sem(C, S) \implies \exists \sigma.\ \sigma \in S \wedge (\langle C, \sigma \rangle \rightarrow \sigma') \tag{3.1}$$
$$\forall \sigma, \sigma'.\ \sigma \in S \wedge (\langle C, \sigma \rangle \rightarrow \sigma') \implies \sigma' \in sem(C, S) \tag{3.2}$$

As mentioned in Section 2.1, this thesis focuses on automating two strict subsets of HHL, i.e. $\forall$-HHL and $\exists$-HHL, which require overapproximation and underapproximation reasoning respectively. When using overapproximation reasoning, we aim at proving that all states in $sem(C, S)$ satisfy some property such as SNI, which can be achieved by using Formula 3.1. In comparison, when reasoning with $\exists$-HHL, the goal is to prove that the states satisfying some property (such as the violation of SNI) exist in $sem(C, S)$, which can be done by leveraging Formula 3.2. We can notice that, in either case, only partial information of $sem(C, S)$ is required. Consequently, to

automate ∀-HHL (resp. ∃-HHL), it is sufficient to encode an approximation of $sem(C, S)$ by using Formula 3.1 (resp. Formula 3.2). These two formulas simply present a high-level idea on how we can model the approximation of $sem(C, S)$ based on the semantics of some command C. The way that the approximated $sem(C, S)$ is actually encoded for a concrete command C will not necessarily have the same shape as these formulas, but it will be logically equivalent to them. In the following sections, we will present the exact Viper encodings of all commands supported by our language presented in Section 3.1.

3.3 Viper Encodings of Commands

Let us recall that our verifier accepts input programs written in a language with the following commands: `skip`, `assume` b, `assert` b, `havoc` x, assignments, conditional statements and loops, where x is a program variable and b is a predicate describing program states. This section shows the ∀-HHL and ∃-HHL Viper encodings of each of these commands except for `skip` and loops. The `skip` command is trivial and does not need to be explictly encoded. As for loops, their encodings are the most complicated and will be presented in Chapter 5.

In general, the Viper encodings of each command C will have the pattern shown by Encoding 3.1. We use S to represent the set of program states immediately before C is executed, and use S_{tmp} to temporarily represent the approximation of $sem(C, S)$. To derive S_{tmp} based on the semantics of C, we use a Viper assertion A to relate the states in S_{tmp} and S. After S_{tmp} is determined, we update S with S_{tmp} so that the variable S now is the approximated $sem(C, S)$, and it will serve as the set of states in which the next command is executed.

```
havoc S_tmp
// A is a Viper assertion that describes what
    states should be in S_tmp based on the semantics
    of C. It is logically equivalent to either
    Formula 3.1 or Formula 3.2, depending on
    whether ∀-HHL or ∃-HHL is used.
assume A
S := S_tmp
```

Encoding 3.1: General form of Viper encodings of any command C

3.3.1 Encodings of Sequential Compositions

A sequential composition of two commands $C_1; C_2$ is encoded as shown by Encoding 3.2. It is simply a sequential composition of the Viper encodings of commands C_1 and C_2. We can notice that, since we always remove prior

knowledge of S_{tmp} before encoding each command, the semantics of C_1 do not affect the encoding of C_2.

```
1  // Encoding of C1
2  havoc S_tmp
3  // A1 is a Viper assertion that describes what
       states should be in S_tmp based on the semantics
       of C1
4  assume A1
5  S := S_tmp
6  // Encoding of C2
7  havoc S_tmp
8  // A2 is a Viper assertion that describes what
       states should be in S_tmp based on the semantics
       of C2
9  assume A2
10 S := S_tmp
```

Encoding 3.2: ∀-HHL and ∃-HHL encodings of $C_1; C_2$

3.3.2 Encodings of `assume` b

Encodings 3.3 and 3.4 show the ∀-HHL and ∃-HHL encodings of the statement `assume` b respectively. In the Encodings, we use the notation $b(\sigma)$ to represent that the expression b is evaluated to `true` in the state σ. The following encodings filter out the states where b does not hold, which is consistent with the big-step semantics of `assume` b [1].

```
1  havoc S_tmp
2  assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ b(σ)
3  S := S_tmp
```

Encoding 3.3: ∀-HHL encoding of `assume b`

```
1  havoc S_tmp
2  assume ∀σ. σ ∈ S ∧ b(σ) ⟹ σ ∈ S_tmp
3  S := S_tmp
```

Encoding 3.4: ∃-HHL encoding of `assume b`

3.3.3 Encodings of `assert` b

The Viper encodings of the statement `assert` b are almost identical to those of `assume` b, except that the states where b does not hold are stored in a variable S_{fail}. S_{fail} is assumed to be an empty set at the beginning of the verification. It is continuously updated with a set union operation, which is encoded with uninterpreted Viper functions and axioms, to store all states where the asserted expression b does not hold during verification.

1 `havoc` S_{tmp}
2 `havoc` S_{fail_tmp}
3 `assume` $\forall\sigma.\ \sigma \in S_{tmp} \implies \sigma \in S \wedge b(\sigma)$
4 `assume` $\forall\sigma.\ \sigma \in S_{fail\_tmp} \implies \sigma \in S \wedge \neg b(\sigma)$
5 S `:=` S_{tmp}
6 S_{fail} `:=` $S_{fail} \bigcup S_{fail\_tmp}$

Encoding 3.5: ∀-HHL encoding of `assert b`

1 `havoc` S_{tmp}
2 `havoc` S_{fail_tmp}
3 `assume` $\forall\sigma.\ \sigma \in S \wedge b(\sigma) \implies \sigma \in S_{tmp}$
4 `assume` $\forall\sigma.\ \sigma \in S \wedge \neg b(\sigma) \implies \sigma \in S_{fail\_tmp}$
5 S `:=` S_{tmp}
6 S_{fail} `:=` $S_{fail} \bigcup S_{fail\_tmp}$

Encoding 3.6: ∃-HHL encoding of `assert b`

3.3.4 Encodings of `havoc` x

The semantics of the statement `havoc` x is to allow x to be updated with an arbitrary value in all states while preserving the values of all other variables in those states.

Let σ, σ' be an arbitrary state in the set of states immediately before and after the execution of this statement respectively. We define a predicate $\mathcal{P}(\sigma, \sigma', x) \triangleq (\forall v.\ v \neq x \implies \sigma(v) = \sigma'(v))$. It expresses that σ and σ' agree on the values of all variables except for x. By using this predicate, we can encode `havoc` x as shown below. We can observe that the value of x is left unconstrained in any state in both the ∀-HHL and ∃-HHL encodings of this statement, which complies with its semantics.

As a side note, if S is initially a non-empty set, Encoding 3.8 allows us to theoretically prove that $\exists\sigma \in S.\ \sigma(x) = k$ where k is any integer after line 3. However, since existential quantifications are hard for SMT-based verifiers like Viper to automatically prove in practice, such a property cannot be automatically verified without hints provided by users. We will discuss this in more detail in Section 4.3.

1 `havoc` S_{tmp}
2 `assume` $\forall\sigma.\ \sigma \in S_{tmp} \implies \exists\sigma_0.\ \sigma_0 \in S \wedge \mathcal{P}(\sigma, \sigma_0, x)$
3 S `:=` S_{tmp}

Encoding 3.7: ∀-HHL encoding of `havoc x`

1 `havoc` S_{tmp}
2 `assume` $\forall\sigma, k.\ \sigma \in S \implies \exists\sigma_1.\ \sigma_1 \in S_{tmp} \wedge \mathcal{P}(\sigma, \sigma_1, x) \wedge \sigma_1(x) = k$
3 S `:=` S_{tmp}

Encoding 3.8: ∃-HHL encoding of `havoc x`

3.3.5 Encodings of Assignments $x := e$

The encodings of an assignment $x := e$ are very similar to those of `havoc` x, which is caused by the similarity between the semantics of these two statements. Instead of updating x to an arbitrary value, the encodings of the assignment $x := e$ update x with a deterministic value in every state. Let us denote the evaluated result of the expression e in a state σ as $e(\sigma)$. Then we can derive the encodings of the assignment as shown below.

```
1  havoc S_tmp
2  assume ∀σ. σ ∈ S_tmp ⟹
       (∃σ_0. σ_0 ∈ S ∧ P(σ, σ_0, x)∧
       σ(x) = e(σ_0))
3  S := S_tmp
```

Encoding 3.9: ∀-HHL encoding of `x:=e`

```
1  havoc S_tmp
2  assume ∀σ. σ ∈ S ⟹
       (∃σ_1. σ_1 ∈ S_tmp ∧ P(σ, σ_1, x)∧
       σ_1(x) = e(σ))
3  S := S_tmp
```

Encoding 3.10: ∃-HHL encoding of `x:=e`

3.3.6 Encodings of Conditional Statements

```
1  S_1 := S
2  S_2 := S
3  // Encoding of the if branch
4  havoc S_tmp
5  assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S_1 ∧ b(σ)
6  S_1 := S_tmp
7  // Update S_1 based on the
       semantics of C_1
8  ...
9  // Encoding of the else branch
10 havoc S_tmp
11 assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S_2 ∧ ¬b(σ)
12 S_2 := S_tmp
13 // Update S_2 based on the
       semantics of C_2
14 ...
15 S_tmp := S_1 ∪ S_2
16 S := S_tmp
```

Encoding 3.11: ∀-HHL encoding of `if(b){C1}else{C2}`

```
1  S_1 := S
2  S_2 := S
3  // Encoding of the if branch
4  havoc S_tmp
5  assume ∀σ. σ ∈ S_1 ∧ b(σ) ⟹ σ ∈ S_tmp
6  S_1 := S_tmp
7  // Update S_1 based on the
       semantics of C_1
8  ...
9  // Encoding of the else branch
10 havoc S_tmp
11 assume ∀σ. σ ∈ S_2 ∧ ¬b(σ) ⟹ σ ∈ S_tmp
12 S_2 := S_tmp
13 // Update S_2 based on the
       semantics of C_2
14 ...
15 S_tmp := S_1 ∪ S_2
16 S := S_tmp
```

Encoding 3.12: ∃-HHL encoding of `if(b){C1}else{C2}`

As explained in Section 3.1, a conditional statement `if`$(b)\{C_1\}$`else`$\{C_2\}$ is equivalent to (`assume` b; C_1) + (`assume` $\neg b$; C_2) where + represents a non-deterministic choice. Let S_1 be the set of states that execute (`assume` b; C_1) and S_2 be the set of states that execute (`assume` $\neg b$; C_2). By definition of non-determinism, the set of states after the if-statement is executed should be $S_1 \bigcup S_2$. By using this idea, we can encode the conditional statement as shown by Encodings 3.11 and 3.12.

We would like to mention that, in the case of nested conditional statements, we will not reuse S_1 and S_2 to respectively represent the set of states executing the nested `if` branch and the nested `else` branch. Instead, we will use fresh variables with different identifiers, such as S_3 and S_4, to represent them.

Chapter 4

Verification Primitives

In this chapter, we introduce the primitives that are useful for guiding the verifier to automatically verify programs with HHL. These primitives are not program commands but enable users to provide verification guidance.

In the following sections, we first present the primitives that allow users to assume and assert hyper-assertions, followed by the primitive that enables synchronous reasoning about conditional statements. We also elaborate on the primitives with which users can provide hints and introduce proof variables. In the end, we show a primitive that allows users to explicitly frame a property around program commands. For each of these primitives, we will present what it does, why it is useful, and how it is encoded.

4.1 Assuming and Asserting Hyper-Assertions

To make it easier for users to debug programs, we introduce statements that allow users to assume and assert hyper-assertions.

To add an assumption about the current set of program states to the program under verification, users may use a `hyperAssume` statement with the syntax `hyperAssume A`, where A is a hyper-assertion. Its semantics are summarized by Rule 4.1. To encode this statement, since A is already a hyper-assertion, we can simply assume it with a Viper statement `assume A`.

$$\frac{}{\vdash [P]\ \texttt{hyperAssume A}\ [P \wedge A]}$$

Rule 4.1: Rule for `hyperAssume A`

On the other hand, if users would like to check whether the current set of program states satisfy some property, they may opt for a `hyperAssert` statement with the syntax `hyperAssert A`, where A is a hyper-assertion. Its

semantics are summarized by Rule 4.2. To encode this statement, since A is already a hyper-assertion, we can simply assert it with a Viper statement `assert A`.

$$\frac{P \models A}{\vdash [P]\ \texttt{hyperAssert A}\ [P]}$$

Rule 4.2: Rule for `hyperAssert A`

4.2 Synchronous Reasoning about Conditional Statements

According to the HHL If Rule [1], two branches of a conditional statement must be reasoned about separately, which makes it difficult to verify some programs. Let us consider Program 4.1. Notice that the program contains a method with a precondition and a postcondition. Although our language does not support method calls at the moment, we still require program commands to be put inside a method. This allows users to directly express the precondition and postcondition as method specifications. It also lays a foundation for incorporating modular reasoning about method calls in the future. In addition, we may also observe that both the precondition and postcondition of the method of Program 4.1 quantify over states in S, but S actually has different values at these two locations. Recall that S is incrementally updated in the Viper encodings of the program, so its value in the precondition is the set of initial states, while its value in the postcondition is the set of final states.

Imagine that we would like to prove that Program 4.1 is monotonic with $\forall$-HHL. Since monotonicity is a 2-safety property, the program precondition and postcondition must be hyper-assertions that relate any two program executions. To do so, we need to introduce a logical variable t to identify the states corresponding to execution 1 and the states corresponding to execution 2. By using the variable t and the definition of monotonicity, we can formulate the program precondition and postcondition as shown by lines 3 and 4 respectively.

In addition, since the program contains loops, we also need to provide loop invariants for the program to be automatically verified. Let us consider the loop in the `if` branch. By manually analyzing the loop body, we may derive the loop invariants as shown by lines 12-13. With these loop invariants and the negated loop guard, the verifier can easily prove that, at the end of the `if` branch, any two states σ_1, σ_2 satisfy $\sigma_1(y) \leq \sigma_2(y)$. Since this loop is identical to the loop in the `else` branch, the verifier can also prove that, at the end of the `else` branch, any two states σ_1, σ_2 satisfy $\sigma_1(y) \leq \sigma_2(y)$. However, this

information does not allow the verifier to prove the postcondition, because the verifier does not know the relation between $\sigma_1(y)$ and $\sigma_2(y)$ where σ_1 comes from the `if` branch and σ_2 comes from the `else` branch.

```
1  method sync(x: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(x) ≤ σ2(x)
4      ensures ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(y) ≤ σ2(y)
5  {
6      var ind: Int
7      ind := 0
8      x' := x
9      y := 1
10     if (x > 0) {
11         while (ind < 5)
12             invariant ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2
                   ⟹ σ1(x') ≤ σ2(x') ∧ σ1(y) ≤ σ2(y)
13             invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ 5 ∧ σ(ind) = n_iter
14         {
15             x' := x' * 2 - 1
16             y := x' + 1
17             ind := ind + 1
18         }
19         y := y + 1
20     } else {
21         while (ind < 5)
22             invariant ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2
                   ⟹ σ1(x') ≤ σ2(x') ∧ σ1(y) ≤ σ2(y)
23             invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ 5 ∧ σ(ind) = n_iter
24         {
25             x' := x' * 2 - 1
26             y := x' + 1
27             ind := ind + 1
28         }
29     }
30 }
```

Program 4.1: A program where a loop with relational invariants is duplicated, where n_{iter} refers to the number of executed loop iterations – does not verify

This problem can be solved by synchronously reasoning about the block of code that is common in both branches. To do so, users may use the `declare-reuse` primitive in a conditional statement of the form `if (b) {C1;C;C1'} else {C2;C;C2'}` with the syntax shown in Listing 4.2, where the code block common in both branches is identified with `<block_id>`.

```
if (b) {
    C1
    declare <block_id> { C }
    C1'
} else {
    C2
    reuse <block_id>
```

```
        C2'
}
```

Listing 4.2: Syntax of using the `declare-reuse` primitive

The semantics of this primitive are summarized by the Synchronous If Rule [1]. More specifically, it allows us to reason independently about the different parts of the two branches, namely commands C_1 and C_2, and then synchronously reason about the common command C by combining the sets of program states from both branches. Finally, we resume independent reasoning to reason about C_1', C_2' respectively.

```
1  method sync(x: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(x) ≤ σ2(x)
4      ensures ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(y) ≤ σ2(y)
5  {
6      var ind: Int
7      ind := 0
8      x' := x
9      y := 1
10     if (x > 0) {
11         declare commonLoop {
12             while (ind < 5)
13                 invariant ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2
                       ⟹ σ1(x') ≤ σ2(x') ∧ σ1(y) ≤ σ2(y)
14                 invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ 5 ∧ σ(ind) = n_iter
15             {
16                 x' := x' * 2 - 1
17                 y := x' + 1
18                 ind := ind + 1
19             }
20         }
21         y := y + 1
22     } else {
23         reuse commonLoop
24     }
25 }
```

Program 4.3: A program where a loop with relational invariants is reused with `delcare-reuse` primitive, where n_{iter} refers to the number of executed loop iterations

```
1  // Independent reasoning of C1 in S1, C2
       in S2
2  ...
3  // Synchronous reasoning starts
4  havoc S_tmp
5  assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S_1 ∧ σ(i) = 0
6  S_1 := S_tmp
7
8  havoc S_tmp
9  assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S_2 ∧ σ(i) = 1
10 S_2 := S_tmp
11 S := S_1 ∪ S_2
12 // Reason about C in S
13 ...
14 // Synchronous reasoning ends
15 havoc S_tmp
16 assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ σ(i) = 0
17 S_1 := S_tmp
18
19 havoc S_tmp
20 assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ σ(i) = 1
21 S_2 := S_tmp
22 // Independent reasoning of C1', C2'
23 ...
```

Encoding 4.4: ∀-HHL encoding of conditional statements containing a `declare-reuse` primitive

```
1  // Independent reasoning of C1 in
       S1, C2 in S2
2  ...
3  // Synchronous reasoning starts
4  assume ∀σ. σ ∈ S_1 ⟹ σ(i) = 0
5  assume ∀σ. σ ∈ S_2 ⟹ σ(i) = 1
6  S := S_1 ∪ S_2
7  // Reason about C in S
8  ...
9
10 // Synchronous reasoning ends
11
12 // Prepare for resuming independent
       reasoning
13 havoc S_tmp
14 assume ∀σ. σ ∈ S ∧ σ(i) = 0 ⟹ σ ∈ S_tmp
15 S_1 := S_tmp
16
17 havoc S_tmp
18 assume ∀σ. σ ∈ S ∧ σ(i) = 1 ⟹ σ ∈ S_tmp
19 S_2 := S_tmp
20
21 // Independent reasoning of C1', C2'
22 ...
```

Encoding 4.5: ∃-HHL encoding of conditional statements containing a `declare-reuse` primitive

By using the `declare-reuse` primitive, Program 4.1 can be updated to Program 4.3 where the loop is synchronously reasoned about. In this way, the verifier can easily prove that the relational invariant from line 13 holds in all states exiting the loop regardless of which branch they belong to, which makes it much easier than before to prove the postcondition.

Apart from loops with relational invariants, synchronous reasoning is also useful when both the `if` and `else` branches call a method with relational specifications. By synchronously reasoning about the method call, we can leverage the relational specifications of the method to efficiently verify hyperproperties.

To enable the synchronous reasoning described above, Encoding 4.4 (resp. Encoding 4.5) needs to be integrated into the general encodings of conditional statements, i.e. Encoding 3.11 (resp. Encoding 3.12), upon detection of a `declare-reuse` primitive. Observe that a fresh logical variable i is introduced in Encodings 4.4 and 4.5. Its purpose in both encodings is to allow us to go back to independent reasoning after synchronous reasoning ends. More specifically, before combining S_1 and S_2, which are the set of states entering the `if` branch and the `else` branch respectively, we mark any state $\sigma \in S_1$ by letting $\sigma(i) = 0$ and mark any state $\sigma \in S_2$ by letting $\sigma(i) = 1$. After

synchronous reasoning, we distinguish the states belonging to the `if` branch from those belonging to the `else` branch based on the value of i in the states and then resume independent reasoning. Since the logical variable i is never modified in the program under verification, such reasoning is guaranteed to be sound.

4.3 Hint Declarations and Uses

Consider the following program with a single `havoc` statement:

```
1 method havoc_example(x: Int)
2     requires ∃σ. σ ∈ S
3     ensures ∃σ. σ ∈ S ∧ σ(x) = 1
4 {
5     havoc x
6 }
```

Program 4.6: A program with a single `havoc` statement – does not verify without hints

As mentioned in Section 3.3.4, the body of the program is encoded as follows when reasoning with ∃-HHL:

```
1 havoc S_tmp
2 assume ∀σ, k. σ ∈ S ⟹ ∃σ_1. σ_1 ∈ S_tmp ∧ P(σ, σ_1, x) ∧ σ_1(x) = k
3 S := S_tmp
```

Encoding 4.7: ∃-HHL encoding of the body of Program 4.6 where $\mathcal{P}(\sigma, \sigma_1, x) \triangleq (\forall v.\ v \neq x \implies \sigma(v) = \sigma_1(v))$

In theory, Encoding 4.7 has provided us with sufficient information to verify Program 4.6. However, in practice, SMT-based verifiers like Viper do not know how to instantiate line 2 of Encoding 4.7 for any integer k, which prevents us from proving the postcondition.

To solve this problem, we enable users to explicitly specify the value with which the encoding of a `havoc` statement should be instantiated. To do so, users first need to declare a hint, identified as `<hint_id>`, as part of the `havoc` statement as shown in Listing 4.8. This will add a trigger to the encoding of the corresponding `havoc` statement. A trigger is essentially a syntactic cue that provides guidance on how the quantifiers in an assertion should be instantiated. When it comes to using the declared hint, users may add a hint use statement as shown in Listing 4.9 to select a concrete value v so that the corresponding `havoc` statement will be encoded with $k = v$. We will explain in more detail about how hint declarations and uses are encoded in Section 6.4.2.

```
havoc x {<hint_id>}
```

Listing 4.8: Syntax of declaring a hint in a `havoc` statement

```
use <hint_id>(v)
```

Listing 4.9: Syntax of using a single value v as a hint

With the primitives described above, Program 4.6 can be updated to Program 4.10 whose encoding is shown in Encoding 4.11. The updated encoding of the program allows the verifier to easily prove the postcondition.

```
1 method havoc_example(x: Int)
2     requires ∃σ. σ ∈ S
3     ensures ∃σ. σ ∈ S ∧ σ(x) = 1
4 {
5     havoc x {hint_x}
6     use hint_x(1)
7 }
```

Program 4.10: A program with a single `havoc` statement – verifies

```
1 havoc S_tmp
2 // Notice that the following assertion now contains
      a trigger
3 assume ∀σ, k. { hint_x(k)} σ ∈ S ⟹
                ∃σ_1. σ_1 ∈ S_tmp ∧ P(σ, σ_1, x) ∧ σ_1(x) = k
4 S := S_tmp
5 // Encoding of use hint_x(1)
6 assume hint_x(1)
```

Encoding 4.11: $\exists$-HHL encoding of the body of Program 4.10, where $\mathcal{P}(\sigma, \sigma', x) \triangleq (\forall v.\ v \neq x \implies \sigma(v) = \sigma'(v))$

In some cases, instead of a single value, users might want to instantiate the `havoc` statement encodings with a range of values that satisfy some condition b. This can be achieved by using the syntax in Listing 4.12.

```
use ∀v. b(v) ⟹ <hint_id>(v)
```

Listing 4.12: Syntax of using a range of values as hints

Nevertheless, since such a hint use statement itself also involves quantifier instantiations, it does not always trigger the encodings of the corresponding `havoc` statement as expected. To tackle this limitation, we also allow users to add hints to a hyper-assertion when they would like to instantiate the encodings with a range of values, as shown by line 3 of Program 4.13 below. In theory, using hints like this is equivalent to adding a hint use statement `use` $\forall v.\ 0 < v < 9 \implies$ `hint_x(`v`)` to the program, but the latter statement does not have the desired behavior in practice.

```
1 method randomNumGen()
2     returns (x: Int)
3     requires ∃σ. σ ∈ S
```

```
4     ensures ∀n. n < 9 ∧ n > 0 ⟹ hint1(n) ∧ (∃σ. σ ∈ S ∧ σ(x) = n)
5 {
6      havoc x {hint1}
7      assume x < 9 && x > 0
8 }
```

Program 4.13: A program that randomly generates an integer between 0 and 9 – verifies

Apart from `havoc` statements, hints can also aid the verifier to automatically reason about loops with ∃-HHL in practice. To declare a hint as part of a loop invariant, users may use the syntax shown in Listing 4.14. We will explain why hints are useful and how to use them when reasoning about loops in the next chapter.

```
while (b)
{<hint_id>} invariant A
{ C }
```

Listing 4.14: Syntax of declaring a hint in a loop invariant

4.4 Proof Variables

We allow users to declare proof variables to aid the verification process. They are especially useful for reasoning about loops with ∃-HHL, which we will elaborate in the next chapter. Similar to program variables, proof variables can be referred to by users after they are declared. Unlike program variables, proof variables are not part of the program under verification and cannot be modified after declaration. Their semantics are summarized by Rule 4.3.

$$\frac{\forall x.(\vdash [P(x)]\ C\ [Q(x)])}{\vdash [\exists x.P(x)]\ C\ [\exists x.Q(x)]}$$

Rule 4.3: Rule for proof variable x

There are two ways to declare a proof variable. Firstly, users may introduce a proof variable `$p` such that $P($`p`$)$, which is a predicate dependent on `$p`, holds with the following syntax:

```
let $p: <type_name> :: P($p)
```

Listing 4.15: Syntax of declaraing a proof variable that satisfies some property P

This declaration is equivalent to saying "given that there exists a value that satisfies some property P, denote that value as `$p`", which is encoded as follows:

```
1  assert ∃p0. P(p0)
2  var $p: <type_name>
3  assume P($p)
```

Encoding 4.16: ∀-HHL and ∃-HHL encoding of `let $p: <type_name> :: P($p)`

As Encoding 4.16 shows, we must first assert that there indeed exists some variable p_0 such that the property $P(p_0)$ holds. In order for users to refer to this variable later in the program, it must be explicitly declared, with identifier `$p` and type `<type_name>`, like a program variable. Since `$p` is a fresh variable, we must add the information that the property P holds for `$p` by using line 3 of Encoding 4.16.

Alternatively, as shown below, users can declare a proof variable `$p` whose value equals the evaluated result of some expression `<exp>`.

```
let $p = <exp>
```

Listing 4.17: Syntax of declaring a proof variable whose value is `<exp>`

The semantics of such a declaration directly imply the following encoding:

```
1  var $p: <type_name>
2  $p := <exp>
```

Encoding 4.18: ∀-HHL and ∃-HHL encoding of `let $p = <exp>`

4.5 Frames

By using the syntax in Listing 4.19, users can express that some property, described by a hyper-assertion A, is preserved across some command C. Under most circumstances, the verifier does not need such an explicit `frame` statement to automatically prove the preservation of properties across C, except when C contains a loop. We will explain the reason for this when we present the encodings for loops in the next chapter.

```
frame (A) {C}
```

Listing 4.19: Syntax of framing a property around some command

Rule 4.4 below summarizes the semantics of this primitive. For it to be sound, the hyper-assertion A must universally quantify over program states and have no other interaction with the current set of program states. In addition, A must not involve program variables that are modified in C. These restrictions will be checked syntactically in the implementation.

We can encode this primitive as shown in Encoding 4.20. First, we must assert that the hyper-assertion A holds in the set of program states before C is executed. Next, because C does not modify any program variables

$$\frac{\vdash [P]\ C\ [Q] \quad \text{A}= \forall \sigma_1, ..., \sigma_n \in S.\ b(\sigma_1\rangle, ..., \langle\sigma_n)}{[P \wedge A]\ \texttt{frame(A)\{C\}}\ [Q \wedge A]}\ freeVars(A) \cap mod(C) = \emptyset$$

Rule 4.4: Rule for `frame A{C}`, where b is a predicate that has no other interaction with the current set of states, $freeVars(A)$ is the set of free program variables involved in A and $mod(C)$ is the set of program variables modified in C

dependent by A, we can soundly assume that A holds in the set of program states after C is executed.

```
1 assert A
2 // Encoding for C starts
3 ...
4 // Encoding for C ends
5 assume A
```

Encoding 4.20: $\forall$-HHL and $\exists$-HHL encoding of `frame` A `{C}`

Chapter 5

Loops

In general, it is non-trivial to reason about loops automatically. This chapter is dedicated for explaining how we design the encodings of loops. It is structured as follows: we first present the general encodings of loops based on the HHL loop rule, and then, by using concrete examples, we show the necessity of providing the verifier with extra guidance on top of the general encodings. In addition, we demonstrate how users can provide such guidance by using verification primitives including proof variables, frames and hints mentioned in the last chapter.

5.1 General Approach

Recall that, in our language, a loop has the syntax shown by Listing 5.1, where b is the loop condition, C is the loop body, and I is an optional hyper-assertion which, if it exists, serves as a loop invariant.

```
while (b)
    invariant I // Optional
{
    C
}
```

Listing 5.1: Syntax of a loop

We encode loops like this based on Rule 5.1 [1]. In this rule, $I(n_{iter})$ represents the loop invariant after n_{iter} iterations of the loop. The premise of the rule states that the invariant $I(n_{iter})$ must be inductive. When this premise is obtained, we can conclude that executing a non-deterministic loop C^*, which, in practice, is adapted to a normal loop with a loop condition as shown above, in a set of initial states satisfying $I(0)$ will result in a set of final states satisfying $\bigotimes I$. More specifically, this means that there exist sets $S_0, S_1, ..., S_k$ such that the set of final states $S = \bigcup_{k=0,1,...,\infty} S_k$, where S_k is the set of states that execute k loop iterations and satisfies $I(k)$.

$$\frac{\vdash [I(n_{iter})]\ C\ [I(n_{iter}+1)]}{\vdash [I(0)]\ C^*\ [\bigotimes I]}$$

Rule 5.1: Rule for loops

By following Rule 5.1, we have designed the ∀-HHL and ∃-HHL encodings of loops as shown by Encodings 5.2 and 5.3 respectively, where we use the notation $I(n_{iter}, S)$ to represent that $I(n_{iter})$ holds in a set of states S. First, we assert that $I(0)$ holds in S (line 1) and let S_0, the set of states where 0 loop iteration is executed, equal to S (line 2). Next, to verify that I is inductive, we forget all information about S and n_{iter} except that n_{iter} is non-negative (lines 3 - 5), and then non-deterministically check that, given that $I(n_{iter})$ and b holds in S, $I(n_{iter}+1)$ holds in the set of states after one iteration of the loop is executed in S (lines 6 - 19). After the loop, we encode the set of states to be a union of all S_k's such that $I(k)$ holds in each S_k (lines 21 - 22). Finally, since the loop is terminated at this point, we assume that the set of final states satisfies $\neg b$ (lines 24 - 26).

```
1  assert I(0, S)
2  assume S = S_0
3  havoc S
4  havoc n_iter
5  assume n_iter ≥ 0
6  if (*) {
7      assume I(n_iter, S)
8      // Assume that the loop condition b
           holds
9      havoc S_tmp
10     assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ b(σ)
11     S := S_tmp
12
13     // Encoding of the loop body C starts
14     ...
15     // Encoding of the loop body C ends
16
17     assert I(n_iter + 1, S)
18     assume false
19 }
20 // Let S be a union of S_ks
21 havoc S
22 assume ∀σ. σ ∈ S ⟹ ∃k. k ≥ 0 ∧ σ ∈ S_k ∧ I(k, S_k)
23 // Assume that the loop condition b does
       not hold
24 havoc S_tmp
25 assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ ¬b(σ)
26 S := S_tmp
```

Encoding 5.2: ∀-HHL encoding of a loop based on Rule 5.1

```
1  assert I(0, S)
2  assume S = S_0
3  havoc S
4  havoc n_iter
5  assume n_iter ≥ 0
6  if (*) {
7      assume I(n_iter, S)
8      // Assume that the loop condition b
           holds
9      havoc S_tmp
10     assume ∀σ. σ ∈ S ∧ b(σ) ⟹ σ ∈ S_tmp
11     S := S_tmp
12
13     // Encoding of the loop body C starts
14     ...
15     // Encoding of the loop body C ends
16
17     assert I(n_iter + 1, S)
18     assume false
19 }
20 // Let S be a union of S_ks
21 havoc S
22 assume ∀k, σ. k ≥ 0 ∧ σ ∈ S_k ∧ I(k, S_k) ⟹ σ ∈ S
23 // Assume that the loop condition b does
       not hold
24 havoc S_tmp
25 assume ∀σ. σ ∈ S ∧ ¬b(σ) ⟹ σ ∈ S_tmp
26 S := S_tmp
```

Encoding 5.3: ∃-HHL encoding of a loop based on Rule 5.1

As an example, let us consider Program 5.4 that uses a loop to compute the product of x and m. To verify this program with ∀-HHL, the verifier generates Encoding 5.5. We may observe that, in the encoding, $I(0)$, $I(n_{iter})$ and $I(n_{iter}+1)$ are identical. This is because the loop invariant of this program does not depend on n_{iter}. With Encoding 5.5, the verifier successfully verifies the program.

Nevertheless, Encodings 5.2 and 5.3 are not sufficient for reasoning about loops in general. As shown by the rest of the chapter, it is necessary for users to use verification primitives to provide the verifier with additional guidance.

```
1  method loop_ex1(x: Int, m: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0 ∧ σ(m) > 0
4      ensures ∀σ. σ ∈ S ⟹ σ(y) = σ(x) * σ(m)
5  {
6      var ind: Int
7      ind := 0
8      y := 0
9      while (ind < m)
10         invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(y) = σ(x) * σ(ind)
11     {
12         y := y + x
13         ind := ind + 1
14     }
15 }
```

Program 5.4: A program whose specifications describe a trace property – verifies

```
1  method loop_ex1_encoding(x: Int, m: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0 ∧ σ(m) > 0
4      ensures ∀σ. σ ∈ S ⟹ σ(y) = σ(x) * σ(m)
5  {
6      var ind: Int
7      // Encoding of ind := 0
8      havoc S_tmp
9      assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S ∧ P(σ, σ0, ind) ∧ σ(ind) = 0
10     S := S_tmp
11     // Encoding of y := 0
12     havoc S_tmp
13     assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S ∧ P(σ, σ0, y) ∧ σ(y) = 0
14     S := S_tmp
15     var n_iter: Int
16     // Assert I(0, S)
17     assert ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(y) = σ(x) * σ(ind)
18     assume S = S0
19     havoc S
20     havoc n_iter
21     assume n_iter ≥ 0
22     if (*) {
23         // Assume I(n_iter, S)
```

```
24          assume
                ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(y) = σ(x) ∗ σ(ind)
25          // Assume the loop condition
26          havoc S_tmp
27          assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ σ(ind) < σ(m)
28          S := S_tmp
29          // Encoding of the loop body
30          havoc S_tmp
31          assume
                ∀σ. σ ∈ S_tmp ⟹ ∃σ_0. σ_0 ∈ S ∧ P(σ, σ_0, y) ∧ σ(y) = σ_0(y) + σ_0(x)

32          S := S_tmp
33          havoc S_tmp
34          assume
                ∀σ. σ ∈ S_tmp ⟹ ∃σ_0. σ_0 ∈ S ∧ P(σ, σ_0, ind) ∧ σ(ind) = σ_0(ind) + 1

35          S := S_tmp
36          // Assert I(n_iter + 1, S)
37          assert
                ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(y) = σ(x) ∗ σ(ind)
38          assume false
39      }
40      // Let S be a union of S_k's
41      havoc S
42      assume ∀σ. σ ∈ S ⟹ ∃k. k ≥ 0 ∧ σ ∈ S_k ∧
            (∀σ. σ ∈ S_k ⟹ σ(ind) ≤ σ(m) ∧ σ(y) = σ(x) ∗ σ(ind))
43      // Assume the negated loop condition
44      havoc S_tmp
45      assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ σ(ind) ≥ σ(m)
46      S := S_tmp
47  }
```

Encoding 5.5: ∀-HHL Encoding of Program 5.4

5.2 Framing

In this section, we explain why we need to frame information about program states around loops and the two ways to do so, namely explicit framing and automatic framing.

5.2.1 Using Explicit Framing

Imagine that we would like to prove that Program 5.4 is monotonic when all initial states agree on the value of m which is known to be positive. By updating the program precondition and postcondition, we obtain Program 5.6. Since the updated postcondition relates two program states, it is desirable to write the loop invariant as relational hyper-assertions as shown by lines 11 - 12 of Program 5.6.

Nevertheless, with the current loop encoding (i.e. Encoding 5.2), the verifier cannot prove the postcondition. This is because, after the loop, it loses the information that $\sigma_1(m) = \sigma_2(m)$ where σ_1 and σ_2 are two states in S.

```
1  method loop_ex2(x: Int, m: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0 ∧ σ(m) > 0
4      requires ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹
                    σ1(x) ≤ σ2(x) ∧ σ1(m) = σ2(m)
5      ensures ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(y) ≤ σ2(y)
6  {
7      var ind: Int
8      ind := 0
9      y := 0
10     while (ind < m)
11         invariant ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹
                        σ1(x) ≤ σ2(x) ∧ σ1(y) ≤ σ2(y)   // I1
12         invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(ind) = n_iter   // I2
13     {
14         y := y + x
15         ind := ind + 1
16     }
17 }
```

Program 5.6: A program whose specifications describe monotonicity – does not verify without framing

To solve this problem, we might be tempted to add the missing information, i.e. $\forall\sigma_1, \sigma_2.\ \sigma_1 \in S \wedge \sigma_2 \in S \wedge \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \implies \sigma_1(m) = \sigma_2(m)$, to the loop invariant. This will make the set of program states after the loop be encoded as shown by Encoding 5.7. With this encoding, the verifier knows $\sigma_1(m) = \sigma_2(m)$ only when σ_1 and σ_2 belong to the same S_k, so it still cannot prove the postcondition. This problem indicates that it is necessary to frame information about program states around the loop, which leads to the introduction of the `frame` primitive mentioned in the last chapter.

```
1      // Let S be a union of S_k's
2      havoc S
3      assume ∀σ. σ ∈ S ⟹ ∃k. k ≥ 0 ∧ σ ∈ S_k ∧
           (∀σ. σ ∈ S_k ⟹ σ(y) ≤ σ(x)) ∧
           (∀σ1, σ2. σ1 ∈ S_k ∧ σ2 ∈ S_k ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(x) ≤ σ2(x) ∧
               σ1(y) ≤ σ2(y) ∧ σ1(m) = σ2(m)) ∧   // Modified I1(k, S_k)
           (∀σ. σ ∈ S_k ⟹ σ(ind) ≤ σ(m) ∧ σ(ind) = k)   // I2(k, S_k)
4      // Assume the negated loop condition
5      havoc S_tmp
6      assume ∀σ. σ ∈ S ⟹ σ(y) ≥ σ(x)
7      S := S_tmp
```

```
8  }
```

Encoding 5.7: ∀-HHL encoding of the set of states after the loop in Program 5.6, with the loop invariant modified as described by the paragraph above

By using the `frame` primitive, we update Program 5.6 to Program 5.8 shown below. The encoding of Program 5.8 is shown by Encoding 5.9. Thanks to the `frame` primitive, the information $\sigma_1(m) = \sigma_2(m)$ where $\sigma_1, \sigma_2 \in S$ is available after the loop, so the verifier can successfully verify that the program is monotonic.

```
1  method loop_ex3(x: Int, m: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0 ∧ σ(m) > 0
4      requires ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹
                    σ1(x) ≤ σ2(x) ∧ σ1(m) = σ2(m)
5      ensures ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(y) ≤ σ2(y)
6  {
7      var ind: Int
8      ind := 0
9      y := 0
10     frame ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(m) = σ2(m)
11     {
12         while (ind < m)
13             invariant ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹
                             σ1(x) ≤ σ2(x) ∧ σ1(y) ≤ σ2(y)
14             invariant ∀σ. σ ∈ S ⟹ σ(ind) ≤ σ(m) ∧ σ(ind) = n_iter
15         {
16             y := y + x
17             ind := ind + 1
18         }
19     }
20 }
```

Program 5.8: A program adapted from updating Program 5.6 with a `frame` statement – verifies

```
1  method loop_ex3_encoding(x: Int, m: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0 ∧ σ(m) > 0
4      requires ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹
                    σ1(x) ≤ σ2(x) ∧ σ1(m) = σ2(m)
5      ensures ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(y) ≤ σ2(y)
6  {
7      var ind: Int
8      // Encoding of ind := 0
9      havoc S_tmp
10     assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S ∧ P(σ, σ0, ind) ∧ σ(ind) = 0
11     S := S_tmp
12     // Encoding of y := 0
13     havoc S_tmp
14     assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S ∧ P(σ, σ0, y) ∧ σ(y) = 0
15     S := S_tmp
16     // Assert the framed hyper-assertion
```

```
17      assert ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(m) = σ2(m)
18      // Encoding of the loop starts
19      // Same pattern as shown by Encoding 5.2
20      ...
21      // Encoding of the loop ends
22      // Assume the framed hyper-assertion
23      assume ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2 ⟹ σ1(m) = σ2(m)
24  }
```

Encoding 5.9: ∀-HHL encoding of Program 5.8

5.2.2 Using Automatic Framing

Explicitly framing all information about program states is cumbersome. Therefore, we have designed a way to automate framing based on the following insight: every state after the loop results from a state before the loop, and they agree on the values of all variables except for those in $mod(C)$ – a set of program variables that are potentially modified by the loop body C. This holds regardless of whether overapproximation or underapproximation reasoning is used. To make use of this insight, we update the loop encodings as shown below.

```
1  assert I(0, S)
2  assume S = S0
3  havoc S
4  havoc n_iter
5  assume n_iter ≥ 0
6  if (*) {
7  // Invariant verification is the same as
       lines 6-17 of Encoding 5.2
8  }
9  havoc S_tmp
10 // Automatic framing
11 assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S∧
       (∀v. v ∉ mod(C) ⟹ σ(v) = σ0(v))
12 // Let S_tmp be a union of S_k's
13 assume ∀σ. σ ∈ S_tmp ⟹ ∃k. k ≥ 0 ∧ σ ∈ S_k ∧ I(k, S_k)
14
15 S := S_tmp
16 // Assume that the loop condition does
       not hold
17 havoc S_tmp
18 assume ∀σ. σ ∈ S_tmp ⟹ σ ∈ S ∧ ¬b(σ)
19 S := S_tmp
```

Encoding 5.10: ∀-HHL encoding of a loop with automatic framing

```
1  assert I(0, S)
2  assume S = S0
3  havoc S
4  havoc n_iter
5  assume n_iter ≥ 0
6  if (*) {
7  // Invariant verification is the same as
       lines 6-17 of Encoding 5.3
8  }
9  havoc S_tmp
10 // Automatic framing
11 assume ∀σ. σ ∈ S_tmp ⟹ ∃σ0. σ0 ∈ S∧
       (∀v. v ∉ mod(C) ⟹ σ(v) = σ0(v))
12 // Let S_tmp be a union of S_k's
13 assume ∀k. k ≥ 0 ⟹ I(k, S_k)
14 assume ∀σ, k. k ≥ 0 ∧ σ ∈ S_k ⟹ σ ∈ S_tmp
15 S := S_tmp
16 // Assume that the loop condition does
       not hold
17 havoc S_tmp
18 assume ∀σ. σ ∈ S ∧ ¬b(σ) ⟹ σ ∈ S_tmp
19 S := S_tmp
```

Encoding 5.11: ∃-HHL encoding of a loop with automatic framing

In fact, automatic framing is strictly stronger than explicit framing. As an example, let us consider Program 4.3 in Section 4.2. Without any kind of

framing, the verifier will lose the information that the value of x in any state coming out of the `if` branch must be strictly greater than the value of x in any state coming out of the `else` branch, so it cannot prove the postcondition. Moreover, this information cannot be explicitly framed. Hence, only by using automatic framing can the verifier prove the postcondition.

5.2.3 Limitations of Automatic Framing

By using the updated loop encodings, information about program states is automatically preserved across loops when the verifier reasons with ∀-HHL, but this is not the case when it reasons with ∃-HHL. Let S, S' be the sets of program states immediately before and after a loop respectively. Assume that the verifier knows $\exists\sigma.\ \sigma \in S \wedge \sigma(x) = 2$ and that the variable x is not modified by the loop body. With automatic framing, if the loop terminates, we should be able to assert that $\exists\sigma'.\ \sigma' \in S' \wedge \sigma'(x) = 2$. Nonetheless, by line 11 of Encoding 5.11, we can learn that every state in S' agrees on the value of x with some state $\sigma_0 \in S$, and because σ_0 is not constrained to be the state $\sigma \in S$ where $\sigma(x) = 2$, we cannot prove the aforementioned assertion. This limitation is fundamental to the current HHL frame rule, which is specialized for reasoning with ∀-HHL. To solve the problem, we must design a frame rule specifialized for reasoning with ∃-HHL and update the encodings accordingly in the future.

5.3 Using Proof Variables

In this section, we present the motivation for introducing proof variables to our language via concrete examples. As we will see in the following subsections, proof variables are especially useful for formulating loop invariants when ∃-HHL is used to reason about loops.

5.3.1 Formulating Invariants of Non-Nested Loops

So far we have seen multiple programs with loops that can be verified with ∀-HHL. Now let us consider Program 5.12 that requires reasoning with ∃-HHL instead. Observe that the loop invariant of the program is an implication whose hypothesis is $n_{iter} \leq 5$. Indeed, the loop in Program 5.12 can execute for at most 5 times in any program state, so there does not exist a program state in which the loop is still running when $n_{iter} > 5$.

```
1  method loop_ex4()
2      returns (y: Int)
3      requires ∃σ. σ ∈ S
4      ensures ∃σ. σ ∈ S ∧ σ(y) = 5
5  {
6      y := 0
7      while (y < 5)
```

```
8           invariant n_iter ≤ 5 ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ 5 ∧ σ(y) = n_iter
9       {
10          y := y + 1
11      }
12  }
```

Program 5.12: A program that requires underapproximation reasoning – verifies

We can easily notice that, in the example above, the number of iterations that the loop can execute is a fixed constant in every program state. However, if we change the loop condition so that the number of iterations that the loop executes depends on the value of some program variable, as shown by Program 5.13 where the loop condition is $y < x$, how do we formulate the loop invariant?

```
1   method loop_ex5(x: Int)
2       returns (y: Int)
3       requires ∃σ. σ ∈ S ∧ σ(x) > 0
4       ensures ∃σ. σ ∈ S ∧ σ(x) = σ(y)
5   {
6       let $p:State ::  $p ∈ S ∧ $p(x) > 0
7       y := 0
8       while (y < x)
9           invariant n_iter ≤ $p(x) ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x)
                ∧σ(y) = n_iter ∧ σ(x) = $p(x)
10      {
11          y := y + 1
12      }
13  }
```

Program 5.13: A program adapted from Program 5.12 by updating the loop condition – should verify in theory

Let us recall Rule 4.3 from the last chapter. This rule allows us to formulate the loop invariant by referring to a state that can reach the loop, such as the state that is guaranteed to exist by the precondition. As a consequence, we can write the loop invariant as $\exists\sigma_0.\ \sigma_0 \in S_0 \wedge \sigma_0(x) > 0 \wedge (n_{iter} \leq \sigma_0(x) \implies \exists\sigma.\ \sigma \in S \wedge \sigma(y) \leq \sigma(x) \wedge \sigma(x) = \sigma_0(x) \wedge \sigma(y) = n_{iter})$, where S_0 represents the set of initial states. However, at the program location where the invariant is expected, S_0 is not accessible to the users. To solve this problem, we allow users to declare proof variables. If we have a proof variable $\$p$ such that it is a state that exists in the set of initial states and also has a positive x value [1], as shown by line 6 of Program 5.13, then we can express the aforementioned loop invariant at the program location where the invariant is expected, as shown by line 9 of the program. The complete encoding of Program 5.13 is shown by Encoding 5.14.

[1] Theoretically, we can simply declare a proof variable that refers to the positive x value in that state, but Viper cannot automatically prove the existence of such an x value in practice.

```
1  method loop_ex5_encoding(x: Int)
2      returns (y: Int)
3      requires ∃σ. σ ∈ S ∧ σ(x) > 0
4      ensures ∃σ. σ ∈ S ∧ σ(x) = σ(y)
5  {
6      var n_iter: Int
7      assert ∃p_0 : State :: p_0 ∈ S ∧ p_0(x) > 0
8      var $p: State
9      assume $p ∈ S ∧ $p(x) > 0
10     havoc S_tmp
11     assume ∀σ. σ ∈ S ⟹ ∃σ_1. σ_1 ∈ S_tmp ∧ P(σ, σ_1, y) ∧ σ_1(y) = 0
12     S := S_tmp
13     assert
           0 ≤ $p(x) ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x) ∧ σ(y) = 0 ∧ σ(x) = $p(x)
14     assume S = S_0
15     havoc S
16     havoc n_iter
17     assume n_iter ≥ 0
18     if (*) {
19         // Assume I(n_iter, S)
20         assume n_iter ≤ $p(x) ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x) ∧
               σ(y) = n_iter ∧ σ(x) = $p(x)
21         // Assume loop condition
22         havoc S_tmp
23         assume ∀σ. σ ∈ S ∧ σ(y) < σ(x) ⟹ σ ∈ S_tmp
24         S := S_tmp
25         // Loop body
26         havoc S_tmp
27         assume
               ∀σ. σ ∈ S ⟹ ∃σ_1. σ_1 ∈ S_tmp ∧ P(σ, σ_1, y) ∧ σ_1(y) = σ(y) + 1
28         S := S_tmp
29         // Assert I(n_iter + 1, S)
30         assert n_iter + 1 ≤ $p(x) ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x) ∧
               σ(y) = n_iter + 1 ∧ σ(x) = $p(x)
31         assume false
32     }
33     havoc S_tmp
34     assume ∀σ. σ ∈ S_tmp ⟹ ∃σ_0. σ_0 ∈ S ∧
           (∀v. v ∉ y ⟹ σ(v) = σ_0(v))
35     assume
           ∀k, σ. k ≥ 0 ⟹ (k ≤ $p(x) ⟹ ∃σ. σ ∈ S_k ∧ σ(y) ≤ σ(x) ∧
           σ(y) = k ∧ σ(x) = $p(x))
36     assume ∀σ, k. k ≥ 0 ∧ σ ∈ S_k ⟹ σ ∈ S_tmp
37     S := S_tmp
38     // Assume negated loop condition
39     havoc S_tmp
40     assume ∀σ. σ ∈ S ∧ σ(y) ≥ σ(x) ⟹ σ ∈ S_tmp
41     S := S_tmp
42 }
```

Encoding 5.14: ∃-HHL encoding of Program 5.13

5.3.2 Formulating Invariants of Nested Loops

Using proof variables as shown in the last subsection does not allow us to formulate the invariant of a loop that is nested inside another loop. Let us consider Program 5.15 below. When formulating the invariant for the nested loop, we need to refer to a state that can reach the loop, but such a state does not necessarily exist. In this case, we would like to formulate the invariant under the assumption that such a state exists. To do so, we first declare another kind of proof variable $\$b$ of type `Bool`. Its value is equal to the value of some expression evaluated in the current set of program states. As shown by line 11 of Program 5.15, $\$b$ is `true` if and only if there exists a state that can reach the nested loop. Afterwards, under the hypothesis that $\$b$ is `true`, we can declare another proof variable $\$q$ to refer to that state and formulate the loop invariant.

```
1  method loop_ex6(x: Int)
2      returns (y: Int)
3      requires ∃σ. σ ∈ S ∧ σ(x) > 0
4      ensures ∃σ. σ ∈ S ∧ σ(x) = σ(y)
5  {
6      let $p:State :: $p ∈ S ∧ $p(x) > 0
7      y := 0
8      while (y < x)
9          invariant
               n_iter <= 1 ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x) ∧ σ(x) = $p(x)∧
               σ(y) = n_iter * σ(x) ∧ $p(x) > 0
10     {
11         let $b: Bool = (∃σ. σ ∈ S ∧ σ(y) ≤ σ(x) ∧ σ(x) = $p(x))
12         let $q:State ::
               $b ⟹ $q ∈ S ∧ $q(y) ≤ $q(x) ∧ $q(x) = $p(x)
13         while (y < x)
14             invariant
                   $b ⟹ (n'_iter ≤ $q(x) − $q(y) ⟹ ∃σ. σ ∈ S ∧ σ(y) ≤ σ(x)∧
                   σ(y) = (n'_iter + $q(y)) ∧ σ(x) = $q(x))
15         {
16             y := y + 1
17         }
18     }
19 }
```

Program 5.15: A program with nested loops – verifies

5.4 Using Hints

Recall that the verifier must be guided by hints when reasoning about `havoc` statements with ∃-HHL. Similar to the ∃-HHL encoding of `havoc` statements, part of the ∃-HHL encoding of loops, i.e. `assume` $\forall k.\ k \geq 0 \implies I(k, S_k)$, also needs to be instantiated for any non-negative integer k, which cannot

be automatically performed by the verifier in practice. Thereby, users must provide hints to guide the verifier.

For example, in order for the verifier to automatically verify Program 5.13, users must update the program by declaring and using hints as shown by Program 5.16 below. In the updated program, the statement that uses the declared hint refers to the proof variable $\$p$, which guides the verifier to instantiate the encoding with $k = \$p(x)$. This makes the encoding shown by Encoding 5.17 explicit to the verifier, thus enabling it to easily prove the postcondition.

```
1  method loop_ex7(x: Int)
2      returns (y: Int)
3      requires ∃σ. σ ∈ S ∧ σ(x) > 0
4      ensures ∃σ. σ ∈ S ∧ σ(x) = σ(y)
5  {
6      let $p: State :: $p ∈ S ∧ $p(x) > 0
7      y := 0
8      while (y < x)
9          {loop_hint} invariant n_iter ≤ $p(x) ⟹ ∃σ. σ ∈ S∧
                                  σ(y) ≤ σ(x) ∧ σ(y) = n_iter ∧ σ(x) = $p(x)
10     {
11         y := y + 1
12     }
13     use loop_hint($p(x))
14 }
```

Program 5.16: A program adapted from Program 5.13 by adding hints – verifies

```
1  assume
       ∀σ. $p(x) ≥ 0 ⟹ ($p(x) ≤ $p(x) ⟹ ∃σ. σ ∈ S_$p(x) ∧ σ(y) ≤ σ(x)∧
       σ(y) = $p(x) ∧ σ(x) = $p(x))
```

Encoding 5.17: ∃-HHL encoding of `use loop_hint(`$p(x)`)`

Chapter 6

Implementation

Based on the approach described in the previous chapters, we have implemented a prototype that can verify programs with ∀-HHL and ∃-HHL. In this chapter, we will present the architecture of the prototype in Section 6.1, and how we model states and sets of states in the implementation in Section 6.2. In Section 6.3, we will describe the assertion language with which users can write hyper-assertions. Finally, we describe in detail how the prototype generates a valid Viper program. If the generated Viper program verifies, it means that the input program is correct.

6.1 Architecture

The prototype consists of 4 components: a parser, a symbol checker, a type checker and a generator. Given any input program written in our language, it is first fed to the parser which performs lexical and syntactic analysis to produce an abstract syntax tree (AST). The AST is then passed to the symbol checker to ensure that all user-defined identifiers are unique in their corresponding scopes. The symbol checker also verifies that all identifiers used have been declared previously. Once the AST passes the symbol checker, it will be type checked based on the predefined type rules. In addition, both the symbol checker and type checker also verify the well-definedness of some language constructs such as proof variable declarations. If neither the symbol checker nor the type checker raises any error, the generator will produce a Viper AST that encodes the input program. Finally, the prototype will call the Silicon backend of Viper to verify the Viper program in the AST form, and then report the verification result returned by Viper.

6.2 Encodings of States and Sets of States

In our implementation, we encode program states and sets of program states using Viper domains. Although Viper has a built-in data structure `Set` that seems to be a reasonable candidate for encoding sets of states, it is finite and therefore cannot encode the infinite set of states resulting from executing a `havoc` statement. For this reason, we have declared a Viper domain to create a new type `SetState` to model sets of states. When it comes to states, the built-in Viper `Map` could be used to encode them, but in this version of the prototype, we have chosen to experiment with the option of using a Viper domain to create a new type `State`. Each of these two domains contains domain functions which can be called to interact with their instances. The complete definition of these domains is presented and explained below.

As Listing 6.1 shows, the `State` domain, which models a program state, is defined with a type parameter `T`. It represents the type of the program variables in a state. At the moment, our prototype only supports reasoning with integers, so `T` is always instantiated with `Int`, but this can be easily generalized in the future. The `State` domain contains two functions. By calling the `get` function with the syntax `get(s, x)`, the verifier can obtain the value of variable x in the state s. By calling the other function with the syntax `everything_equal_except(s1, s2, x)`, the verifier can check whether states s_1 and s_2 agree on the values of all variables except for x. This function essentially encodes the predicate $\mathcal{P}$ introduced in Section 3.3.4. It is useful for encoding `havoc` statements and assignments. The semantics of this function are defined by the axiom, i.e. lines 5-11 of Listing 6.1. Because variables in Viper do not have identifies like objects, the axiom has to compare the values of two `Int` variables to determine whether they are different. As a result, the generator of the prototype must emit Viper assignments to map each declared program variable to a unique integer. For instance, given two program variables x and y, the generator will emit Viper assignments `x := 1` and `y := 2`, which does not affect the values of those variables in a state s, namely `get(s, x)` and `get(s, y)`.

```
1  domain State[T]  {
2    function get(s: State[T], x: T): T
3    function equal_on_everything_except(s1: State[T],
4                          s2: State[T], x: T): Bool
5    axiom equal_on_everything_except_def {
6      (forall s1: State[T], s2: State[T], x: T ::
7        {(equal_on_everything_except(s1, s2, x): Bool)}
8        (equal_on_everything_except(s1, s2, x): Bool) ==>
9        (forall y: T ::x != y ==>
10             (get(s1, y): T) == (get(s2, y): T)))
11   }
12 }
```

Listing 6.1: A Viper domain that models a program state

The SetState domain, as shown by Listing 6.2, models a set of program states. It has two domain functions in_set and set_union. The former function returns a Boolean value indicating whether a state s is in a set of states S, while the latter function returns the union of the two input sets.

```
1  domain SetState[T]  {
2    function in_set(s: State[T], S: SetState[T]): Bool
3    function set_union(S1: SetState[T], S2: SetState[T]): SetState[T]
4    axiom set_union_def {
5      (forall S1: SetState[T], S2: SetState[T] ::
6        { (set_union(S1, S2): SetState[T]) }
7        (forall s: State[T] ::
8          ((in_set(s, S1): Bool) || (in_set(s, S2): Bool)) ==
9              (in_set(s, (set_union(S1, S2): SetState[T])): Bool)))
10   }
11 }
```

Listing 6.2: A Viper domain that models a set of program states

6.3 Assertion Language

To enable users to write hyper-assertions, i.e. expressions that interact with program states, as shown in the programs from the previous chapters, we have designed an assertion language as shown by Figure 6.1.

In the grammar rules, an asterisk represents that its preceding element can have zero or more instances, while a question mark means that its preceding element can have at most 1 instance. As shown by the $\langle id \rangle$ rule of Figure 6.1, identifiers can potentially begin with an underscore or a dollar sign. The purpose of this design is to make it easier to distinguish among different types of variables defined by users during parsing. To be more specific, identifiers of variables local to a hyper-assertion must start with an underscore, identifiers of proof variables must start with a dollar sign, and identifiers of program variables must start with neither.

Moreover, users can express that a state s exists in the current set of program states S by using a state-exists expression <s>. We can notice that the syntax does not involve S, because S is automatically computed at the program location where this expression occurs and does not need to be explicitly specified. Additionally, using state-exists expressions is the only way for users to interact with S, which makes it easier to automatically detect whether $\forall$-HHL or $\exists$-HHL encodings should be generated.

For instance, if users would like to write a hyper-assertion $\forall\sigma.\ \sigma \in S \implies \sigma(x) > 0$ in a program, they can use the assertion language to write it as forall _s:State::<_s> ==> get(_s, x) > 0. In the future, we would like to simplify the syntax of the assertion language, such as simplifying get(_s, x) to _s(x), to allow users to write hyper-assertions more efficiently.

⟨*hyperExp*⟩ ::= ⟨*quantExp*⟩ | ⟨*otherExp*⟩

⟨*quantExp*⟩ ::= (‘`forall`’ | ‘`exists`’) ⟨*varDecl*⟩ ‘`::`’ ⟨*hyperExp*⟩

⟨*varDecl*⟩ ::= ⟨*id*⟩ ‘`:`’ ⟨*type*⟩ ‘`,`’ ⟨*varDecl*⟩*

⟨*otherExp*⟩ ::= ⟨*hyperExp*⟩ ⟨*binaryOp*⟩ ⟨*hyperExp*⟩
| ⟨*unaryOp*⟩ ⟨*hyperExp*⟩
| ⟨*basicExp*⟩

⟨*binaryOp*⟩ ::= ‘`==>`’ | ‘`&&`’ | ‘`||`’ | ‘`>=`’ | ‘`<=`’ | ‘`>`’ | ‘`<`’
| ‘`==`’ | ‘`!=`’ | ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’

⟨*unaryOp*⟩ ::= ‘`-`’ | ‘`!`’

⟨*basicExp*⟩ ::= ⟨*id*⟩ | ⟨*integer*⟩ | ⟨*boolLit*⟩ |
| ‘`<`’ ⟨*id*⟩ ‘`>`’ (State-exists expression)
| ‘`get(`’ ⟨*id*⟩ ‘`,`’ ⟨*id*⟩ ‘`)`’ (Getting var. value in a state)
| ⟨*id*⟩‘`(`’ ⟨*basicExp*⟩ ‘`)`’ (Hint use)
| ‘`(`’ ⟨*hyperExp*⟩ ‘`)`’

⟨*boolLit*⟩ ::= ‘`true`’ | ‘`false`’

⟨*type*⟩ ::= ‘`Int`’ | ‘`Bool`’ | ‘`State`’

⟨*id*⟩ ::= (‘`$`’ | ‘_’)? ⟨*letter*⟩ (⟨*letter*⟩ | digit | ‘_’)*

⟨*integer*⟩ ::= ⟨*digit*⟩ ⟨*digit*⟩*

⟨*letter*⟩ ::= ‘`a`’ | ‘`b`’ | ... | ‘`z`’ | ‘`A`’ | ‘`B`’ | ... | ‘`Z`’

⟨*digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ... | ‘`9`’

Figure 6.1: Grammar rules of the assertion language

6.4 Viper Code Generation

In the previous chapters, we have shown the encodings of all program commands and verification primitives supported by our language. To obtain a valid Viper program from the encodings, it remains for us to translate all hyper-assertions as well as hint declarations and uses to valid Viper code.

6.4.1 Translating Hyper-Assertions

Translation of hyper-assertions provided by users, such as loop invariants, is straightforward, since all elements supported by our assertion language are syntactically valid and semantically equivalent in Viper except for state-exists expressions and hints.

To translate a state-exists expression `<s>`, the prototype emits the code `in_set(s, S)` where `in_set` is the domain function in Listing 6.2 and S is the current set of program states computed by our encodings. As for the translation of hints in a hyper-assertion, we will explain it in detail in the next subsection.

Apart from the hyper-assertions written by users, the encodings contain lots of hyper-assertions which we have designed to encode the semantics of different program commands and primitives. During code generation, the prototype directly emits Viper assertions that are semantically equivalent to those hyper-assertions. This can be easily done for all hyper-assertions with one exception – the hyper-assertion that encodes $S = \bigcup_{k=0,1,\ldots,\infty} S_k$ in the loop encodings, where S is the set of program states after a loop and S_k is the set of program states after k loop iterations.

Let us look back at the ∀-HHL encoding of loops. As shown by line 22 of Encoding 5.2 in Section 5.1, we express that S can be split into S_k's where the invariant I_k holds in S_k by writing `assume` $\forall\sigma.\ \sigma \in S \implies \exists k.\ k \geq 0 \wedge \sigma \in S_k \wedge I(k, S_k)$. Since S_k is local to this hyper-assertion, it is seemingly reasonable to treat it as an assertion variable during Viper code generation, which would result in the following Viper code:

```
assume forall _s: State[Int] :: in_set(_s, S) ==>
    (exists _k: Int, _Sk: SetState[Int] :: _k >= 0 &&
        in_set(_s, _Sk) && I(_k, _Sk))
```

Listing 6.3: Viper code which incorrectly expresses `assume` $\forall\sigma.\ \sigma \in S \implies \exists k.\ k \geq 0 \wedge \sigma \in S_k \wedge I(k, S_k)$

In this way, for two states σ_1 and σ_2 that execute the same number of loop iterations, they can end up in two different sets of states S_{k_1} and S_{k_2} respectively. To see why this is problematic, consider the following program where one of the loop invariants is relational.

```
1  method (x: Int, k: Int, t: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(k) > 0
4      requires ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2
           ⟹ σ1(x) ≤ σ2(x) ∧ σ1(k) = σ2(k)
5      ensures ∀σ1',σ2'. σ1' ∈ S' ∧ σ2' ∈ S' ∧ σ1'(t) = 1 ∧ σ2'(t) = 2
           ⟹ σ1'(y) ≤ σ2'(y)
6  {
7      var x1: Int
8      x1 := x
9      var ind: Int
10     ind := 0
11     y := 0
12     while (ind < k)
13         invariant ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ∧ σ1(t) = 1 ∧ σ2(t) = 2
               ⟹ σ1(x1) ≤ σ2(x1) ∧ σ1(y) = σ2(y)
14         invariant ∀σ. σ(ind) ≤ σ(k) ∧ σ(ind) = n_iter
15     {
16         x1 := x1 * 2 - 1
17         y := x1 - 1
18         ind := ind + 1
19     }
20 }
```

Program 6.4: A program with a relational loop invariant

By analyzing the program, we can easily learn that the loop runs for exactly k iterations in any program state. Let σ_1 and σ_2 be two initial states. After the loop, they must be in $S_{\sigma_1(k)}$ and $S_{\sigma_2(k)}$ respectively. Since $\sigma_1(k) = \sigma_2(k)$, they must end up in the same set of states $S_{\sigma_1(k)}$ where $I(\sigma_1(k))$ holds. By using the relational loop invariant on line 13 of the program, the verifier can easily prove the postcondition. However, if we were to follow the Viper code in Listing 6.3, then σ_1 and σ_2 could respectively end up in S_{k_1} and S_{k_2} such that $S_{k_1} \neq S_{k_2}$ after the loop, which prevents the prototype from proving the postcondition.

It is obvious that the solution is to guarantee that every unique k must be consistently mapped to the same unique S_k. To achieve this, we make use of Viper functions which always return the same result for the same input value. More precisely, we declare a Viper function `function get_Sk(k:Int):SetState[Int]` [1] and use its return value to represent S_k. This results in the following updated Viper code:

```
assume forall _s: State[Int] :: in_set(_s, S) ==>
    (exists _k: Int :: _k >= 0 &&
        in_set(_s, get_Sk(_k)) && I(_k, get_Sk(_k)))
```

Listing 6.5: Viper code which correctly expresses `assume` $\forall\sigma.\ \sigma \in S \implies \exists k.\ k \geq 0 \wedge \sigma \in S_k \wedge I(k, S_k)$

6.4.2 Translating Hint Declarations and Uses

Recall that, in Section 4.3, we have mentioned that hints will be encoded as triggers, i.e. syntactic cues that guide quantifier instantiations. In the Viper verification language, users can declare triggers as part of a quantified assertion so that the assertion is only instantiated when the trigger expressions are matched [13]. Since calls to domain functions and top-level functions can be used as triggers in Viper, upon detection of a hint declared with `<hint_id>` as its identifier, we declare two Viper functions, identified as `<hint_id>` and `<hint_id>_wrapper` as shown by Listing 6.6. During implementation, calls to the first function will be used as trigger expressions, while calls to the second function will be used to trigger the quantified assertions containing the corresponding trigger expressions.

```
function <hint_id> (k: Int): Bool = {
    k > 0 || k <= 0
}
function <hint_id>_wrapper(k: Int): Bool = {
    <hint_id>(k)
}
```

Listing 6.6: Viper functions generated when a hint identified as `<hint_id>` is declared

[1] In actual implementation, a function like this is generated with a unique identifier for each loop detected in the input program.

Notice that the first function always returns `true`, and the second function is simply a wrapper of the first function. The purpose of this design is to handle all possible use cases of hints, especially where users use hints in a hyper-assertion, as shown by the postcondition of Program 4.13 in Section 4.3. Since the hint is written as a conjunct in the hyper-assertion, we must ensure that it simply provides guidance and does not affect the validity of the Viper assertion which the hyper-assertion is translated to. To achieve this, we fix the return value of the function `<hint_id>` to be `true` during implementation. However, a conjunct that is constantly `true` can be optimized away, which counteracts the purpose of using hints. Therefore, instead of letting the function return a Boolean literal, we use an expression that trivially evaluates to `true` as its function body. Additionally, due to limitations of the Silicon backend of Viper, when translating a hint used in a hyper-assertion, we must call a function that wraps around the actual function implementing the hint, hence the generation of the wrapper function. Only in this way can the corresponding quantified assertion be triggered as expected.

As an example, let us look at the following Viper code which shows how we translate the hint declaration and use in Program 4.13. Since the hint of the program is declared as part of the `havoc x` statement, the quantified assertion in the encoding of that statement, i.e. lines 25-30, contains a trigger `hint1(k)` where `hint1` is the function implementing the declared hint. As for the hint use, it is translated to `hint1_wrapper(_n)`, i.e. a function call to the wrapper function, and kept as a conjunct of the postcondition, which successfully triggers lines 25-30 and allows the prototype to verify the postcondition. As a side note, when hints are part of a program postcondition, we also translate the postcondition to an `inhale-exhale` assertion, as shown by lines 10-20. By doing so, the hints are only inhaled when the prototype asserts the postcondition, which prevents the hints from affecting modular reasoning of method calls that we will support in the future.

```
1  function hint1 (k: Int): Bool = {
2      k > 0 || k <= 0
3  }
4  function hint1_wrapper(k: Int): Bool = {
5      hint(k)
6  }
7
8  method randomNumGen() returns (x: Int)
9    requires ...
10   ensures [
11     // The postcondition that is inhaled
12     (forall _n: Int ::_n < 9 && _n > 0 ==>
13       true &&
14       (exists _s: State[Int] :: (in_set(_s, S): Bool) &&
15         (get(_s, x): Int) == _n)),
16     // The postcondition that is exhaled (containing hints)
17     (forall _n: Int ::_n < 9 && _n > 0 ==>
```

```
18        hint1_wrapper(_n) &&
19        (exists _s: State[Int] :: (in_set(_s, S): Bool) &&
20          (get(_s, x): Int) == _n))]
21  {
22    ...
23    // Viper encoding of havoc x {hint}
24    S_temp := havocSet()
25    assume (forall s: State[Int], k: Int ::
26        { hint1(k), (in_set(s, S): Bool) }
27        (in_set(s, S): Bool) ==>
28        (exists s1: State[Int] :: (in_set(s1, S_temp): Bool) &&
29          (equal_on_everything_except(s, s1, x): Bool) &&
30          (get(s1, x): Int) == k))
31    S := S_temp
32    ...
33  }
```

Listing 6.7: Viper translation of Program 4.13

Chapter 7

Evaluation

This chapter presents the results of evaluating our prototype that authomates ∀-HHL and ∃-HHL against existing state-of-the-art verifiers including Descartes [4], HyPa [14], PCSat [15], and ORHLE [8]. By conducting the evaluation, we aim at answering the following research questions:

- **RQ1**: Can our prototype successfully prove or disprove meaningful program properties as the state-of-the-art verifiers?
- **RQ2**: Can our prototype prove or disprove meaningful program properties within a reasonable time limit?
- **RQ3**: How does the amount of user annotations required by our prototype compare to that required by state-of-the-art verifiers?

In the following sections, we will present our evaluation methodology, and then show the detailed evaluation result. Lastly, we answer the research questions posed above.

7.1 General Methodology

To evaluate our prototype, we use a benchmark suite composed of publicly accessible benchmarks of the existing verifiers under comparison. For each program in the benchmark suite, we check whether our prototype produces the expected output and meanwhile measure its runtime $t(s)$. The runtime measurement is collected by executing the prototype on a MacBook Pro running macOS Ventura 13.3 with a 2.3 GHz 8-Core Intel Core i9 processor and 32 GB RAM.

We also quantitatively analyze the input taken by our prototype and other verifiers for the same benchmark. More specifically, we categorize the code in each benchmark into 3 types: program commands, program specifications and proofs needed to complete verification. Sometimes, it is impossible

to unambiguously distinguish between specifications and proofs. In this case, we combine them into one category called user annotations. For any benchmark encoded with our language, we regard the method preconditions and postconditions as program specifications, all loop invariants and any code that uses the aforementioned verification primitives as proofs, and the rest of the code as program commands.

7.2 Evaluation of ∀-HHL Implementation

To evaluate our ∀-HHL implementation, we have compiled a benchmark suite composed of 19 programs that can be verified by performing only over-approximation reasoning. These programs are taken from the benchmarks of Descartes, HyPa and PCSat and then manually translated into the syntax accepted by our prototype.

7.2.1 Evaluation on Descartes Benchmarks

Descartes is a verifier developed based on Cartesian Hoare Logic and can perform overapproximation reasoning to prove hyperproperties. In theory, our prototype can encode and verify all benchmarks from Descartes, but only a subset of them have been selected for our evaluation. This is because Descartes's benchmarks are Java programs containing objects. When the objects have plenty of fields, manually translating the Java programs into our syntax is extremely cumbersome. Additionally, some of Descartes's benchmarks use data structures such as arrays that our prototype does not support at the moment. As a result, only those benchmarks that do not involve arrays or objects of more than 3 fields are used in our evaluation.

P1: $\forall x, y, z.\ (eq(x,y) = 1 \land eq(y,z) = 1) \implies eq(x,z) = 1$
P2: $\forall x, y.\ eq(x,y) = 1 \iff eq(y,x) = 1$
P3: $\forall x, y.\ eq(x,y) = eq(x,y)$
P4: $\forall x, y.\ sign(cmp(x,y)) = -sign(cmp(y,x))$
P5: $\forall x, y, z.\ (cmp(x,y) > 0 \land cmp(y,z) > 0) \implies cmp(x,z) > 0$
P6: $\forall x, y, z.\ cmp(x,y) = 0 \implies (sign(cmp(x,z)) = sign(cmp(y,z)))$

Figure 7.1: Hyperproperties that a correctly implemented comparator should satisfy. An equality comparator should satisfy properties P1-P3, where $eq(x,y) = 1$ when x is equal to y. A comparator that checks for more than equality should satisfy properties P4-P6, where $cmp(x,y)$ is a positive (resp. negative) integer when x is strictly greater (resp. less) than y. $cmp(x,y)$ is 0 when x and y are equal. $sign(cmp(x,y))$ represents the sign of $cmp(x,y)$.

All selected benchmarks from Descartes implement the Java `Comparator` interface and return the result of comparing two inputs x and y. For a comparator that only checks whether x is equal to y, we verify that it satisfies properties P1-P3 in Figure 7.1; while for a comparator that checks whether x

is greater than, equal to, or less than y, we verify that it satisfies properties P4-P6 in Figure 7.1.

Benchmark	Descartes			Our Prototype				
	$\#_C$	$\#_S$	$\#_P$	$\#_C$	$\#_S$	$\#_P$	$t(s)$	Res.
DefaultCookie	34	44	0	41	15	0	4.47	✓
Contact_False	62	60	0	114	20	0	125.66	✓
Match_False	15	60	0	16	20	0	6.76	✓
Match	13	60	0	20	20	0	5.79	✓
Datapoint_False	50	60	0	50	20	0	17.83	✓
NameComparator_False	20	60	0	36	21	0	1.23	✓
NameComparator	21	60	0	40	21	0	1.27	✓
FileItem_False	14	60	0	9	19	0	2.87	✓
FileItem	28	60	0	29	19	0	3.28	✓
Time_False	19	60	0	16	18	0	1.80	✓
Time	15	60	0	23	18	0	2.06	✓
Node_False	18	60	0	16	25	0	2.65	✓
Node	18	60	0	16	25	0	2.00	✓
Solution_False	46	60	0	58	22	0	Times out	×
Solution	46	60	0	58	21	0	Times out	×

Table 7.1: Evaluation on Descartes benchmarks. $\#_C$ is the number of lines of program commands, $\#_S$ is the number of lines of specifications, $\#_P$ is the number of lines of proofs. Note that we ignore all empty lines. For Descartes, we regard all code in the input Java programs as commands and the encodings of properties P1-P6 as specifications. DefaultCookie is the only equality comparator for which properties P1-P3 are verified. All the other comparators check for more than equality. To verify their correctness, Descartes reuses the same encodings of properties P4-P6, hence the identical value of $\#_S$ for those benchmarks. In addition, all benchmarks whose names ending with False are buggy and should not be verified, while all the other benchmarks are non-buggy and should be verified. The value of the last column, i.e. Res., indicates whether our prototype gives the expected verification result or not.

Table 7.1 shows the results of running our prototype on the selected benchmarks from Descartes. With the exceptions of the last two benchmarks where our prototype times out during verification, it is able to verify that each non-buggy benchmark satisfies the desired hyperproperties and report that each buggy benchmark fails to verify. In the case of verification failure, our prototype can also pinpoint the violated properties. Even though it cannot provide a counterexample like Descartes, it can formally prove that the buggy benchmarks are indeed problematic by switching to using underapproximation reasoning as shown by Section 7.3.2.

7.2.2 Evaluation on HyPa and PCSat Benchmarks

HyPa and PCSat are two verifiers that can prove k-safety properties and hyperproperties beyond k-safety. Since our goal is to evaluate our ∀-HHL implementation, we have taken only those benchmarks that are array-free and verify k-safety properties for multiple executions of the same program from the benchmark suites of HyPa and PCSat.

The results of running our prototype on the selected benchmarks from HyPa and PCSat can be found in Table 7.2. Each of these benchmarks is associated with a 2-safety property, which can be successfully verified by our prototype.

Benchmark	HyPa			PCSat			Our prototype				
	$\#_C$	$\#_S$	$\#_P$	$\#_C$	$\#_S$	$\#_P$	$\#_C$	$\#_S$	$\#_P$	$t(s)$	Res.
HalfSquareNI	28	25	47	12	2	45	20	2	4	4.22	✓
SquaresSum	23	25	21	6	2	28	12	4	1	1.15	✓
DoubleSquareNI	25	26	21	-	-	-	21	3	3	1.17	✓
DoubleSquareNIff	23	25	15	11	2	31	21	3	3	1.02	✓

Table 7.2: Evaluation on HyPa and PCSat benchmarks. Some entries in the table are left empty because the corresponding benchmark does not exist in the public benchmark suite of the corresponding verifier. $\#_C$ is the number of lines of program commands, $\#_S$ is the number of lines of specifications, $\#_P$ is the number of lines of proofs. Note that we ignore all empty lines. For HyPa, we regard all code in .hypa files and safety automata as specifications and predicates as proofs. For PCSat, we consider all inputs as proofs, and refer to the comments in each input file to determine the amount of program commands and specifications in the corresponding benchmark. All benchmarks are expected to be verified. The value of the last column, i.e. Res., indicates whether our prototype gives the expected verification result or not.

7.3 Evaluation of ∃-HHL Implementation

To evaluate our ∃-HHL implementation, we have composed a benchmark suite of 14 programs that can be verified by performing only underapproximation reasoning. These programs are taken from the benchmark suites of ORHLE and Descartes and then manually translated into the syntax accepted by our prototype.

7.3.1 Evaluation on ORHLE Benchmarks

ORHLE is a verifier developed based on RHLE. It can prove and disprove ∀∃-hyperproperties. Since our goal is to evaluate our ∃-HHL implementation, we have taken only those benchmarks that are array-free and disprove k-safety properties from the benchmark suite of ORHLE.

Table 7.3 summarizes the results of running our prototype on the selected ORHLE benchmarks. For each benchmark, our prototype gives verification results consistent as ORHLE.

Benchmark	ORHLE		Our Prototype			
	$\#_C$	$\#_A$	$\#_C$	$\#_A$	$t(s)$	Res.
BlackJack_DoNothing_False	3	12	4	2	1.04	✓
BlackJack_Once_False	5	12	7	12	0.91	✓
BlackJack_Until21	6	18	11	8	1.25	✓
Sandbox	15	19	26	13	1.42	✓
HTTPRequest	19	19	22	8	4.27	✓
SleepAndContinue	26	33	45	19	3.34	✓

Table 7.3: Evaluation on ORHLE benchmarks. $\#_C$ is the number of lines of program commands, and $\#_A$ is the number of user annotations. Note that we ignore all empty lines. For ORHLE, we consider all functions with concrete implementation as commands and the rest of the code in the input files as user annotations. We do not further categorize annotations into specifications and proofs because the boundary between them is ambiguous. In addition, all benchmarks whose names ending with False are buggy and should not be verified, while all the other benchmarks are non-buggy and should be verified. The value of the last column, i.e. Res., indicates whether our prototype gives the expected verification result or not.

7.3.2 Evaluation on Buggy Descartes Benchmarks

Benchmark	Our Prototype				
	$\#_C$	$\#_S$	$\#_P$	$t(s)$	Res.
Solution_False	58	9	0	6.23	✓
Contact_False	114	18	0	32.43	✓
Node_False	16	9	0	2.56	✓
Time_False	16	6	0	0.82	✓
FileItem_False	9	9	0	2.81	✓
Datapoint_False	50	28	0	24.31	✓
NameComparator_False	36	7	0	0.88	✓
Match_False	16	17	0	14.77	✓

Table 7.4: Evaluation on buggy Descartes benchmarks. $\#_C$ is the number of lines of program commands, $\#_S$ is the number of lines of specifications, $\#_P$ is the number of lines of proofs. Note that we ignore all empty lines. All benchmarks are buggy programs which violate at least one of the properties shown by Figure 7.1. Our prototype is expected to prove such violation by showing the existence of erroneous execution traces. The value of the last column, i.e. Res., indicates whether our prototype gives the expected verification result or not.

The results of running our prototype on the buggy benchmarks from Descartes can be found in Table 7.4. Unlike the evaluation from Section 7.2.1, our proto-

type now performs underapproximation reasoning to prove the existence of multiple bad executions, thus proving that the benchmarks are indeed buggy. Since Descartes cannot perform underapproximation reasoning, only the data of running our prototype on these benchmarks is available. As shown Table 7.4, our prototype can prove that each of these benchmarks is buggy as expected.

7.4 Discussion

After running our prototype on the benchmarks taken from multiple state-of-the-art verifiers, we may conclude that it can successfully prove and disprove meaningful program properties, including both trace properties and hyperproperties. Although our prototype does not always terminate, it never produces an unsound result.

Moreover, as the data shows, our prototype is usually quite efficient, except when the programs under verification contain lots of nested conditional statements, such as Contact_False and Solution in Table 7.1.

Furthermore, by quantitatively analyzing the inputs taken by all verifiers under comparison, we can see that our prototype does not seem to require more user annotations than other verifiers to complete the verification. However, this is not a definite conclusion, because the amount of user annotations in an input file cannot be precisely measured by counting the number of lines of code like we did, especially when the syntax of the input languages accepted by different verifiers has very few similarities. When compared to Descartes, the syntax of our language is much more complicated and every line of our annotation is much longer. More importantly, Descartes does not require proofs provided by users during verification and can reuse specifications to verify the same properties for different programs, so Descartes generally needs less user annotations than our prototype. Similar to Descartes, Hypa also supports an input language with less complicated syntax. Nevertheless, it requires specifications and proofs that are difficult for users not familiar with the verifier to write, while our prototype allows users to easily write annotations as predicates with universal and/or existential quantifiers, so average users will find our prototype easier to use. When it comes to ORHLE, it supports an input language similar to ours, but its syntax is more concise. Notwithstanding, it sometimes needs to both existentially and universally quantify over program states even when it performs only over- or underapproximation reasoning. Consequently, in those cases, our prototype requires less user annotations than ORHLE. Lastly, PCSat does not take programs with commands but rather programs reduced to a certain form as its input. The input files accepted by PCSat are often very lengthy and also difficult for average users to read and write, so it requires more user annotations and is

also more difficult to use than our prototype in general.

Chapter 8

Conclusion and Future Work

In this chapter, we first briefly summarize and conclude our work, and then elaborate on its limitations and how to address them.

8.1 Conclusion

In this thesis, we have proposed an approach to build an automatic verifier that reasons with $\forall$-HHL and $\exists$-HHL to prove and disprove program hyperproperties. We have defined a language which includes a set of commonly used program commands and verification primitives that allow users to provide verification guidance. Given any input program written in this language, we have designed Viper encodings to track a set of program states from the beginning to the end of the program based on the semantics of the commands and primitives.

Moreover, a prototype has been developed to implement the proposed approach. The prototype translates any input program written in the aforementioned language to a Viper program with the designed encodings, and then delegates the verification of the Viper program to the Silicon back-end of Viper. After evaluating the prototype with benchmarks of existing state-of-the-art verifiers, we have learnt that our prototype can successfully prove and disprove trace properties and hyperproperties within a reasonable time limit most of the time.

8.2 Future Work

Our work is not without limitations. In the following subsections, we discuss the downsides of our approach and implementation, and also propose solutions that can potentially improve our work in the future.

8.2.1 Beyond Over- and Underapproximation

This thesis focuses on the part of HHL that performs only overapproximation or underapproximation reasoning, but, in fact, HHL is theoretically capable of verifying hyperproperties that involve both universal and existential quantifications. Recall that, in Section 3.2, we have explained that Formulas 3.1 and 3.2 combined allow us to compute the true set of program states, $sem(C, S)$, after some command C is executed in the initial set of states S. We have also shown that, when reasoning with only $\forall$-HHL (resp. $\exists$-HHL), it suffices to compute an approximation of $sem(C, S)$ by using only Formula 3.1 (resp. Formula 3.2). As a result, it is obvious that by combining $\forall$-HHL and $\exists$-HHL encodings of any command C, we are able to compute the exact value of $sem(C, S)$ and prove $\forall\exists$- and $\exists\forall$-hyperproperties.

We have done an experiment where we let the prototype emit both $\forall$-HHL and $\exists$-HHL encodings of the commands in a program that is known to satisfy GNI, i.e. a $\forall\exists$-hyperproperty. With the current encoding design, the Silicon back-end of Viper does not terminate when verifying the generated Viper program. Non-termination like this can also be occasionally observed when only overapproximation (resp. underapproximation) reasoning is used to verify universal (resp. existential) hyperproperties as well. One example program that can trigger this behavior is the Solution test case in Section 7.2.1. By manually analyzing the generated Viper encodings and also using the profiling feature of Silicon, we have discovered that this is caused by an issue known as matching loops, namely infinite instantiations of quantifiers.

To solve the matching loop problem, we propose the following solutions. First of all, we can use a tool specifically designed for debugging quantifier instantiations in SMT solvers, such as the Axiom Profiler [16], which might provide us with more guidance in detecting and removing the cause of matching loops in our encodings. Secondly, by using the information given by the tool, and by using manual inspection, we can rewrite the current encodings into semantically equivalent forms that circumvent matching loops. Alternatively, we can consider different encoding designs. For instance, we may opt for Viper's built-in `Map` data structure to encode program states because it has been well tested.

8.2.2 Better Support for Loops

Although the current encodings allow us to reason about loops to a great extent, there exist possible improvements that can make them more robust.

Recall that, in Section 5.2, we have shown that information about program states can be framed around a loop by using explicit or automatic framing. Nevertheless, this does not imply that information about program states is

preserved in every loop iteration. For instance, consider Program 8.1 shown below.

```
1  method loop_support1(x: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) = 2
4      ensures ∀σ. σ ∈ S ⟹ σ(x) = 2
5  {
6      y := 0
7      while (y < 5)
8          invariant ∀σ. σ ∈ S ⟹ σ(y) ≤ 5
9      {
10         hyperAssert ∀σ. σ ∈ S ⟹ σ(x) = 2
11         y := y + 1
12     }
13 }
```

Program 8.1: A program with a loop and trivial specifications – does not verify

The verifier fails on line 10 of the program. This is because the information that x is not modified by the loop body is not available to the verifier when it checks the invariant preservation in a separate proof obligation. To solve this problem, users may add the program properties established by the code prior to the loop as a loop invariant so that they are preserved in every loop iteration. We can easily automate this in the same fashion as we automate framing around the loop.

Apart from framing information around each loop iteration, we can also have a more specialized encoding for loops that run for the same number of iterations in all program executions, as is done in the paper authored by Eilers et al. [17]. This makes it much easier to use relational loop invariants to prove hyperproperties. For example, the relational invariant in Program 8.2 does not allow the verifier to directly prove the program postcondition, because the current loop encodings only express that the invariant holds in every S_k, where S_k is the set of program states after $k \geq 0$ loop iterations. This certainly does not imply that the invariant also holds in S' which is the union of all S_k's. If we could prove that, for any two initial states σ_1, σ_2 such that $\sigma_1(x) = \sigma_2(x)$, the loop executes the same number of iterations, say k_0, in them, then we can prove that $S' = S_{k_0}$, which directly implies the postcondition.

```
1  method loop_support2(x: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) > 0
4      requires ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ⟹ σ1(x) = σ2(x)
5      ensures ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ⟹ σ1(y) = σ2(y)
6  {
7      y := 0
8      while (y < x)
9          invariant ∀σ1, σ2. σ1 ∈ S ∧ σ2 ∈ S ⟹ σ1(y) = σ2(y)
10     {
```

```
11            y := y + 1
12        }
13    }
```

Program 8.2: A program with a loop and a relational loop invariant – does not verify

Last but not least, as mentioned in Section 5.2.3, we do not have a rule specialized for framing information about program states when reasoning with ∃-HHL. Such a rule should be added and the ∃-HHL loop encoding should be updated accordingly in the future.

8.2.3 Support for Method Calls

At the moment, even though our verifier expects any input program to be composed of methods with specifications, it cannot reason about method calls. Supporting modular reasoning about method calls is non-trivial due to the method specifications being hyper-assertions. Let us look at Program 8.3 as an example. Immediately before line 6, we know that the value of x is either 5 or 7 in any program state in the method `foo`. If we simply check whether the precondition of the callee, i.e. method `bar`, holds at this point, we will conclude that the method call `bar(x)` should fail. However, we could decompose the set of states in `foo` into two sets such that one of them includes all states where $x = 5$ and the other includes all states where $x = 7$. Since each of them satisfies the precondition of `bar`, the method call should be allowed to execute in each set respectively. Since we can infer that the method `bar` is deterministic from its precondition and postcondition, we should be able to prove the postcondition of `foo`, which essentially says that the final set of states has at most two values for y. For the time being, it is unclear how we can handle scenarios like this automatically.

```
1  method foo(x: Int)
2      returns (y: Int)
3      requires ∀σ. σ ∈ S ⟹ σ(x) = 5 ∨ σ(x) = 7
4      ensures ∀σ1,σ2,σ3. σ1 ∈ S ∧ σ2 ∈ S ∧ σ3 ∈ S ⟹
           σ1(y) = σ2(y) ∨ σ2(y) = σ3(y) ∨ σ3(y) = σ1(1)
5  {
6      y := bar(x)
7  }
8
9  // This is a deterministic method
10 method bar(x: Int)
11     returns (y: Int)
12     requires ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ⟹ σ1(x) = σ2(x)
13     ensures ∀σ1,σ2. σ1 ∈ S ∧ σ2 ∈ S ⟹ σ1(y) = σ2(y)
14 {
15     // Method body
16 }
```

Program 8.3: A program with a method call that we cannot handle yet

8.2.4 Support for Comprehensions

There are hyperproperties that require comprehensions, such as counting and summation, rather than quantification over the program states. As an example, one hyperproperty that requires counting is the existence of n different outputs for any given input. HHL is able to prove such properties theoretically, but it remains for us to explore how to do so automatically in practice.

Bibliography

[1] Thibault Dardinier and Peter Müller. Hyper hoare logic: (dis-)proving program hyperproperties (extended version). *ArXiv*, abs/2301.10037, 2023. doi: 10.48550/arXiv.2301.10037.

[2] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

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

[4] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. *SIGPLAN Not.*, 51(6):57–69, jun 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908092. URL https://doi.org/10.1145/2980983.2908092.

[5] Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4 (POPL), dec 2019. doi: 10.1145/3371078. URL https://doi.org/10.1145/3371078.

[6] Edsko de Vries and Vasileios Koutavas. Reverse hoare logic. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, pages 155–171, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24690-6. doi: 10.1007/978-3-642-24690-6_12.

[7] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *ArXiv*, abs/2303.03111, 2023. doi: 10.48550/arXiv.2303.03111.

[8] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. Rhle: Modular deductive verification of relational ∀∃ properties. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 67–87, Cham, 2022. Springer Nature Switzerland. ISBN 978-3-031-21037-2.

[9] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49122-5. doi: 10.1007/978-3-662-49122-5_2.

[10] Malte H. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.

[11] K. Rustan M. Leino. This is boogie 2. June 2008. URL https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[13] Group of Programming Methodology of ETH Zürich. Viper tutorial, 2022. URL https://viper.ethz.ch/tutorial/.

[14] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In *Computer Aided Verification*, pages 341–362. Springer International Publishing, 2022. doi: 10.1007/978-3-031-13185-1_17. URL https://doi.org/10.1007%2F978-3-031-13185-1_17.

[15] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 742–766, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81685-8.

[16] Nils Becker, Peter Müller, and Alexander J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing. doi: 10.1007/978-3-030-17462-0_6.

[17] Marco Eilers, Thibault Dardinier, and Peter Müller. Commcsl: Proving information flow security for concurrent programs using abstract commutativity, 2023.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

AN AUTOMATIC PROGRAM VERIFIER FOR HYPERPROPERTIES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
LI	AWQI

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich, Oct 18 2023	李安琪

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.