Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automated Compositional Verification of Hyperproperties

Bachelor's Thesis

Patrick Neugebauer

Advisors: Thibault Dardinier, Anqi Li, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

Hyperproperties are program properties that relate multiple executions of a program (e.g., determinism). To address the challenge of verifying these properties, Hyper Hoare Logic (HHL) has been developed. Hypra is an automated verifier that automates HHL. It works by translating Hypra programs into Viper programs that model the intended behavior.

However, a limitation of Hypra is its inability to use a method's hyperproperty specification (e.g., determinism) to verify a different property that arises from the interaction of multiple calls to this method. This thesis addresses this limitation by introducing a domain-specific language (DSL) that can be used to apply method specifications in a non-trivial manner. This approach enables the verification of complex properties that arise from method composition across programs.

Further, the DSL is extended to support the application of method specifications across loop iterations, addressing and solving unique challenges introduced by loops. Finally, the expressiveness of this DSL is demonstrated through multiple examples.

Acknowledgements

First, I would like to thank my supervisors, Thibault Dardinier and Anqi Li, for their constant feedback throughout the thesis. Their inputs were invaluable as this was my first written thesis on this scale. The proposed project was also fascinating.

Additionally, I would also like to thank my family for their support.

Contents

Chapter 1

Introduction and Foundation

Automated program verifiers are already well established and have demonstrated their effectiveness in a wide variety of applications. Nevertheless, there exists a class of program properties (hyperproperties [1]) that cannot be verified using "conventional" verifiers. Hyperproperties are properties that can only be proven by analyzing multiple individual program traces in relation to one another.

To prove such properties, Hyper Hoare Logic (HHL) [2] has been developed. It extends Hoare Logic [3, 4] by introducing the concept of hyperproperties. HHL introduces hyper-triples of the form $\{P\}\ S\ \{Q\}$, where P and Q denote the pre- and postconditions of the program S, respectively. This approach is analogous to Hoare-triples in Hoare Logic. However, the key difference is that with hyper-triples, we can reason over multiple program states simultaneously, rather than being limited to just a single state. HHL achieves this by introducing the ability to quantify over program states, as P and Q are hyper-assertions reasoning over **sets of states**, unlike program-assertions reasoning over **states**. For example, the hyper-triple for determinism is:

$$\{\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\sigma_1(x) = \sigma_2(x)\}$$
$$\text{y := x + 1}$$
$$\{\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\sigma_1(y) = \sigma_2(y)\}$$
which is a special notation for:
$$\{\lambda S_{pre}.\forall\sigma_1, \sigma_2 \in S_{pre}.\sigma_1(x) = \sigma_2(x)\}$$
$$\text{y := x + 1}$$
$$\{\lambda S_{post}.\forall\sigma_1, \sigma_2 \in S_{post}.\sigma_1(y) = \sigma_2(y)\}$$

Figure 1.1: In this figure S_{pre}/S_{post} represents a set of program states before and after the statement y := x + 1, while σ_1, σ_2 are individual program states and $\sigma_1(x)$ is the function returning the value x in the program state σ_1.

To prove such hyper-triples, a program verifier called Hypra [5] has been developed, which enables automatic reasoning about hyperproperties.

1.1 Hypra

Hypra [5] works by encoding the semantics of an input program with its specifications into Viper [6].

As shown in Figure 1.1, verifying a hyper-triple over a program essentially means verifying that the postcondition of the hyper-triple holds on the set of program states S_{post}. S_{post} denotes the set of program states that can be reached by executing the program on each state in S_{pre}, and S_{pre} is the set of states that satisfies the precondition of the hyper-triple.

Hypra works by translating each method into a separate Viper method with the set of input states S_{pre} as a method parameter and the set of output states S_{post} as a return value. The generated Viper program maintains a set of states, S. Each statement from the Hypra program is translated into an assumption that updates this set. An example of this behavior can be found in Figure 1.2.

method det(x : **Int**)
returns (y : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}

method det
 (x : **Int**, y : **Int**, S_{pre} : **SetState**)
returns (S_{post} : **SetState**)
 requires $\forall\sigma_1, \sigma_2.\sigma_1 \in S_{pre} \land \sigma_2 \in S_{pre}$
  $\Rightarrow \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\sigma_1, \sigma_2.\sigma_1 \in S_{post} \land \sigma_2 \in S_{post}$
  $\Rightarrow \sigma_1(y) = \sigma_2(y)$
{
 var S : **SetState**
 $S := S_{pre}$
 $S :=$ the encoding of $y := x + 1$
 $S_{post} := S$
}

Figure 1.2: The Viper (right) translation of a deterministic Hypra (left) method. The pre- and postconditions of the Viper method have been translated to use the set of states S_{pre} and S_{post}.

1.2 Limitations

In Figure 1.3, `det` is a small deterministic method for which Hypra can be used to prove the determinism property. However, what happens if we write a program that uses a deterministic method twice on the same input? From the determinism property, we would expect that the output of both function

method det(x : **Int**)
returns (y : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}

method main(u : **Int**)
returns (v : **Int**, w : **Int**)
 ensures $\forall\langle\sigma\rangle.\, \sigma(v) = \sigma(w)$
{
 $v := \text{det}(u)$
 $w := \text{det}(u)$
}

Figure 1.3: In this example we want to show that the deterministic method `det` applied twice to the same input results in the same output.

calls will be the same. Nevertheless, Hypra cannot prove this property by writing the program specified in Figure 1.3.

To understand why this is the case, we simplify the example so that the `main` method has a stronger precondition, which states that all the input variables across all states have the same value. By doing this, we obtain the program in Figure 1.4.

method det(x : **Int**)
returns (y : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}
method main(a : **Int**)
returns (b : **Int**, c : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a)$
 ensures $\forall\langle\sigma\rangle.\, \sigma(b) = \sigma(c)$
{
 // $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a)$
 $b := f(a)$
 // $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a) \land \sigma_1(b) = \sigma_2(b)$
 $c := f(a)$
 // $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a) \land \sigma_1(b) = \sigma_2(b) \land \sigma_1(c) = \sigma_2(c)$
}

Figure 1.4: This figure shows a program that applied the method f twice to the input argument a. The comments highlight the properties that Hypra automatically infers about the current set of program states at each respective point.

At each method call, Hypra asserts the method's precondition on the current set of states S and then updates this set by assuming the postcondition on S, thereby obtaining the properties outlined in the comments of Figure 1.4.

However, from the properties that Hypra inferred about the states at the end of the program, we cannot prove the postcondition of `main`, since we never applied the specification to both method calls at the same time. Therefore, this way of handling method calls is insufficient to prove all properties arising from the composition of multiple method calls.

1.3 Goals

This leads us to the goals of this thesis. As shown before, our main goal is to prove complex properties that arise from the composition of specifications across multiple method calls. To achieve this, we also added the goals of named specification, enabling us to specify which specification to apply on those method calls. Additionally, we aimed to incorporate logical variables, as they are a valuable tool for hyperproperties and applying them. The end goal was that to apply what we learned about method specification composition and extend those concepts to loops.

Chapter 2

A Domain-Specific Language for Compositional Verification

In the previous chapter, we saw that Hypra works by tracking a set of states that can reach each point in a program. With Hypra, we can already prove specific properties that arise from the sequential application of a method specification, for example, that a deterministic method applied twice is still deterministic. However, we cannot reason about properties that arise from applying a method's specification to multiple calls simultaneously. An example of this limitation is shown in Figure 1.4. To prove more properties about method calls, we require a more expressive way to instruct Hypra on how a method specification should be applied to a method call or multiple method calls.

To solve this problem, we added several new structures to Hypra:

- **Named Specifications:** Allow us to have multiple specifications in one method.
- **Logical Variables:** Allow us to label sets of states in a specification to relate them to each other.
- **Apply Statement:** This new statement enables us to modify the application of a named method specification across one or multiple method calls, thus enabling us to verify properties that relate them to each other.

Named specifications server two purposes: firstly, they enable the inclusion of multiple specifications with conflicting pre- and postconditions within a single method; secondly, they allow us to specify which specification is applied using the apply statement.

Logical Variables enable the tagging of states within a method's specification. Although we can technically use a program variable for this purpose, it

would impact the program's behavior, which we aim to avoid.

Finally, with the introduction of the apply statement, we can apply a modified method specification (which typically is a hyperproperty) to a method call or multiple method calls. More specifically, this statement applies a method specification to the set of states that can occur before or after a method call, or multiple calls. It has two parts: the first part allows us to specify which named specification to apply, in the case that a method has various specifications. The second part enables us to modify the method specification so that it is applicable to our particular context. For this second part, we designed a new Domain-Specific Language (DSL) that enables us to manipulate the specification and on which set of states S we want to apply the specification.

Hypra then translates the DSL-expression together with the named specification specified in the apply statement into an assertion that Viper can check. It does this with the help of an algorithm that we designed.

2.1 Named Specifications and Logical Variables

In this section, we will introduce named specifications and logical variables, both of which are features we use in our DSL and are general improvements to Hypra.

2.1.1 Named Specifications

A single method can satisfy multiple distinct hyperproperties. For example, both hyperproperties determinism and monotonicity hold for a method that adds one to its input. Those two hyperproperties have different preconditions and postconditions; yet, they often both hold on specific methods. However, when calling a method with multiple distinct properties, a program may only satisfy the precondition of one of them, and therefore asserting both of the preconditions on method call would fail.

To address this limitation, we introduced the ability to assign names to specifications, allowing us to apply them individually to a method call via the apply statement. This also allows us to write hyperproperties with mutually exclusive pre- and postconditions without assuming false. We also changed Hypra to not automatically assert/assume the pre-and postconditions for named specifications, since they might not always hold for the current set of states.

With the ability to name specifications comes the need to name loop invariants. For instance, in Figure 2.1, certain invariants do not hold for some preconditions but are necessary for the postconditions of a different specification.

```
method nTimesX(n : Int, x : Int) returns (y : Int)
 geZero: requires ∀⟨σ⟩. σ(x) ≥ 0
 geZero: ensures ∀⟨σ⟩. σ(y) ≥ 0
 leZero: requires ∀⟨σ⟩. σ(x) ≤ 0
 leZero: ensures ∀⟨σ⟩. σ(y) ≤ 0
{
 y := 0
 while(n > 0)
  geZero: invariant ∀⟨σ⟩. σ(x) ≥ 0 ∧ σ(y) ≥ 0
  leZero: invariant ∀⟨σ⟩. σ(x) ≤ 0 ∧ σ(y) ≤ 0
 {
  y := y + x
  n := n − 1
 }
}
```

Figure 2.1: The figure illustrates a method that adds a number to itself n times. The key point is that proving two different properties of this method (one for positive numbers, one for negative) requires two distinct loop invariants. Using the wrong invariant for either property will cause the proof to fail.

This leads us to the concept that all *ghost* statements should be able to have a name, since the same issue as with invariants presents itself with them. Furthermore, our concept of unnamed *ghost* statements has changed as follows:

- **Named Specification:** a specification that holds for a method but does not need to hold on method call.
- **Named *Ghost* Statement:** is part of the proof for a named specification.
- **Unnamed Specification:** a specification that holds for all named specifications and also needs to hold on method call.
- **Unnamed *Ghost* Statement:** is part of the proof for each named and unnamed specification.

One can think of an unnamed specification as a default specification that always has to hold on method call and for each other specification.

Rules for Named Specifications The rules for named specifications are simple:

- All *ghost* statements can have a name. These include:
  - preconditions and postconditions
  - invariants

  - hyperasserts and hyperassumes
  - the newly added apply statement
  - use statements (for hints)
- All statements without a name are treated as if they apply to each named specification (this is a sort of default).
- A name can only be used if it was defined as a part of a named specification.

2.1.2 Logical Variables

Sometimes, it is essential to distinguish between different input states in a specification. For example (Figure 2.2), if we have a monotonic method, when one input is larger than another, its output will be larger than or equal to the output of the smaller input. Such a specification can be easily expressed with the use of logical variables (HHL page 2.2 [2] and Kleymann [7]) to tag different input states:

$$\{\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(x) \leq \sigma_2(x)\}$$
$$\text{S}$$
$$\{\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(y) \leq \sigma_2(y)\}$$

Figure 2.2: A monotonicity specification using logical variables to tag different sets of states.

The introduction of logical variables is also partially motivated by the DSL. By adding the ability to use logical variables in specifications, we make the DSL 2.3 more expressive. Logical variables are used to tag states and relate them to each other within a specification. This means that when using the DSL to apply a method specification on multiple different method calls, we can give each different call a (unique) tag corresponding to the specification. Figure 2.8 provides an example of this behavior.

However, how do we assign logical variables different values to tag the states when we want to use the specifications that contain them? Since logical variables are immutable, they should be assigned a value on method call. However, there are numerous cases where we want to use a specification that utilizes logical variables on differently tagged states (see Figure 2.3).

Therefore, we cannot assign a single value to them on method call, but we assign values to them when we want to apply the specifications. This limitation makes sense, since logical variables are not part of the program and are only part of the proof; therefore, we should only be able to use them in statements that are part of the proof.

method f(x : **Int**) **logical** (t : **Int**) **returns** (y : **Int**)
 mono: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(x) = \sigma_2(x)$
 mono: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(y) = \sigma_2(y)$
method main(x : **Int**) **returns** (u : **Int**, v : **Int**, w : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\langle\sigma\rangle.\, \sigma(u) \leq \sigma(v) \land \sigma(v) \leq \sigma(w)$
{
 $u := \mathrm{f}(x)$ @call$_1$
 $v := \mathrm{f}(x + 10)$ @call$_2$
 $w := \mathrm{f}(x + 100)$ @call$_3$
 apply mono **on** @call$_1(t \to 1) \cup$ @call$_2(t \to 2)$
 apply mono **on** @call$_2(t \to 1) \cup$ @call$_3(t \to 2)$
}

Figure 2.3: This is the same example as Figure 2.8. However, here we use three different method calls to illustrate that we need the ability to give different assignments to the same logical variable in different apply statements. If we didn't have this ability @call$_2$ could only have a fixed value for the logical variable t and therefore either the first or the second apply statement would fail.

There still exists one ambiguity: if a logical variable is used in an unnamed specification, then we assert/assume the pre-/postcondition when calling this method, but the logical variables in the specification do not have a value. In such a case, no state in the assertion has a value defined for the logical variables, meaning that the assertion would fail. To solve this issue, we disallow the use of logical variables in unnamed statements and require them to be assigned to a value (via logical variable assignment in the DSL 2.8) in each apply statement.

Rules for Logical Variables The rules for logical variables directly follow from the fact that they can only be assigned in apply statements:

- Only named statements can use logical variables.
- For a logical variable to be defined, it must be declared in the method head under logical variables and used in at least one pre- or postcondition of the named specification for which it is defined.
- If we use an apply statement of a specification with a defined logical variable, we require the user to assign it a value via logical variable assignment.

2.1.3 Implementation

The implementation strategy for named specifications involves generating a distinct method in Viper for each specification, ensuring it matches the correct specification. In Figure 2.4, we show the old way of proving multiple

different properties over one method, along with the new way. The code generated in Viper resembles what the old example would have produced. However, our approach introduces additional nuances: since we allow for unnamed specifications (specifications that also need to hold for all named specifications), we also have to generate a base method that verifies the unnamed specifications. Additionally, for each named specification, we also generate the unnamed proof statements and specifications.

method add_one_det(x : **Int**) **returns** (y : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x)$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}
method add_one_mono(x : **Int**, t : **Bool**) **returns** (y : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(x) \leq \sigma_2(x)$
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(y) \leq \sigma_2(y)$
{
 $y := x + 1$
}
method add_one(x : **Int**) **logical** (t : **Bool**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(y)$
 mono: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) \wedge \neg\sigma_2(t) \Rightarrow \sigma_1(x) \leq \sigma_2(x)$
 mono: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) \wedge \neg\sigma_2(t) \Rightarrow \sigma_1(y) \leq \sigma_2(y)$
{
 $y := x + 1$
}

Figure 2.4: In this example we prove that determinism and monotonicity both hold on a method that just adds one to the input. The first two methods show how we had to prove this before adding logical variables and named specifications to Hypra, by creating the same method twice with different pre- and postconditions. The last method shows how to prove those hyperproperties with just one method and named specifications. While the logical variable t replaces the program variable t used before to prove the mono specification, clearly stating that t is not part of the program and ensuring immutability.

As for the logical variables, we treat them as if they are normal variables in the generated methods; however, in the symbol checker, we enforce the rules for logical variables. This restriction implies that they cannot be changed or interact with the program, as named statements cannot interact with the program's behavior.

2.2 Labels

We added the ability to assign a label to each method call. This allows us to reference the method calls later in our DSL 2.3. In the Introduction 1 section, we briefly mentioned that Hypra works by tracking a set of reachable states over a program. Since each method call interacts with two sets of reachable states, the set of states before the call and the set of states after the call, a label needs to be able to reference both sets of states. For the implementation of labels, we leveraged Hypra's approach to tracking sets of reachable states over a program to our advantage. Whenever Hypra encounters a method call with a label, it saves the set of states in a unique variable.

```
method main(x : Int)
returns (y : Int)
{
  y := f(x) @call
}
```

```
var S_label_pre_call: SetState
var S_label_post_call: SetState
... some other code encoding the program ...
S_label_pre_call := S
... the encoding of the method call by framing the unchanged variables aswell as asserting the preconditions of the method on S and assuming the postconditions of the call on a new S...
S_label_post_call := S
... the encoding of the rest of the program ...
```

Figure 2.5: In this figure we show the concrete Viper encoding of a Label named @call on the right side and the Hypra program on the left side. Here S denotes the current set of states before and after the method call.

2.3 Grammar and Features

In this section, we will briefly introduce the grammar of our DSL, followed by introducing and motivating each feature with a simple example.

2.3.1 Grammar

Now that we have introduced the necessary features (Logical Variables, Named Specifications, and Labels) for the apply statement's DSL, we will present its grammar:

$$E_{\text{top}} ::= \forall x.E \mid \forall x.b_{\text{top}} \Rightarrow E$$
$$E ::= @l \mid E \cup E \mid E[b] \mid E(t \rightarrow e)$$

In this grammar, $@l$ represents a label, while b represents a boolean predicate over program variables and top-level quantified variables, t is a logical variable, and e is an expression that evaluates to the type of t. For the top level, x can be any quantified variable of any supported type, and b_{top} is a boolean predicate over the top-level quantified variables. E_{top} cannot be used recursively, as E is a distinct variable.

Now that we have the grammar of the DSL, we explain and motivate the purpose of each feature.

2.3.2 Labels

We have earlier (in Section 2.2) introduced the ability to tag each method call with a label. Labels in our DSL allow us to reference specific method calls. They are the fundamental building blocks of the DSL and are therefore required to apply a specification. Intuitively, this can be explained since applying a specification of a method to nothing would also result in nothing.

2.3.3 Unions

method f(x : **Int**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}
method main(u : **Int**) **returns** (v : **Int**, w : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(u) = \sigma_2(u)$
 ensures $\forall\langle\sigma\rangle.\, \sigma(v) = \sigma(w)$
{
 $v := \text{f}(u)$ $@\text{call}_1$
 $w := \text{f}(u)$ $@\text{call}_2$
 apply det **on** $@\text{call}_1 \cup @\text{call}_2$
}

Figure 2.6: This is our proposed solution for overcoming the limitation in Figure 1.3. To solve this example we added a way to reference the function call via labels and also added a name (det) to the determinism specification. Then we used the union ($\cup$) operation to state that we want to apply our specification on the union of both method calls. However, the precondition on main in this example is stronger that in Figure 1.3, and we will solve this issue later with top-level operators.

The union ($\cup$) operator is the binary operator that allows us to combine multiple specifications of different (or the same) method calls. It enables us to apply a specification on multiple method calls simultaneously, resulting in

the ability to reason about programs with non-trivial properties that arise from having several method calls. An example for using unions and labels in the DSL can be found in Figure 2.6.

2.3.4 Projections

In many cases, a method specification may only hold for a subset of states that can occur at the method calls. However, we should still be able to use a specification on this subset (see Figure 2.7) to prove a property. To handle this case, a feature of the DSL is projections, which allow us to project the set of states of a label to a smaller subset for which a predicate holds.

method f(x : **Int**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(y) = \sigma_2(y)$
{
 $y := x + 1$
}
method main(u : **Int**) **returns** (v : **Int**)
 requires $\forall\langle\sigma\rangle. \sigma(u) = 1 \vee \sigma(u) = 5$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle, \langle\sigma_3\rangle. \sigma_1(v) = \sigma_2(v) \vee \sigma_1(v) = \sigma_3(v) \vee \sigma_2(v) = \sigma_3(v)$
{
 $v :=$ f(u) @call
 apply det **on** @call$[u = 1]$
 apply det **on** @call$[u = 5]$
}

Figure 2.7: A small example motivating projections. The specification of `main` proves that if the only values for v are 1 or 5 then the output can also have at most two different values. In this example the `det` precondition of f would not hold in the apply statement since u can be either 1 or 5.

2.3.5 Logical Variable Assignment

Logical Variables (Dardinier and Müller [2] section 2.2, and Kleymann [7]) are a powerful tool for tagging specific states, which then allows us to express properties on those tagged states. Since we can use logical variables in method specifications, want to use the DSL to tag states when applying such a specification by assigning them a value. Therefore, we introduce the ability to assign values to logical variables in the DSL. This behavior is illustrated in Figure 2.8.

method f(x : **Int**) **logical** (t : **Int**) **returns** (y : **Int**)
 mono: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(x) \leq \sigma_2(x)$
 mono: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(y) \leq \sigma_2(y)$
method main(u : **Int**) **returns** (v : **Int**, w : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(u) = \sigma_2(u)$
 ensures $\forall\langle\sigma\rangle.\, \sigma(v) \leq \sigma(w)$
{
 $v :=$ f(u) @call$_1$
 $w :=$ f($u + 10$) @call$_2$
 apply mono **on** @call$_1(t \to 1) \cup$ @call$_2(t \to 2)$
}

Figure 2.8: In this example we have a monotonic method f that uses a logical variable t for its method specification mono. To then apply this property on the two method calls we need to specify a value for t since the monotonic property requires one.

2.3.6 Universal Quantification

With projections in our DSL, we can prove a method specification on a subset of our original set of states. However, for most hypersafety properties, it is often necessary to apply a specification on the entire set of states rather than just a subset. To prove such a property, we may need to apply a specification on multiple subsets so that it is then applied to the whole set of states. However, this may require an unbounded amount of apply statements in some cases. To address this issue, we introduce a top-level forall quantifier, which enables us to write multiple apply statements within a single construct.

method f(x : **Int**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$
method main(u : **Int**) **returns** (v : **Int**, w : **Int**)
 ensures $\forall\langle\sigma\rangle.\, \sigma(v) = \sigma(w)$
{
 $v :=$ det(u) @call$_1$
 $w :=$ det(u) @call$_2$
 apply det **on** $\forall i$: **Int**. (@call$_1 \cup$ @call$_2)[u = i]$
}

Figure 2.9: This is the initial motivating example from Figure 1.3. However, compared to the example in Figure 2.6 we do not require a precondition for `main`, as we have the ability to apply the `det` specification projected on each possible value with the help of a top-level forall quantifier.

This top-level forall can be combined with an implication to narrow its scope. This is useful since even a projected specification might not hold for every value in the quantifier's range.

2.4 The Translation Algorithm

As previously discussed, we aim to apply a method specification to a set of states that can be altered through our DSL. Each method specification has a pre- and postcondition. To ensure correctness, we must assert the precondition of the specification on the set of states before the function call(s) and assume the postcondition on the set of states after the function call(s).

In the translation of the apply statement, we follow the same approach. We generate a modified assertion from the pre- and postconditions to then later assert/assume. To achieve this, we developed an algorithm that accepts an assertion in the form of a pre- or postcondition and an apply expression, which specifies how it should be modified, and then generates a new assertion. The apply expression states on which set of states the assertion should be applied, as well as how it should be altered.

method f(x : **Int**)
returns (y : **Int**)
 det: **requires**
  $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x)$
 det: **ensures**
  $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(y)$

method main(u : **Int**, v : **Int**)
returns (w : **Int**, x : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(u) = \sigma_2(v)$
{
 $w := f(u)$ @call$_1$
 $x := f(v)$ @call$_2$
 apply det **on** @call$_1$ $\cup$ @call$_2$
}

The **apply** line gets translated to
assert
$\forall\langle\sigma_1$@call$_1\rangle, \langle\sigma_2$@call$_1\rangle.\sigma_1(u)=\sigma_2(u)$
$\wedge\forall\langle\sigma_1$@call$_1\rangle, \langle\sigma_2$@call$_2\rangle.\sigma_1(u)=\sigma_2(v)$
$\wedge\forall\langle\sigma_1$@call$_2\rangle, \langle\sigma_2$@call$_1\rangle.\sigma_1(v)=\sigma_2(u)$
$\wedge\forall\langle\sigma_1$@call$_2\rangle, \langle\sigma_2$@call$_2\rangle.\sigma_1(v)=\sigma_2(v)$

assume
$\forall\langle\sigma_1$@call$'_1\rangle, \langle\sigma_2$@call$'_1\rangle.\sigma_1(w)=\sigma_2(w)$
$\wedge\forall\langle\sigma_1$@call$'_1\rangle, \langle\sigma_2$@call$'_2\rangle.\sigma_1(w)=\sigma_2(x)$
$\wedge\forall\langle\sigma_1$@call$'_2\rangle, \langle\sigma_2$@call$'_1\rangle.\sigma_1(x)=\sigma_2(w)$
$\wedge\forall\langle\sigma_1$@call$'_2\rangle, \langle\sigma_2$@call$'_2\rangle.\sigma_1(x)=\sigma_2(x)$

Figure 2.10: The translation of the pre- and postcondition of the specification det applied on the DSL stating @call$_1$ $\cup$ @call$_2$. The notation $\langle\sigma$@call$\rangle$ stands for a state before the method call labeled with @call, while $\langle\sigma$@call'$\rangle$ stands for a state after the method call respectively.

An example of the translation of a simple apply statement can be found in Figure 2.10. In this example, the `det` specification, combined with a union operator in the apply expression, generates a conjunction of four assertions. Each assertion corresponds to one of the possible combinations of states being part of the first or second method call. This can also be seen in the variables that have been changed to correctly reflect the variables used at those method calls. Finally, the conjunction arises from the fact that we have a forall operator in the `det` specification, which means that each combination of states and labels must hold.

2.4.1 Definition of the Algorithm

The algorithm we created recurses over the specification with **rSpec** as well as over the apply expression with **rDSL**.

Each specification has a pre- and postcondition. When translating each one, the difference lies in that we assert the precondition and assume the postcondition, as well as in the fact that the precondition references the set of states before each of the calls referenced by the labels, and the postcondition references the set of states after each of the calls referenced by the labels.

A Simplified Setting

To gain an intuition for how the algorithm works, we first examine a simpler setting. In this setting, the DSL only supports unions and labels, meaning that all we can do in this DSL is apply a specification to multiple method calls.

We saw in Figure 2.10 that when creating the new assertion, we have to consider each possible combination for the quantified state variable and the label used in the DSL. This means that for every quantified state, we have to generate a separate assertion for each label.

This leads us to a simplified algorithm, where we have two different recursive functions: **rSpec** takes as input an assertion (in the first call, this is a specification pre- or postcondition), and recurses down this assertion until it finds a quantifier that quantifies over states; then, it calls **rDSL** to generate the correct assertion for this quantified state. Then, **rDSL** generates a new assertion for each of the labels by copying the rest of the assertion being quantified over. For each copy, it replaces the quantified state with a state corresponding to the label. Finally, it combines the generated expressions into one by using an **AND** ($\wedge$) or an **OR** ($\vee$) for either a $\forall$ or an $\exists$ quantifier.

This last step can be explained intuitively: for a $\forall$ quantifier, all combinations of states must hold, and for an $\exists$ quantifier, at least one of the combinations must hold.

The definition of an algorithm translating the mini DSL would look like:

$$
\begin{aligned}
&\mathbf{rSpec}(\mathcal{Q}\langle\sigma\rangle.A, E) && \triangleq \mathbf{rDSL}(A, E, E, \mathcal{Q})\\
&\mathbf{rDSL}(A, E_1 \cup E_2, E_o, \forall) && \triangleq \mathbf{rDSL}(A, E_1, E_o, \forall) \wedge \mathbf{rDSL}(A, E_2, E_o, \forall)\\
&\mathbf{rDSL}(A, E_1 \cup E_2, E_o, \exists) && \triangleq \mathbf{rDSL}(A, E_1, E_o, \exists) \vee \mathbf{rDSL}(A, E_2, E_o, \exists)\\
&\mathbf{rDSL}(A, @label, E_o, \mathcal{Q}) && \triangleq \mathcal{Q}\langle s@\text{label}\rangle.\mathbf{rSpec}(A, E_o)\\
&\mathbf{rSpec}(A_1 \wedge A_2, E) && \triangleq \mathbf{rSpec}(A_1, E) \wedge \mathbf{rSpec}(A_2, E)\\
&\mathbf{rSpec}(A_1 \vee A_2, E) && \triangleq \mathbf{rSpec}(A_1, E) \vee \mathbf{rSpec}(A_2, E)\\
&\mathbf{rSpec}(A_1 \Rightarrow A_2, E) && \triangleq \mathbf{rSpec}(A_1, E) \Rightarrow \mathbf{rSpec}(A_2, E)\\
&\mathbf{rSpec}(\neg A, E) && \triangleq \neg\mathbf{rSpec}(A, E)\\
&\mathbf{rSpec}(|A|, E) && \triangleq |\mathbf{rSpec}(A, E)|\\
&\mathbf{rSpec}(l, E) && \triangleq l\text{: here } l \text{ is either a literal or a variable.}
\end{aligned}
$$

In the above definition, A is the assertion (initially a pre-/postcondition), E is an apply expression, and $\mathcal{Q}$ represents a quantifier. All cases with only **rSpec** correspond to our algorithm recursing down the assertion to find the next quantified state variable, except for the first case which is called if such a variable has been found. The cases for **rDSL** correspond to the translation of the apply expression for one quantified state.

In this algorithm, we must keep track of the original apply expression E_o, as we need to generate the correct assertion whenever **rSpec** encounters a quantifier. We do this because, after **rDSL** finishes and calls **rSpec** again, the original apply expression would be lost if E_o is not passed down. This would prevent **rDSL** from creating the correct assertion for all subsequent state quantifiers found by **rDSL**.

The Full Algorithm

Now that we have an intuition for how this algorithm should work, let's examine the entire algorithm. The outer function is defined as $\mathbf{rSpec}(A, E, p)$ where:

- A: An assertion that is part of or the entire specification to be translated.
- E: An apply expression on which the specification has to be applied.
- p: A boolean stating if E is part of a precondition or postcondition.

And it recurses as follows:

$$
\begin{aligned}
\textbf{rSpec}(\mathcal{Q}\langle s\rangle.A, E, p) &\triangleq \textbf{rDSL}(A, E, E, \mathcal{Q}, s, p)\\
\textbf{rSpec}(A_1 \wedge A_2, E, p) &\triangleq \textbf{rSpec}(A_1, E, p) \wedge \textbf{rSpec}(A_2, E, p)\\
\textbf{rSpec}(A_1 \vee A_2, E, p) &\triangleq \textbf{rSpec}(A_1, E, p) \vee \textbf{rSpec}(A_2, E, p)\\
\textbf{rSpec}(A_1 \Rightarrow A_2, E, p) &\triangleq \textbf{rSpec}(A_1, E, p) \Rightarrow \textbf{rSpec}(A_2, E, p)\\
\textbf{rSpec}(\neg A, E, p) &\triangleq \neg\textbf{rSpec}(A, E, p)\\
\textbf{rSpec}(|A|, E, p) &\triangleq |\textbf{rSpec}(A, E, p)|\\
\textbf{rSpec}(l, E, p) &\triangleq l\text{: here } l \text{ is either a literal or a variable.}
\end{aligned}
$$

Here, $\mathcal{Q}\langle s\rangle.E$ represents a hyperassertion where we assert over the state s, where $\mathcal{Q}$ can be either a $\forall$ or $\exists$ quantifier. The remaining cases are simple; they essentially recurse down the AST of the assertion until it either reaches a leaf or encounters another quantifier quantifying over states.

As explained in the previous section, the idea of **rSpec** is that it recurses down the assertion of the pre- or postcondition until it finds a quantified state for which it then calls **rDSL**. This means that if our assertion lacks quantified states, **rSpec** returns the same assertion as in the input.

The inner function is defined as $\textbf{rDSL}(A, E_c, E_o, \mathcal{Q}, s, p)$ where:

- A: An assertion that is part of or the entire specification to be translated.
- E_c: The current apply expression that is being recursed.
- E_o: The original apply expression that was part of the apply statement.
- $\mathcal{Q}$: The current quantifier being recursed over.
- s: The name of the quantified state variable.
- p: A boolean stating if E is part of a precondition or postcondition.

The base case of the recursion is when we hit a label in the apply expression

(the label case). The recursion is as follows:

$$\begin{aligned}
&\mathbf{rDSL}(A, @\text{label}, E_o, \mathcal{Q}, s, p) && \triangleq \mathcal{Q}\langle s@\text{label}\rangle.\mathbf{rSpec}(\\
&&&\quad \mathbf{mapProgVars}(A, s, \mathbf{getMapping}(@\text{label})), E_o, p) && \text{(label case)}\\
&\mathbf{rDSL}(A, E_1 \cup E_2, E_o, \forall, s, p) && \triangleq \mathbf{rDSL}(A, E_1, E_o, \forall, s, p) \wedge \mathbf{rDSL}(A, E_2, E_o, \forall, s, p)\\
&\mathbf{rDSL}(A, E_1 \cup E_2, E_o, \exists, s, p) && \triangleq \mathbf{rDSL}(A, E_1, E_o, \exists, s, p) \vee \mathbf{rDSL}(A, E_2, E_o, \exists, s, p) && \text{(union cases)}\\
&\mathbf{rDSL}(A, E[b], E_o, \forall, s, p) && \triangleq \mathbf{rDSL}(s(b) \Rightarrow A, E, E_o, \forall, s, p)\\
&\mathbf{rDSL}(A, E[b], E_o, \exists, s, p) && \triangleq \mathbf{rDSL}(s(b) \wedge A, E, E_o, \exists, s, p) && \text{(projection cases)}\\
&\mathbf{rDSL}(A, E(t \to v), E_o, \forall, s, p) && \triangleq \mathbf{rDSL}(\mathbf{mapProgVars}(A, s, (t \to v)), E, E_o, \forall, s, p) && \text{(logical variable assignment)}\\
&\mathbf{rDSL}(A, \forall x.E, E_o, \mathcal{Q}, s, p) && \triangleq \forall x.\mathbf{rSpec}(A, E, E, \mathcal{Q}, s, p)\\
&\mathbf{rDSL}(A, \forall x.e \Rightarrow E, E_o, \mathcal{Q}, s, p) && \triangleq \forall x.e \Rightarrow \mathbf{rSpec}(A, E, E, \mathcal{Q}, s, p) && \text{(top-level cases)}
\end{aligned}$$

Label Case Here, it is important to note that the set of states referenced by $\langle s@\text{label}\rangle$ depends on p, because if p states that the assertion is part of a pre-condition, s references the set of states before the method call of the label, and the other way around. Additionally, the function $\mathbf{mapProgVars}(A, s, @\text{label})$ accepts an assertion and a state variable s, along with a mapping, and maps the program variables that depend on s with the mapping. $\mathbf{getMapping}(@\text{call})$ takes in a label and returns a mapping that maps program variables of the method to the program variables of the method call. We do this because when a method is called, the input variable names do not need to match the variables used in the method specification. Therefore, the translation of the specification needs to mention the variable names that have been used in the method call, rather than the variables of the method itself. An example of this behavior can be found in Figure 2.10, where the translation mentions the input or return variables of the method calls, respectively.

Union Cases The cases where we want to create a union of two different function calls are simple to translate. For a $\forall$ quantifier, we encode the union as an **AND** ($\wedge$), and for an $\exists$ quantifier, as an **OR** ($\vee$). This translation mimics the translation of our simple algorithm, except that we do it recursively. We already showed how unions are translated in Figure 2.10.

Projection Cases Furthermore, we have cases corresponding to projections. For this, we add this predicate to the expression either as the left-hand side of an implication for a $\forall$ quantifier or as an additional conjuction for an $\exists$

quantifier. For an example of such a translation, recall the apply statement in Figure 2.7:

$$\textbf{apply}\ \text{det}\ \textbf{on}\ @\text{call}[u = 1]$$

For which the precondition gets translated to:

$$\forall\langle\sigma_1@\text{call}\rangle, \langle\sigma_2@\text{call}\rangle.\, \sigma_1(u = 1) \Rightarrow (\sigma_2(u = 1) \Rightarrow \sigma_1(u) = \sigma_2(u))$$

Logical Variable Assignment Case For the logical variable assignment case, we can reuse the **mapProgVars** function, along with the mapping provided in the DSL, because logical variables are handled similarly to program variables. Recall the motivating example in Figure 2.8 for logical variable assignment, in which we have an apply statement.

$$\textbf{apply}\ \text{mono}\ \textbf{on}\ @\text{call}_1(t \to 1) \cup @\text{call}_2(t \to 1)$$

For such an apply statement, the precondition gets translated to:

$$\begin{aligned}
&\forall\langle\sigma_1@\text{call}_1\rangle, \langle\sigma_2@\text{call}_1\rangle.\, 1 = 1 \wedge 1 = 2 \Rightarrow \sigma_1(u) \leq \sigma_2(u)\\
\wedge&\forall\langle\sigma_1@\text{call}_1\rangle, \langle\sigma_2@\text{call}_2\rangle.\, 1 = 1 \wedge 2 = 2 \Rightarrow \sigma_1(u) \leq \sigma_2(u + 10)\\
\wedge&\forall\langle\sigma_1@\text{call}_2\rangle, \langle\sigma_2@\text{call}_1\rangle.\, 2 = 1 \wedge 1 = 2 \Rightarrow \sigma_1(u + 10) \leq \sigma_2(u)\\
\wedge&\forall\langle\sigma_1@\text{call}_2\rangle, \langle\sigma_2@\text{call}_2\rangle.\, 2 = 1 \wedge 2 = 2 \Rightarrow \sigma_1(u + 10) \leq \sigma_2(u + 10)
\end{aligned}$$

In this translation, we clearly see that only the second case needs to hold from the implication, which is also the correct one.

Top-Level Cases The final two cases are those involving top-level quantifiers. These cases are handled recursively as if they could appear at any point in the DSL; however, we do not allow that in the symbol checker. Additionally, since they are only at the top level, we call **rSpec** with E instead of the original E_o, permanently modifying the apply expression to remove those top-level operators, as they should not be recursed. Finally, both of these cases are checked before being translated by **rSpec**, as having them translated for each of the labels would result in multiple generated quantifiers or implications.

The main idea of this algorithm is that the **rSpec** part recurses down the specification, trying to find a quantified state variable. Once it finds this variable, it will call **rDSL** to translate the apply expression with the quantified variable. Then **rDSL** will recurse down the apply expression until it finds a leaf in the form of a @label. Afterwards, it will call **rSpec** again on the remaining method specification with the quantifier translated.

Triggers

Triggers are a challenging aspect of many verification tools; in Hypra, triggers are automatically generated for preconditions and invariants that contain quantified states. The strategy for generating triggers can be found in section 3.2 of Dardinier *et al.* [5]. It involves normalizing the assertion first and then determining the correct trigger based on the highest level state quantifier. To generate the correct triggers for our hyperassertions generated by an apply statement, we use the same algorithm.

However, when using a top-level forall quantifier, we need to generate triggers differently, as we do not quantify over states. Therefore, we added the requirement for the user to specify at least one trigger when using a to-level forall quantifier in the DSL. We can then reuse the hinting infrastructure already present in Hypra to trigger this trigger.

2.4.2 Length of the Generated Assertion

For this algorithm, estimating the length of the generated assertion is interesting, as it helps estimate the additional burden from the apply statements on the SMT solver.

In the following proof of an upper bound on the size we will first prove lemma 2.1 and lemma 2.2, both of which allow us to normalize each apply expression. Then we use lemma 2.3 to simplify our normalized expression into one for which it is easy to prove an upper bound. Finally, for this expression, we then prove the upper bound by induction.

Definition without Auxiliary Variables

To determine the length of an assertion generated by an apply statement, we go over the definitions again. The variable p only determines whether the final assertion is an assert or an assume statement in Viper and therefore does not impact the size of the assertion. Similar to p, the quantified variable s only impacts which variables are replaced in the assertion, and thus s also does not affect the size of the final assertion. Consequently, we can simplify the algorithm to only use E, A, and Q for this analysis.

We also omit the case for logical variable assignment, since most of the time we use this operation to exchange a symbol one-to-one. We do this so the proof is not overly complex; however, the case for logical variable assignment would be handled much like we do for projections in this proof. Furthermore, we now briefly list the new algorithm again, for which we proved an upper

bound on the size:

$$
\begin{aligned}
&\mathbf{rSpec}(l, A) && \triangleq l \\
&\mathbf{rSpec}(A_1 * A_2, E) && \triangleq \mathbf{rSpec}(A_1, E) * \mathbf{rSpec}(A_2, E) \\
&\mathbf{rSpec}(\mathbf{op}(A), E) && \triangleq \mathbf{op}(\mathbf{rSpec}(A, E)) \\
&\mathbf{rSpec}(\mathcal{Q}\langle s\rangle.A, E) && \triangleq \mathbf{rDSL}(A, E, E, \mathcal{Q}) \\
&\mathbf{rDSL}(A, E[b], E_o, \mathcal{Q}) && \triangleq \mathbf{rDSL}(b(\Rightarrow \text{or} \wedge)A, E, E_o, \mathcal{Q}) \\
&\mathbf{rDSL}(A, A_1 \cup E_2, E_o, \mathcal{Q}) && \triangleq \mathbf{rDSL}(A, E_1, E_o, \mathcal{Q})(\wedge \text{or} \vee)\mathbf{rDSL}(A, E_2, E_o, \mathcal{Q}) \\
&\mathbf{rDSL}(A, @\text{label}, E_o, \mathcal{Q}) && \triangleq \mathbf{rSpec}(A, E_o, \mathcal{Q}) \\
&\mathbf{rDSL}(A, \forall x.E, E_o, \mathcal{Q}) && \triangleq \forall x.\mathbf{rSpec}(A, E) \\
&\mathbf{rDSL}(A, \forall x.e \Rightarrow E, E_o, \mathcal{Q}) && \triangleq \forall x.e \Rightarrow \mathbf{rSpec}(A, E)
\end{aligned}
$$

Lemmas

To get an upper bound for the size of the final assertion, we first need to prove some lemmas. The first two lemmas will be used to normalize each apply expression, while the last one will be used to simplify the normalized expression into an expression for which we can easily prove an upper bound.

Lemma 2.1 *Projections distribute over the union operator in **rDSL**.*

We can know that the omitted quantifier $\mathcal{Q}$ is constant when recursing down the apply expression, since this variable can only change in the initial call of **rDSL**; therefore, the translation of the union operator always yields the same binary operator. Without loss of generality, we assume that in the current call, $\mathcal{Q}$ is $\forall$:

$$
\begin{aligned}
\mathbf{rDSL}(A, (E_1 \cup E_2)[b], E_o, \forall) &= \mathbf{rDSL}(b \Rightarrow A, E_1 \cup E_2, E_o, \forall) \\
&= \mathbf{rDSL}(b \Rightarrow A, E_1, E_o, \forall) \wedge \mathbf{rDSL}(b \Rightarrow A, E_2, E_o, \forall) \\
&= \mathbf{rDSL}(A, E_1[b], E_o, \forall) \wedge \mathbf{rDSL}(A, E_2[b], E_o, \forall) \\
&= \mathbf{rDSL}(A, (E_1[b] \cup E_2[b]), E_o, \forall)
\end{aligned}
$$

Lemma 2.2 *In **rDSL**, the union operator is associative.*

For this lemma, we also assume without loss of generality that the current quantifier is a $\forall$ quantifier:

$$
\begin{aligned}
\mathbf{rDSL}(A, (E_1 \cup E_2) \cup E_3, E_o, \forall) &= \mathbf{rDSL}(A, E_1 \cup E_2, E_o, \forall) \wedge \mathbf{rDSL}(A, E_3, E_o, \forall) \\
&= \mathbf{rDSL}(A, E_1, E_o, \forall) \wedge \mathbf{rDSL}(A, E_2, E_o, \forall) \\
&\quad \wedge \mathbf{rDSL}(A, E_3, E_o, \forall) \\
&= \mathbf{rDSL}(A, E_1, E_o, \forall) \wedge \mathbf{rDSL}(A, E_2 \cup E_3, E_o, \forall) \\
&= \mathbf{rDSL}(A, E_1 \cup (E_2 \cup E_3), E_o, \forall)
\end{aligned}
$$

Now let's take a look at the last lemma that we need to get an upper bound for the size:

Lemma 2.3 *The following equation holds*

$$|\textbf{\textit{rDSL}}(A, E_1[b_1] \cup E_2[b_2], E_o, \mathcal{Q})| \leq |\textbf{\textit{rDSL}}(A, (E_1 \cup E_2)[b_{max}], E_o, \mathcal{Q})|$$

While b_{max} is defined as:

$$b_{max} = \begin{cases} b_1 & \textit{if } |b_2| \leq |b_1| \\ b_2 & \textit{otherwise} \end{cases}$$

To prove it, let's simplify each side individually. Here we can also assume without loss of generality that the current quantifier is a $\forall$:

$$\begin{aligned} |\textbf{rDSL}(A, E_1[b_1] \cup E_2[b_2], E_o, \forall)| &= |\textbf{rDSL}(A, E_1[b_1], E_o, \forall) \\ &\quad \wedge \textbf{rDSL}(A, E_2[b_2], E_o, \forall)| \\ &= |\textbf{rDSL}(b_1 \Rightarrow A, E_1, E_o, \forall) \\ &\quad \wedge \textbf{rDSL}(b_2 \Rightarrow A, E_2, E_o, \forall)| \\ &= |\textbf{rDSL}(b_1 \Rightarrow A, E_1, E_o, \forall)| \\ &\quad + |\textbf{rDSL}(b_2 \Rightarrow A, E_2, E_o, \forall)| + 1 \end{aligned}$$

And the other side simplifies to:

$$\begin{aligned} |\textbf{rDSL}(A, (E_1 \cup E_2)[b_{\max}], E_o)| &= |\textbf{rDSL}(b_{\max} \Rightarrow A, E_1 \cup E_2, E_o)| \\ &= |\textbf{rDSL}(b_{\max} \Rightarrow A, E_1, E_o) \\ &\quad \wedge \textbf{rDSL}(b_{\max} \Rightarrow A, E_2, E_o)| \\ &= |\textbf{rDSL}(b_{\max} \Rightarrow A, E_1, E_o)| \\ &\quad + |\textbf{rDSL}(b_{\max} \Rightarrow A, E_2, E_o)| + 1 \end{aligned}$$

Since $|b_1 \Rightarrow A| \leq |b_{\max} \Rightarrow A|$ holds for A_1 and
$|b_2 \Rightarrow A| \leq |b_{\max} \Rightarrow A|$ holds for A_2; together with the assumption that **rDSL** is monotonic, we prove that the initial lemma holds.

Only Top-Level Operators

For the actual proof, let's start simple and consider the case where we already know the assertion generated by **rSpec** without top-level operators, let's call this R:

$$\textbf{rDSL}(A, E, E, \mathcal{Q}) = R$$

Now we can calculate the size with both top-level operators:

$$\begin{aligned}|\mathbf{rDSL}(A, \forall x.e \Rightarrow E, E_o, \mathcal{Q})| &= |\forall x.e \Rightarrow \mathbf{rSpec}(A, E)| \\ &= |\forall x.e \Rightarrow R|\end{aligned}$$

Therefore, we know that both top-level operators only change the size of the assertion precisely by their own size. Now we can consider the simpler case without the top-level operators and try to find the size of R.

Size of rDSL Compared to rSpec

Now, let's determine the size of **rDSL** in comparison to the size of **rSpec**. To do that, we use lemma 2.1 together with lemma 2.2 and the fact that we simplified away logical variable assignment and top-level operators to rewrite the apply expression E:

$$E = @\text{call}_1[b_1] \cup @\text{call}_2[b_2] \cup \cdots \cup @\text{call}_n[b_n]$$

Here n is the total number of labels in the apply expression, and each f is the combination of all projections of higher levels. Now we use lemma 2.3 to obtain the following inequality:

$$\begin{aligned}|\mathbf{rDSL}(A, E, E, \mathcal{Q})| &= |\mathbf{rDSL}(A, @\text{call}_1[b_1] \cup \cdots \cup @\text{call}_n[b_n], E, \mathcal{Q})| \\ &\leq |\mathbf{rDSL}(A, (@\text{call}_1 \cup \cdots \cup @\text{call}_n)[b_{\max}], E, \mathcal{Q})\end{aligned}$$

Now let's look at the size of **rDSL** compared to **rSpec**, but we use $E' = (@\text{call}_1 \cup \cdots \cup @\text{call}_n)[b_{\max}]$ since we know that $|\mathbf{rDSL}(A, E, E, \mathcal{Q})| \leq |\mathbf{rDSL}(A, E', E', \mathcal{Q})|$ from before.

$$\begin{aligned}|\mathbf{rDSL}(A, E', E', \mathcal{Q})| &= |\mathbf{rDSL}(A, (@\text{call}_1 \cup \cdots \cup @\text{call}_n)[b_{\max}], E', \mathcal{Q})| \\ &= |\mathbf{rDSL}(A, @\text{call}_1[b_{\max}], E', \mathcal{Q}) \wedge \ldots \\ &\quad \wedge \mathbf{rDSL}(A, @\text{call}_n[b_{\max}], E', \mathcal{Q})| \\ &= |\mathbf{rDSL}(A, @\text{call}_1[b_{\max}], E', \mathcal{Q})| + \ldots \\ &\quad + |\mathbf{rDSL}(A, @\text{call}_n[b_{\max}], E', \mathcal{Q})| + n \\ &= n|\mathbf{rDSL}(A, @\text{call}[b_{\max}], E', \mathcal{Q})| + n \\ &= n|\mathbf{rDSL}(b_{\max} \Rightarrow A, @\text{call}, E', \mathcal{Q})| + n \\ &= n|\mathbf{rSpec}(b_{\max} \Rightarrow A, E', \mathcal{Q})| + n\end{aligned}$$

Base Size Of The Algorithm

Let's examine the case where A does not include any quantifiers over states. We refer to this as A_0. From the definition of **rSpec**, we know that if there are no quantifiers in A, **rDSL** never gets called. Then **rSpec** simplifies to only

recurse down the assertion without interacting with the result. We can see this in the following cases:

$$\begin{aligned}\mathbf{rSpec}(l, E) &= l \\ \mathbf{rSpec}(A_1 * A_2, E) &= \mathbf{rSpec}(A_1, E) * \mathbf{rSpec}(A_2, E) \\ \mathbf{rSpec}(\mathbf{op}(A), E) &= \mathbf{op}(\mathbf{rSpec}(A, E))\end{aligned}$$

From which we get the following property:

$$\mathbf{rSpec}(A_0, E) = A_0$$

Proof by Induction

Now we have all the pieces for an induction for the following induction hypothesis:

$$|\mathbf{rSpec}(A, E)| \leq \begin{cases} (|b_{\max}| + 2)\frac{n^{m+1}-n}{n-1} + |A|n^m & \text{if } n > 1 \\ m(|b_{\max}| + 2) + |A| & \text{if } n = 1 \end{cases}$$

Here, m denotes the number of quantified states in A and n the number of labels used in A. In this formula, $b_{\max}$ is the largest predicate after simplifying E to the form:

$$E = @\text{call}_1[b_1] \cup \cdots \cup @\text{call}_k[b_{\max}] \cup \cdots \cup @\text{call}_n[b_n]$$

Base Case for $n > 1$ For the base case, let's say that A_0 is any expression that has no quantifiers over states. We know from before that if there are no states in A, then **rSpec** returns A; therefore:

$$\begin{aligned}|\mathbf{rSpec}(A_0, E)| &= |A_0| \\ &= 0 + |A_0| \\ &= (|b_{\max}| + 2)\frac{n^1 - n}{n - 1} + |A_0|n^0\end{aligned}$$

Base Case for $n = 0$ Similar to the other base case A_0 is also an expression that has no quantifiers over states, then:

$$\begin{aligned}|\mathbf{rSpec}(A_0, E)| &= |A_0| \\ &= 0 + |A_0| \\ &= 0(|b_{\max}| + 2) + |A_0|\end{aligned}$$

Induction Step for $n > 1$ We proceed by induction on m. Without loss of generality, we can assume that each quantifier over states scopes over the whole remaining assertion of A, since every assertion can be written with all the quantifiers at the top:

$$\begin{aligned}
|\mathbf{rSpec}(A_{m+1}, E)| &\leq |\mathbf{rSpec}(\mathcal{Q}\langle s\rangle.A_m, E)| \\
&\leq |\mathbf{rDSL}(A_m, E, E, \mathcal{Q})| \\
&\leq n|\mathbf{rSpec}(b_{\max}(\Rightarrow \text{or}\wedge)A_m, E)| + n \\
&\leq n(|\mathbf{rSpec}(b_{\max}, E)(\Rightarrow \text{or}\wedge)\mathbf{rSpec}(A_m, E)|) + n \\
&\leq n(|\mathbf{rSpec}(b_{\max}, E)| + |\mathbf{rSpec}(A_m, E)| + 1) + n \\
&\leq n(|b_{\max}| + |\mathbf{rSpec}(A_m, E)| + 2) \\
&\leq n(|b_{\max}| + 2 + (|b_{\max}| + 2)\frac{n^{m+1} - n}{n-1} + |A|n^m) \\
&\leq n(|b_{\max}| + 2) + (|b_{\max}| + 2)\frac{n^{m+2} - n^2}{n-1} + |A|n^{m+1} \\
&\leq (|b_{\max}| + 2)(n + \frac{n^{m+2} - n^2}{n-1}) + |A|n^{m+1} \\
&\leq (|b_{\max}| + 2)(\frac{n^{m+2} - n}{n-1}) + |A|n^{m+1}
\end{aligned}$$

For the second step, we use the previously proven fact that $|\mathbf{rDSL}(A, E, E, \mathcal{Q})| \leq |\mathbf{rDSL}(A, E', E', \mathcal{Q})|$ together with the size of $\mathbf{rDSL}(A, E', E', \mathcal{Q})$. For the fifth step, we utilize the knowledge that a predicate cannot have any quantifiers over a state; therefore, we know that **rSpec** returns the predicate on its own. Finally, in the sixth step, we apply our induction hypothesis for $n > 1$ and then solve the inequality.

Induction Step for $n = 1$ To prove this induction step for $n = 1$, we use the same fact as before that $|\mathbf{rDSL}(A, E, E), \mathcal{Q}| \leq |\mathbf{rDSL}(A, E', E', \mathcal{Q})|$ holds.

$$\begin{aligned}
|\mathbf{rSpec}(A_{m+1}, E)| &\leq |\mathbf{rSpec}(\mathcal{Q}\langle s\rangle.A_m, E)| \\
&\leq |\mathbf{rDSL}(A_m, E, E, \mathcal{Q})| \\
&\leq 1|\mathbf{rSpec}(b_{\max}(\Rightarrow \text{or}\wedge)A_m, E)| + 1 \\
&\leq (|\mathbf{rSpec}(b_{\max}, E)(\Rightarrow \text{or}\wedge)\mathbf{rSpec}(A_m, E)|) + 1 \\
&\leq |\mathbf{rSpec}(b_{\max}, E)| + |\mathbf{rSpec}(A_m, E)| + 2 \\
&\leq |b_{\max}| + 2 + m(|b_{\max}| + 2) + |A| \\
&\leq (m+1)(|b_{\max}| + 2) + |A|
\end{aligned}$$

The proof for $n = 1$ uses the same steps as for the other case, with the exception that we can use the fact that $n = 1$ in the second step to simplify the proof.

Conclusion

To be correct at this point, we would have to add the possibility of top-level operators back in. However, we observed that they only change the size of the assertion linearly once; therefore, we disregard them.

Since we often do not really use that many projections, $|b_{\max}| \ll |A|$ is usually the case, and most hyperproperties only use at most four different states, we can conclude that the size of the generated assertion is often in practice $\leq \mathcal{O}(|E|n^4)$.

Chapter 3

Loops

Loops often pose a significant challenge when dealing with the formal verification of programs. In this regard, Hypra [5] is no exception. Hypra verifies loops by creating a new Viper method for each loop. For this, it uses one of four different loop rules specified in Figure 8 in Dardinier and Müller [2]. All the invariants specified by the user are used as preconditions and postconditions for this method.

A key challenge with using labels in a loop is that their information must be encoded and passed into each iteration. This is because a loop only preserves its invariant by default. We must explicitly provide the label's information to the loop so that we can then infer meaningful properties from the apply statement within the loop's body.

This leads us to two different problems to solve. First, we need a way to use an apply statement on the union of a method call defined before the loop, and a method call in the current loop iteration. For this, we need to have enough information about how those two calls are related to each other. And secondly, we need a way to use an apply statement on the union of a method call in the current loop iteration and a method call in a previous one. To solve this, we have to find solutions to the following questions. What happens when we want to apply a method specification to two method calls that occur in different loop iterations? How do we reference the specific iteration we mean, and how do we preserve enough information about the previous sets of states from the earlier method calls to prove something meaningful?

3.1 Using Labels Declared Outside the Loop

If we want to use a label that references a method call declared outside the loop, we need a way to reference the set of states $S_{@\text{label}}$ occurring at the method call, since it is not declared inside the translated Viper method. To

do that, we pass $S_{@\text{label}}$ as a method argument into the Viper method. This argument lets us use the label inside the loop. However, we do not gain the ability to prove anything we have not been able to prove before, since all the information of how $S_{@\text{label}}$ is related to our current set of states S is lost. To address this, we have to consider the following points:

- To apply a method specification on multiple method calls, we often have to establish a property about the set of states at each call to hold.
- We need to know how the different sets of states of the method calls are related to each other to draw meaningful conclusions from the apply statement.

3.1.1 Referencing Labels in Invariants

To apply a method specification on multiple method calls, some precondition often needs to hold on the sets of states before the method calls.

```
method f(x : Int) returns (y : Int)
 one: requires ∀⟨σ⟩. σ(x) = 1
 one: ensures ∀⟨σ⟩. σ(y) = 1
method main(u : Int, n : Int) returns (v : Int, w : Int)
 requires ∀⟨σ⟩. σ(u) = 1
 ensures ∀⟨σ⟩. σ(w) = 1
{
 v := f(u) @entry
 apply one on @entry // needed for the second invariant on loop entry
 w := v
 while(n > 0)
  invariant ∀⟨σ@entry⟩. σ(u) = 1
  invariant ∀⟨σ1@entry'⟩, ⟨σ2⟩. σ1(v) = σ2(v)
  invariant ∀⟨σ⟩. σ(w) = 1
 {
  w := v
  apply one on @entry
  n := n − 1
 }
}
```

Figure 3.1: In this example we need the invariant so that we can apply the one specification.

However, since we verify the loop in a separate method, we cannot preserve information about the program unless we explicitly state the information we want to maintain in the invariant. This means that if we want to preserve information about the sets of states $S_{@\text{call}}$ and $S'_{@\text{call}}$ that occur before and

after a method call that has happened before the loop, we need to be able to write this information into the invariant. To solve this, we added the ability to reference those sets of states when writing an invariant directly with the notation $\langle \sigma$**@call**$\rangle$ and $\langle \sigma$**@call'**$\rangle$, meaning that the state σ is part of $S_{@\text{call}}$ and $S'_{@\text{call}}$, respectively.

For example, in Figure 3.1, without additional information about the initial method call, the precondition of the apply specification would not hold. This apply statement combined with the invariant that states v remains the same at the start of each loop iteration as after the method call, lets us conclude that $w = 1$. Although, this example can be proven differently, it already demonstrates the value of being able to write references to method calls in invariants.

3.1.2 Weak Framing for Labels Used Outside the Loop

Hypra automatically encodes all basic statements in a way that links the sets of states S before each statement to the set of states S' after the statement. The encoding of such a statement has the form of:

$$\forall \langle \sigma_1 @\text{S} \rangle . \exists \langle \sigma_2 @\text{S}' \rangle . \, E$$

Here, σ_1@S represents a state in the set of states S before the statement, and the same for σ_2 and @S', while E is an expression encoding the statement behavior. This means that between most of the encoded statements, we have a way to relate them to each other, making it possible to preserve facts about the current set of states generated previously over the whole program. One of those crucial facts is that certain states have to exist if other states have existed. For example, for a variable assignment, we know that if there existed a state before the assignment, then there also has to exist a state after. Going forward, we will use the same approach for the weak framing of loops.

Before this thesis, when referencing a label from before the loop, we lacked the information that for all the states in the loop, a state exists at the method call before the loop. This makes applying a specification to the union of a label before the loop and a label inside the loop yield no useful information, because we cannot connect the set of states from those two method calls. To understand this problem, it is helpful to look at an example:

In Figure 3.2, we have the two invariants:

$$\forall \langle \sigma_1 @\text{entry}' \rangle, \langle \sigma_2 \rangle . \, \sigma_1(u) = \sigma_2(u)$$
$$\forall \langle \sigma_1 @\text{entry}' \rangle, \langle \sigma_2 \rangle . \, \sigma_1(v) = \sigma_2(w)$$

The idea behind them is that together with the apply statement, they prove the final invariant and thus prove the specification of `main`. Then we have the apply statement that gives us:

```
method f(x : Int) returns (y : Int)
 id: ensures ∀⟨σ⟩. σ(x) = σ(y)
method main(u : Int, i : Int) returns (v : Int, w : Int)
 requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(u) = σ2(u)
 ensures ∀⟨σ⟩. σ(v) = σ(w)
{
 v := f(u) @entry
 apply id on @entry // needed for the second invariant on loop entry
 w := v // also needed for the second invariant to hold on entry
 while(i > 0)
  invariant ∀⟨σ1@entry'⟩, ⟨σ2⟩. σ1(u) = σ2(u)
  invariant ∀⟨σ1@entry'⟩, ⟨σ2⟩. σ1(v) = σ2(w)
  invariant ∀⟨σ⟩. σ(v) = σ(w)
 {
  // Weak framing in the form of ∀⟨σ1⟩. ∃⟨σ2@entry⟩. E
  w := f(u) @loop
  apply id on @entry ∪ @loop
  i := i − 1
 }
}
```

Figure 3.2: This example illustrates the need for Hypra to know that the call @entry existed for the last invariant to hold.

$$(\forall\langle\sigma@\text{loop}'\rangle.\, \sigma(u) = \sigma(w)) \land (\forall\langle\sigma@\text{entry}'\rangle.\, \sigma(u) = \sigma(v))$$

By combining the invariants with the apply statement, we get that the following needs to hold at the end of one loop iteration:

$$\forall\langle\sigma_1@\text{entry}'\rangle, \langle\sigma_2@\text{loop}'\rangle.\, \sigma_1(v) = \sigma_2(w)$$
$$\forall\langle\sigma_1@\text{entry}'\rangle, \langle\sigma_2@\text{loop}'\rangle.\, \sigma_1(v) = \sigma_2(v)$$

However, from this, we cannot prove that:

$$\forall\langle\sigma@\text{loop}'\rangle.\, \sigma(v) = \sigma(w)$$

Holds since we do not know that at least one state σ_1@entry′ existed. To solve this issue, we introduce an automatically generated weak framing in the form of $\forall\langle\sigma\rangle.\exists\langle\sigma@\text{call}'\rangle.E$. With this, we can prove our last invariant. We will come back to what E is in the next section. Intuitively, introducing such a statement makes sense since applying a method specification to a method call that did not occur should not yield any additional information about a different method call.

3.1.3 Strong Framing for Labels used Before a Loop

By applying specifications to a method call outside the loop in combination with a method call from inside the loop, we often obtain a property that relates the variables in the states after each method call to each other. However, we do not know how the variables in the states after the method call relate to the variables in the states at the start of the loop iteration. Therefore, all properties requiring such variable information are quite bothersome to prove. To do so, one must write all the properties about how variables have changed within loop invariant.

To obtain this information automatically, we need to identify which variables remain unchanged from the label outside the loop to the start of a loop iteration. We can then add this information at the beginning of the loop by framing the variables that did not change from the method call to the loop, and inside the loop. Hypra already automatically frames (4.3 of Dardinier *et al.* [5]) variables that did not change for each method call.

method f(x : **Int**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(y)$
method main(u : **Int**, i : **Int**) **returns** (v : **Int**, w : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(u) = \sigma_2(u)$
 ensures $\forall\langle\sigma\rangle.\ \sigma(v) = \sigma(w)$
{
 $v := \mathrm{f}(u)$ @entry
 $w := v$ // needed for the second invariant to hold on entry
 while($i > 0$)
  invariant $\forall\langle\sigma_1@\text{entry}\rangle, \langle\sigma_2@\text{entry}\rangle.\ \sigma_1(u) = \sigma_2(u)$
  invariant $\forall\langle\sigma\rangle.\ \sigma(v) = \sigma(w)$
 {
  $w := \mathrm{f}(u)$ @loop
  apply det **on** @entry $\cup$ @loop
  $i := i - 1$
 }
}

Figure 3.3: In this example we want to show that applying a deterministic method f any amount of times to the same input results in the same output, it is an extension of our initial Figure 1.3.

In Figure 3.3, we see an example of a proof that requires the variable v to be framed. This is the case because from the apply statement, we obtain the property $\forall\langle\sigma_1@\text{entry}'\rangle, \langle\sigma_2@\text{loop}'\rangle.\sigma_1(v) = \sigma_2(w)$. However, to get from this property to the second loop invariant, we need to know that v has not changed between @entry and @loop. On a side note, because we have strong

framing between @entry and the loop, we also do not need to write the invariants $\forall\langle\sigma_1@\text{entry}\rangle, \langle\sigma_2\rangle.\sigma_1(u) = \sigma_2(u)$ and $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\sigma_1(u) = \sigma_2(u)$, as they can be inferred from our first invariant in combination with strong framing.

The strong framing works by collecting two sets of variables that might have changed. The first consists of the variables that may have changed between the method call and the start of the loop, while the second set includes variables that may have changed within a loop iteration. Then, at the beginning of the loop, we add a statement in the generated Viper file stating that either this is the first iteration, in which case we only frame the first set of variables, or it is the nth iteration, in which case we frame over both sets. We call this way of framing the variables disjunctive framing, and why we make this distinction between variables will become clear in Section 3.2.3.

3.2 Using Labels Declared in a Different Loop Iteration

To use labels declared in a different loop iteration, we create a new set of states $S^*_{@\text{loop}}$ for each method call with a label. This set is the union of all the previous sets of states $S_{@\text{loop}}$ in earlier loop iterations before or after a method call (with the label @loop) defined in the loop. We call the labels referencing this set of states accumulative labels.

Then, for this new set of states, we take the same approach as earlier for the set of states $S_{@\text{call}}$, meaning that we can reference this new set of states in invariants. We are also automatically generating the correct framing, and we can reference this set of states in the apply statement.

3.2.1 Referencing Accumulative Labels in Invariants

To add this new ability to reference the new accumulative labels, we had to slightly change the implementation of labels so that whenever a label is encountered, not only is the current set of states saved in a unique variable, but also the accumulative set of states of this label gets updated to the union of the accumulative set and the new set.

3.2.2 Weak Framing for Accumulative Labels

Here, we need to be careful, as not every loop iteration has a previous loop iteration; therefore, we cannot simply implement the same weak framing as for the labels outside a loop. However, having a forall-exists property stating that a previous method call has occurred is still relevant to prove properties that correlate method calls in previous loop iterations with method calls in the current loop iteration. For the same reason as stated in Section 3.1.2.

Therefore, we implemented a weaker version of the same clause, stating that either we had a previous method call via weak framing, or the set of states referenced by the accumulative label is empty. An example of this can be seen in Figure 3.4.

3.2.3 Disjunctive Framing for All Labels

For framing the variables of the new set of states $S^*_{@\text{loop}}$, we collect the set of variables that might have changed between the method call and the end of the loop. Then, at the start of the loop, we added two options:

- **Outside Framing:** For all labels outside the loop, we frame over the statements leading up to the loop, and we set the sets of states referenced by accumulative labels to empty.
- **Inside Framing:** For all labels outside the loop, we frame over all statements leading up to the loop and all statements in the loop. For the accumulative labels within the loop, we frame the variables that have not changed from the label declaration to the end of the loop.

We call this way of framing variables disjunctive framing; an example of it can be found in Figure 3.4. However, in this example, the variables x and i are also included in the final framing just not shown. Finally, with this implemented, we can prove properties about all method calls that happened in previous iterations.

```
method f(x : Int) returns (y : Int)
 det: requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(x) = σ2(x)
 det: ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(y) = σ2(y)
method main(x : Int, i : Int) returns (b : Int, c : Int)
 requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(x) = σ2(x)
 ensures ∀⟨σ⟩. σ(b) = σ(c)
{
 b := f(x) @entry
 c := b
 while(i > 0)
  invariant ∀⟨σ1@entry⟩, ⟨σ2@entry⟩. σ(x) = σ(x)
  invariant ∀⟨σ1@loop*⟩, ⟨σ2@loop*⟩. σ(x) = σ(x)
  invariant ∀⟨σ⟩. σ(b) = σ(c)
 {
  // The following is a part of the framing provided.
  // if (*) (∀⟨σ1⟩.∃⟨σ2@entry'⟩. σ1(b) = σ2(b) ∧ σ1(c) = σ2(c)) ∧ (¬∃⟨σ@loop*⟩)
  // else ∀⟨σ1⟩.∃⟨σ2@loop*'⟩. σ1(b) = σ2(b) ∧ σ1(c) = σ2(c)
  b := f(x) @loop
  apply det on @entry ∪ @loop
  apply det on @loop* ∪ @loop
  i := i − 1
 }
}
```

Figure 3.4: This example is the same as in Figure 3.3, with the exception that we have one more assignment to c before the method call @loop making our disjunctive framing necessary.

To understand why we came up with disjunctive framing, we look at the example in Figure 3.4. In this example, we have an assignment to the variable b inside the loop, meaning that we cannot make the strong assumption that:

$$\forall\langle\sigma_1\rangle.\exists\langle\sigma_2@\text{entry}'\rangle.\, \sigma_1(b) = \sigma_2(b)$$

Holds in all cases. For all the assertions provided, $\langle\sigma_1\rangle$ references a state at the start of a loop iteration. This assumption only holds at the beginning of the first iteration of the loop and is equivalent to what our strong outside framing provides. Now, for the other case, where we are in the i-th loop iteration, we need another way to generate that b is the same at the start of the loop iteration and after the method call @entry. This is where the inside framing comes in. It provides us with:

$$\forall\langle\sigma_1\rangle.\exists\langle\sigma_2@\text{loop*}'\rangle.\, \sigma_1(b) = \sigma_2(b)$$

With both framing possibilities and apply statements, we get that b has the equivalent value as in a previous iteration or as before the loop. Combining this with our invariant and the framing of c, we get that c must be the same as b in all states.

3.3 Rules for Using Labels Inside Loops

In the following two sections, we will go over the rules implemented in Hypra to enforce correctness when using our new apply statements.

3.3.1 Basic Rules

The basic rules are as follows:

- A label has to be declared before it is used.
- A label can be used in the following program structures:
  - apply statements
  - invariants
  - hyperasserts/hyperassumes

Those rules follow from the fact that we cannot reference a label that was not declared. Additionally, as previously explained, we can use labels in invariants and apply statements. We also added the ability to use them in hyperassertions.

3.3.2 Advanced Rules

The introduction of the weak framing also introduces a forall exists property in the form of $\forall\langle\sigma_1\rangle.\exists\langle\sigma_2@\text{label}\rangle.\ E$. This essentially means that if there exists at least one state inside the loop, there has to be at least one state at each label before the loop. The same holds for accumulative labels if we are in the $n{+}1$-th iteration.

However, this assumption might not always hold; consider Figure 3.5, since the method call might not be on the main path leading up to the loop. This means our clause introduces unsoundness in states where the method call was not executed. Therefore, we had to come up with rules disallowing the unsound use of labels.

The idea behind the rules is that we introduce unsoundness only when we have a weak framing of a method call that might not be executed in every possible execution of a program, which means that those are the only cases that we need to exclude.

```
method setFalse() returns (b : Bool)
 returnsFalse: ensures ∀⟨σ⟩. ¬σ(b)
method main() returns (b : Bool)
 ensures false
{
 var i: Int
 i := 1
 b := true
 if (false) b := setFalse() @entry
 while(i > 0)
  invariant ∀⟨σ⟩. σ(b)
  invariant ∀⟨σ⟩. σ(i) ≠ 1 ⇒ ¬σ(b)
 {
  apply returnsFalse on @entry
  i := i − 1
 }
}
```

Figure 3.5: A Simple example showcasing why we need to be careful when we allow the use of labels inside loops. From the apply statement we get that the variable b after the method call setFalse() is false. Then from your framing we get that the variable b has to be the same at the start of the loop as at the label. And finally from our forall-exists clause we get that there has to be a state that called the method. When combining the three properties we get that the value of b inside the loop has to be false. However, this is clearly not the case since b was never set to false in any state.

To determine if a label can be used, we calculate the least encompassing scope (LES) from the scope in which the label was declared and the scope in which the label was used. This LES scope is the smallest scope that encompasses both the declaration and the use of the label. Then, we calculate whether the use/declaration of a label occurred within the LES itself, in a smaller scope that is not in a loop, or in a loop. Then we apply the following table to the result to see if the use is allowed:

Table 3.1: Label Rules

Declaration\ Use	In LES	No Loop	Loop
In LES	✓	✓	✓
No Loop	✓	✓	✗
Loop	✗	✗	✗

For the first row being in the LES indicates that the declaration is on the main path, as there cannot be any branches without changing the scope.

For the second row, the first two columns are cases where there are no loops, and therefore, we do not have the forall-exists clause that might introduce

unsoundness. Then, the case where the declaration is in No Loop and the Use is in Loop is precisely the case of Figure 3.5, where the declaration is not on the main path to the loop, since it is not in the LES.

And finally, we have the last row of the table. These are the cases where the declaration is in the loop itself. Using a label declared in a previous loop outside it does not make sense, as we cannot even specify which iteration of the loop we mean. To apply this table, we have to think about some special cases:

- **Labels used in Invariants:** are calculated as if they were statements within the scope of the loop they are part of. This is the case because using an invariant with a reference to a label already utilizes the loop framing.
- **Accumulative Labels:** they are calculated as if they were declared outside the loop. This makes sense because whenever we use an accumulative label, we do that to reference one or multiple method calls that have been declared in a previous loop iteration, much like when we use a label declared before the loop. Therefore, we also have a forall-exists clause, similar to using a label declared outside the loop, meaning that we need to apply the same restriction as in that case. This means that if we have a label declaration inside the loop that is not on the main path of the loop, we disallow the use of union labels in the invariants. This also has the effect that, in theory, we can use a union label that was declared inside a loop outside the loop. However, while this behavior does not create any unsoundness, we do not know if there are any uses for it.

From this table, we also see that there are no limitations on using labels (except that they have to be declared first) in methods that do not use any loops. This makes sense since we do not introduce anything that might be unsound in those cases. We can use this approach to prove some interesting properties. For example, if we have two method calls in different if statements, we can still prove specific properties about them.

3.3.3 Limitations

Suppose we have a method call that is defined within a loop; the only way to reference it after the loop is by using an accumulative label. In this case, it is by design, as there is no way for us to know which method call to reference in a loop that has been executed multiple times. However, even when using an accumulative label, we currently cannot prove anything by using an apply statement later, as we miss the forall-exists clause and the framing when leaving a loop for the labels defined within. We can show this by the following example in Figure 3.6.

```
method f(x : Int) returns (y : Int)
 det: requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(x) = σ2(x)
 det: ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(y) = σ2(y)
method f(a : Int) returns (b : Int, c : Int)
 requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(a) = σ2(a)
 ensures ∀⟨σ⟩. σ(b) = σ(c)
{
 var i : Int
 i := 0
 while(i < 10)
  invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(i) = σ2(i) ∧ σ1(a) = σ2(a)
  invariant ∀⟨σ1@loop*⟩, ⟨σ2⟩. σ1(a) = σ2(a)
  invariant ∀⟨σ1@loop*⟩, ⟨σ2@loop*⟩. σ1(a) = σ2(a)
  invariant ∀⟨σ1@loop*'⟩, ⟨σ2⟩. σ2(i) > 0 ⇒ σ1(b) = σ2(b)
 {
  b :=f(a) @loop
  apply det on @loop*
  i := i + 1
 }
 c :=f(a) @exit
 apply det on @exit ∪ @loop*
}
```

Figure 3.6: The program provided currently cannot be proven by Hypra because we do not have the framing when leaving a loop, making our last apply statement fail.

Chapter 4

Evaluation

Here, we present some examples that demonstrate the extended abilities of Hypra that we achieved with the completion of this thesis. There are two types of proofs that we can construct with our new features. We can either directly write our statements as pre- and postconditions of the program we want to prove, or we can construct a new program that simulates the behavior we want to prove in some way, and then prove our statements on this program.

4.1 Hyperproperties

In this section, we will show some examples of programs for which we can directly prove properties by writing them as pre- and postconditions.

4.1.1 Monotonicity and Determinism with Fibonacci

Hypra enables the verification of certain hyperproperties in programs that contain loops. However, with this thesis, we can now prove that multiple different hyperproperties hold over the same method. We demonstrated this in the example in Figure 4.1 in which we prove that the two different hyperproperties determinism and monotonicity hold for a Fibonacci method. For this example, Dardinier and Müller [2] initially demonstrated that with the use of HHL, we can prove that Fibonacci is a monotonic function. This proof had to be done by hand or with Hypra by using an extra program variable; however, with the addition of logical variables, we can prove it without using a program variable to tag the sets of states. Therefore, it does not impact the program's behavior.

In Figure 4.1, we have two named specifications, one for monotonicity and one for determinism. For the determinism specification, writing the invariant is straightforward; we state that in each iteration, all variables have the same

```
method fib(n : Int) logical (t : Int) returns (b : Int)
 det: requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n)
 det: requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(b) = σ2(b)
 mono: requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(t) = 1 ∧ σ2(t) = 2 ⇒ σ1(n) ≤ σ2(n)
 mono: ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(t) = 1 ∧ σ2(t) = 2 ⇒ σ1(b) ≤ σ2(b)
{
 var a : Int
 a := 0
 b := 1
 while (n > 0)
  invariant ∀⟨σ⟩.σ(b) ≥ σ(a) ∧ σ(a) ≥ 0
  det: invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n) ∧ σ1(a) = σ2(a) ∧ σ1(b) = σ2(b)
  mono: invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(t) = 1 ∧ σ2(t) = 2
    ⇒ σ1(n) ≤ σ2(n) ∧ σ1(a) ≤ σ2(a) ∧ σ1(b) ≤ σ2(b)
 {
  var tmp : Int
  tmp := b
  b := a + b
  a := b
  n := n − 1
 }
}
```

Figure 4.1: This is a simple program computing the n-th fibonacci number for which we can prove that it is deterministic and monotonic. It uses a named invariant for each of the named specifications and uses logical variables for the monotonicity property. For example, we can later use this method to prove that the main method holds in Figure 2.8 if f is the fibonacci method and u is greater or equal to 0.

value for each state, which Hypra can verify because all the statements are deterministic.

Then, for the monotonicity specification, it gets more complicated. We can first write the `mono` invariant, which states that the values of n, a, and b are always greater in all states where the logical variable t has a value of 2 compared to the states where the logical variable has a value of 1. Since this specification has a precondition that ensures this fact upon entry, and the value of n is decreased by 1 in each iteration, this holds for n. However, for the other two variables, a and b, proving this invariant becomes more complicated. For this `mono` invariant to hold for these two variables, we must use an additional invariant that states the value of b is always greater than the value of a, and a is greater than or equal to 0. We need this invariant because, together with the statements of the loop, it implies that a and b increase in every iteration. This then allows us to prove the `mono` invariant.

In the `mono` invariant it is essential to note that we cannot simply use the `det`

specification for the values of a and b, as in the case where the value of n differs across all states, the loop may have a different number of executions for each state, which forces us to use a different loop rule for this case that assumes that the invariant also holds if the loop is not entered. To read more about loop rules, see section 4.1 in Dardinier *et al.* [5].

This example demonstrates that in our implementation, Hypra automatically detects and applies the correct loop rule for each specification, as well as that we can utilize logical variables in certain specifications without affecting all specifications. It also demonstrates the need to have named invariants.

4.1.2 Non-Deterministic Associative Fold

method op(x : **Int**, y : **Int**) **logical** (t : **Int**) **returns** (z : **Int**)
requires $\exists\langle\sigma\rangle.\ \text{true}$
ensures $\exists\langle\sigma\rangle.\ \text{true}$
det: **requires** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x) \wedge \sigma_1(y) = \sigma_2(y)$
det: **ensures** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(z) = \sigma_2(z)$
com: **requires** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(a) = \sigma_2(b) \wedge \sigma_1(b) = \sigma_2(a)$
com: **ensures** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(r) = \sigma_2(r)$
asso: **requires** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle,\langle\sigma_3\rangle,\langle\sigma_4\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \wedge \sigma_3(t) = 3 \wedge \sigma_4(t) = 4$
$\Rightarrow \sigma_1(y) = \sigma_2(x) \wedge \sigma_2(y) = \sigma_3(y) \wedge \sigma_1(x) = \sigma_4(x)$
asso: **ensures** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle,\langle\sigma_3\rangle,\langle\sigma_4\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \wedge \sigma_3(t) = 3 \wedge \sigma_4(t) = 4$
$\Rightarrow (\sigma_3(x) = \sigma_1(z) \wedge \sigma_4(y) = \sigma_2(z) \Rightarrow \sigma_3(z) = \sigma_4(z))$
method fold(a : **Int**, b : **Int**, c : **Int**, r : **Int**) **returns** (d : **Int**)
requires $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(a) = \sigma_2(a) \wedge \sigma_1(b) = \sigma_2(b) \wedge \sigma_1(c) = \sigma_2(c)$
requires $\forall\langle\sigma\rangle.\ \sigma(r) \geq 1 \wedge \sigma(r) \leq 3$
requires $\exists\langle\sigma_1\rangle,\langle\sigma_2\rangle,\langle\sigma_3\rangle.\ \sigma_1(r) = 1 \wedge \sigma_1(r) = 2 \wedge \sigma_1(r) = 3$
ensures $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\ \sigma_1(d) = \sigma_2(d)$
{
var tmp : **Int**
if $(r = 1)$ { $//d = ((a * b) * c)$
$tmp := \text{op}(a, b)$ @first$_1$
$d := \text{op}(tmp, c)$ @second$_1$
} **if** $(r = 2)$ { $//d = ((b * a) * c)$
$tmp := \text{op}(b, a)$ @first$_2$
$d := \text{op}(tmp, c)$ @second$_2$
} **if** $(r = 3)$ { $//d = (b * (a * c))$
$tmp := \text{op}(a, c)$ @first$_3$
$d := \text{op}(b, tmp)$ @second$_3$
}
//Proof Statements
apply com **on** @first$_1(t \to 1) \cup$ @first$_2(t \to 2)$
apply det **on** @second$_1 \cup$ @second$_2$
apply asso **on** @first$_2(t \to 1) \cup$ @first$_3(t \to 2) \cup$ @second$_2(t \to 3) \cup$
@second$_3(t \to 4)$
}

Figure 4.2: This example shows that a fold method applied in three different ways on an associative, commutative, deterministic method f is deterministic.

Consider a fold method that dynamically determines its reduction order based on a heuristic function. This approach to a fold method is typically used to speed up the method (for example, to parallelize it). For example, suppose we have a list of matrices and want to multiply them together. In this

case, we can speed up the process by considering the order of multiplication strategically.

In the example in Figure 4.2, we have a fold method that takes four arguments as input, and depending on the value of r, we apply the method `op` differently on the other three arguments. For this example, the method `op` is associative, commutative, and deterministic. We verified the correctness of the associative specification against Sousa and Dillig [8] Figure 3.

What we want to prove is that the fold method is deterministic, meaning that if the input across all states is the same, then the output of the fold method is the same. In this example, the input argument r could be non-deterministic. Still, the precondition of `fold` provides two guarantees for it: first, that the resulting value of r is always between 1 and 3, and second, that each value of r is executed in at least one state. The first guarantee is necessary so that we know at least one if statement is always executed. As for the second one, we will revisit this once we understand the apply statements.

To understand this example, we need to understand the apply statements. The first apply statement states that the result of @first$_1$ and @first$_2$ is equal using the commutative specification. The second apply statement states that the result of @second$_1$ and @second$_2$ is equal using the determinism specification together with the fact established from the first apply statement. With those two apply statements, we have proven that the value of r is the same for the first two ways to apply the method. Then the third apply statement uses the associative property of `op` to prove that the value of r is the same for the last two ways to apply op.

Now that we have proven that:

$$\forall\langle\sigma_1\text{@second}'_1\rangle, \langle\sigma_2\text{@second}'_2\rangle.\, \sigma_1(r) = \sigma_2(r)$$
$$\forall\langle\sigma_1\text{@second}'_2\rangle, \langle\sigma_2\text{@second}'_3\rangle.\, \sigma_1(r) = \sigma_2(r)$$

It should follow that:

$$\forall\langle\sigma_1\text{@second}'_1\rangle, \langle\sigma_2\text{@second}'_3\rangle.\, \sigma_1(r) = \sigma_2(r)$$

However, this is only the case since we know of the existence of at least one state that has executed the method call @second$_2$, and that is precisely the reason why we need the exists property for the heuristic method.

This example shows two limitations that still exist within Hypra. The first one is that we should still be able to derive properties from method calls that have not been executed. The second limitation is that, with the third apply statement, we have created a significant load for the SMT solver. This is the case since this property has four states it quantifies over, and we also apply it to four different method calls. This means that we have created an

assertion of the size $\leq |E|4^4 = 256|E|$, which is enormous (was confirmed by checking the generated Viper file). However, in theory, we only need an assertion that has precisely the size of the pre- and postconditions of the associative specification, meaning that in this area, the algorithm can still be improved. We can also observe this behavior in the example translation for logical variable assignment 2.4.1, where we generate all four possible combinations of logical variables. Still, only one is useful for our proof.

4.1.3 Deterministic Idempotent Method Applied N-Times

method abs(x : **Int**) **returns** (y : **Int**)
 det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(x) = \sigma_2(x)$
 det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(y)$
 idem: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(y) = \sigma_2(x) \Rightarrow \sigma_1(y) = \sigma_2(y)$
{
 if ($x < 0$) { $y := -x$ }
 else { $y := x$ }
}
method main(u : **Int**, i : **Int**) **returns** (v : **Int**)
 requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(u) = \sigma_2(u)$
 ensures $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(v) = \sigma_2(v)$
{
 $v :=$ abs(u) @entry
 apply det **on** @entry
 while($i > 0$)
  invariant $\forall\langle\sigma_1\rangle, \langle\sigma_2$@entry$'\rangle.\ \sigma_1(v) = \sigma_2(v)$
 {
  $v :=$ abs(v) @loop
  apply idem **on** @entry $\cup$ @loop
  $i := i - 1$
 }
}

Figure 4.3: In this example we prove for the method `main` that applying a deterministic idempotent method any amount of times to its output is the same as applying the method once. To prove this fact we need the loop invariant stating that v is the same in any state in the loop as well as after the first call of `f`.

For our following example (Figure 4.3), we aim to demonstrate that applying a deterministic idempotent method `abs` multiple times to its output yields the same result as applying it once. This specification represents a basic compiler optimization that can be applied to loops for idempotent statements. This example is inspired by D'Osualdo *et al.* [9] section 2.5.

To prove this property, we write a method that applies a deterministic

idempotent method $i + 1$ times on its output. Then, for this method, we write a simple determinism pre- and postcondition that only depends on the first input of `abs` and the final result. This means that if this method specification holds, we can prove that the output has nothing to do with i, and therefore the output does not depend on how many times `abs` has been called.

To prove the postcondition, we first prove that the result of the first method call is deterministic, via the first apply statement. Now, we apply our idempotent specification to the union of the current method call and the first one in each loop iteration, which allows us to prove that after each of these method calls, the result is the same as after the first one. This then leads us to the invariant, which states that in each iteration of the loop, the value of v must remain the same as after the first call. Now that we know the invariant holds, we can automatically prove the postcondition. Since after the first method call, all states have the same value for v, and together with the invariant after the loop, we know that the postcondition holds.

As a final note, `abs` must be deterministic; otherwise, calling the method `main` could result in different return values, thereby invalidating our specification of `main`.

4.1.4 Applying a Monotonic Method on a Sorted List

method f(x : **Int**) **logical** (t : **Int**) **returns** (y : **Int**)
mono: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(x) \leq \sigma_2(x)$
mono: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(y) \leq \sigma_2(y)$
method main(a : **Seq[Int]**) **returns** (b : **Seq[Int]**)
requires $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a)$
requires $\forall\langle\sigma\rangle.\forall j, k : \textbf{Int}.\, j \geq 0 \land j \leq k \land k < |a| \Rightarrow \sigma(a[j]) \leq \sigma(a[k])$
ensures $\forall\langle\sigma\rangle.\forall j, k : \textbf{Int}.\, j \geq 0 \land j \leq k \land k < |b| \Rightarrow \sigma(b[j]) \leq \sigma(b[k])$
ensures $\forall\langle\sigma\rangle.\, |\sigma(a)| = |\sigma(b)|$
{
var i : **Int**
$i := 0$
var u : **Int**
var v : **Int**
$b :=$ **Seq[Int]**()
dummyCall() @entry
while($i < |a|$)
invariant $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(a) = \sigma_2(a) \land \sigma_1(i) = \sigma_2(i)$
invariant $\forall\langle\sigma\rangle.\, |\sigma(a)| \geq \sigma(i) \land |\sigma(b)| = \sigma(i)$
invariant $\forall\langle\sigma@\text{entry}'\rangle.\, \sigma(i) = 0$
invariant $\forall\langle\sigma_1@\text{loop}^*\rangle, \langle\sigma_2\rangle.\, \sigma_2(i) \leq |\sigma_2(a)| \Rightarrow \sigma_1(u) \leq \sigma_2(a[i])$
invariant $\forall\langle\sigma\rangle.\, \sigma(i) \neq 0 \Rightarrow \sigma(v) = \sigma(b[i-1])$
invariant $\forall\langle\sigma\rangle.\forall j, k : \textbf{Int}.\, j \geq 0 \land j \leq k \land k < |a| \Rightarrow \sigma(a[j]) \leq \sigma(a[k])$
invariant $\forall\langle\sigma\rangle.\forall j, k : \textbf{Int}.\, j \geq 0 \land j \leq k \land k < |b| \Rightarrow \sigma(b[j]) \leq \sigma(b[k])$
{
$u := a[i]$
$v :=$ f(u) @loop
$b := b \oplus$ **Seq[Int]**(v)
apply mono **on** @loop$^*(t \to 1) \cup$ @loop$(t \to 2)$
$i := i + 1$
}
}

Figure 4.4: A proof that a monotonic method applied to each element of a sorted list results in a new sorted list.

Consider the scenario where we have a sorted list and want to apply a monotonic method to each element of the list. If we do that, it logically follows that the new list is also sorted. However, proving this property is not straightforward, as it is essentially a composition of the monotonic property over multiple method calls within a loop. Hypra allows us to prove for certain methods that they are monotonic, as seen in Figure 4.1. With the extended capabilities to apply method specifications, we can now also prove

that applying a monotonic method on a sorted list returns another sorted list.

To achieve this, we constructed the Hypra program in Figure 4.4, in which we have a `main` method that takes as input a sequence of integers a and, for each element in a, applies the monotonic method `f` to it and appends the result to the resulting sequence b. In this program, we have three supporting variables: the loop counter i, the variable u, which is the i-th element of a, and v, which is the result of $\texttt{f}(u)$. To prove the property that we want, we need two different preconditions: one of the preconditions states that the list a is sorted in each state, while the other precondition says that all states have the same list a. We need this precondition so that for the states of previous iterations and the current iteration, we know that a should be the same. However, the precondition that states all states have the same value for a can be later relaxed by proving this property holds for each different input value, much like we will do in Figure 4.6.

For the postcondition, we have that the list b is sorted and that its length is the same as the length of a.

At the core of this proof is the apply statement, which applies the monotonic property on all previous method calls in conjunction with the current method call. For the precondition of this statement to hold, we must consider two different possibilities. The first case corresponds to the first loop iteration, where no method calls have occurred before. In this case, i is 0; we obtain this property from our invariant, which states that for all states after the `dummyCall()` at the loop's entry, $i = 0$. Together with the outside framing, where we say $S^*_{@\text{loop}}$ is empty and the value of i is the same as after the `dummyCall()`. This call is specially needed for framing, which ensures that the apply statement verifies in the first iteration. However, this call is only required for its label and the framing it provides. Having the ability to label any point in a program would eliminate this call.

The second case is where we are not in the first loop iteration; therefore, we had an earlier method call of `f`. In this case, for our precondition of the apply statement to hold, we need to know that the input of each of the previous method calls is smaller than the input of the current method call. Since the variable u is precisely the value that in each loop iteration was used as the input for our method call, we can express this property in our invariant that combines the previous method calls to the current loop iteration. This invariant does not hold after the last invariant since the value for $a[i]$ if $|a| = i$ is undefined.

Now that the precondition of the apply statement holds, we need to ensure that the property we obtain from it results in our postcondition. To do that, we added the invariant that connects v to $b[i-1]$. This invariant, together with the fact that every previous v is smaller than the current v, proves that

the last element in the list b is greater than all previous elements. And this, coupled with the invariant that states that b has been sorted so far, proves that b is still sorted.

We have now discussed all invariants except the second one, and indeed, we only need this invariant to prove that the length of b is the same as a. With all those invariants, it would be easy to add a determinism specification to our `main` method, since our apply statement implies determinism.

One issue with the specification of `main` is that it does not explicitly state that b is composed of results obtained by applying `f` to elements of a. This means that, in theory, we could append i to b in each iteration instead of the result, and all our invariants and method specifications would hold. This abstraction, combined with the specification, also allows for more modularity and easier proofs.

4.2 Relational Properties

In this section, we demonstrate the extended capabilities of Hypra by providing examples in which we construct product programs for which we prove properties with Hypra. Those properties correspond to relational properties between different programs.

4.2.1 Determinism from Commutativity

method op(a : **Int**, b : **Int**) **logical** (t : **Int**) **returns** (r : **Int**)
com: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(a) = \sigma_2(b) \land \sigma_1(b) = \sigma_2(a)$
com: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \land \sigma_2(t) = 2 \Rightarrow \sigma_1(r) = \sigma_2(r)$
method main(a : **Int**, b : **Int**) **returns** (r : **Int**)
det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(a) = \sigma_2(a) \land \sigma_1(b) = \sigma_2(b)$
det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(r) = \sigma_2(r)$
{
//Prog
$r := \text{op}(a, b)$ @call
//Helper Prog
var r_{helper} : **Int**
$r_{\text{helper}} := \text{op}(b, a)$ @helper
//Proof Statement
det: **apply** com **on** @call($t \to 1$) $\cup$ @helper($t \to 2$)
}

Figure 4.5: This is a proof that every commutative method (here op) is also deterministic.

With the addition of our apply statement and the union operator, we can

prove new hyperproperties from other hyperproperties. We show this in Figure 4.5, where we prove that a commutative method `op` is also deterministic.

We prove this simple statement by creating a new method that calls `op` and returns its result. For this new method, we use determinism as a specification; therefore, if we prove the specification of the main method, we prove that `op` is deterministic. However, applying our commutative specification of `op` to a single method call would not yield our desired result, since the specification combines different method calls with reversed inputs, and having only one method call in our apply statement would not yield any properties. Because of this, we add another method call with reversed arguments to apply our specification on. Finally, we can use the apply statement on both method calls to prove our determinism specification.

However, this new main method does not behave the same as our `op` method, because we now have two method calls, meaning that if the second one does not terminate, the behavior of the `main` method is not the same as `op`. This could verify our determinism property for cases where the second method call does not return. A proposed solution for this limitation is ghost calls.

4.2.2 Equivalence of Different Method Applications

method op(a : **Int**, b : **Int**) **logical** (t : **Int**) **returns** (r : **Int**)
det: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(a) = \sigma_2(a) \wedge \sigma_1(b) = \sigma_2(b)$
det: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(r) = \sigma_2(r)$
com: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(a) = \sigma_2(b) \wedge \sigma_1(b) = \sigma_2(a)$
com: **ensures** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(t) = 1 \wedge \sigma_2(t) = 2 \Rightarrow \sigma_1(r) = \sigma_2(r)$
method main_aux(a : **Int**, b : **Int**) **returns** (z_1 : **Int**, z_2 : **Int**)
eq_input: **requires** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(a) = \sigma_2(a) \wedge \sigma_1(b) = \sigma_2(b)$
eq_input: **ensure** $\forall\langle\sigma\rangle.\ \sigma(z_1) = \sigma(z_2)$
{
//Prog_1
var x_1 : **Int**
$x_1 := \text{op}(a, b)$ @call_{11}
$z_1 := \text{op}(x_1, x_1)$ @call_{12}
//Prog_2
var x_2 : **Int**
var y_2 : **Int**
$x_2 := \text{op}(a, b)$ @call_{21}
$y_2 := \text{op}(b, a)$ @call_{22}
$z_2 := \text{op}(x_2, y_2)$ @call_{23}
//Proof Statements
eq_input: **apply** det **on** @$\text{call}_{11} \cup$ @call_{21}
eq_input: **apply** com **on** @$\text{call}_{21}(t \to 1) \cup$ @$\text{call}_{22}(t \to 2)$
eq_input: **apply** det **on** @$\text{call}_{12} \cup$ @call_{23}
}
method main(a : **Int**, b : **Int**) **returns** (z_1 : **Int**, z_2 : **Int**)
ensure $\forall\langle\sigma\rangle.\ \sigma_1(z_1) = \sigma_2(z_2)$
{
$z_1, z_2 := \text{main\_aux}(a, b)$ @call
//Proof Statement
apply eq_input: **on** $\forall x, y$: **Int**. @call$[a = x \wedge b = y]$
}

Figure 4.6: This is a proof that for a deterministic commutative method op having the statements $x := \text{op}(a, b)$, $z := \text{op}(x, x)$ results in the same output as the statements $x := \text{op}(a, b)$, $y := \text{op}(b, a)$, $z := \text{op}(x, y)$. This is an example provided by D'Osualdo *et al.* [9].

D'Osualdo *et al.* [9] shows in Example 1.1 that supporting hyper-triple composition is challenging for many relational logics. The example states that a deterministic commutative method op applied in two different ways results in the same output. We implemented this example in Figure 4.6 with our newly added DSL and apply statement.

To prove this example, we created two new methods: main_aux and main. The method main_aux executes both programs sequentially, with distinct variables, but the same input variables. We then assumed, as a precondition, that for any two states, the input arguments are the same. We did this because having this stronger precondition simplifies applying our specifications, as we no longer need to worry about different states having different input values. Then, we created the postcondition that for any state, the output of both programs is the same, as we want to prove that the output of both ways to apply those methods is the same. Finally, we applied the method specifications of op three times:

- The first apply statement states that x_1 and x_2 must be the same using the determinism specification.
- The second apply statement states that x_2 and y_1 must be the same using the commutative specification.
- The last apply statement states that z_1 and z_2 must be the same, and with that, proving our postcondition for the main_aux method.

We have now proven that if all our states have the same input value for a and b, then the output value of both different programs must be the same. Now, with the main method, we extend this proof so that it no longer requires the stronger precondition. We achieve this by executing main_aux and then applying the specification on each different subset of states where a and b are the same.

This example shows two features that we enabled with our work. Firstly, the original complex example (from D'Osualdo *et al.* [9]) of composing multiple method calls has been solved for some programs using an automated program verification tool, Hypra [5]. Secondly, we have introduced a method for breaking down a proof into smaller steps. This involves first proving a specification on a stronger precondition and then applying this specification on multiple smaller subsets of states, thereby enabling the proof of a stronger specification. This is especially useful when dealing with loops; having the ability to write stronger preconditions often allows us to use a stronger loop rule, which then gives us a stronger postcondition.

Chapter 5

Conclusion and Future Work

With the conclusion of this thesis, we will review the original goals drafted in the Introduction to illustrate what we achieved, discuss some limitations, and outline how to address them.

5.1 Conclusion

Our initial goals for this thesis were the following:

- Create a new way to apply method specifications in a custom manner to a method call or multiple method calls. Allowing us to reason about properties that arise from the composition of specifications of multiple method calls.
- Extend this way to apply method specifications to method calls within loops.

To achieve this, we have proposed a solution in the form of an apply statement, which enables us to apply a specification in a complex manner to a method call or multiple method calls. This apply statement uses a new DSL that we created to solve the challenge of specifying how a specification should be applied. This DSL supports multiple new operations, each of which solves a unique challenge and can be used in combination with the others.

We then extended the capabilities of the apply statement to work with loops in two different ways: either by using labels that have been defined before the loop or by using labels defined within a different iteration of the loop. To address the challenge of losing information about the labels when entering loops, we developed three complementary solutions that collectively aid in proving properties about loops.

Finally, with the help of some examples, we demonstrated what our solutions can accomplish. For example, in Figure 4.4, we were able to prove that a

monotonic method applied to each element of a sorted list results in another sorted list.

5.2 Future Work

In this segment, we will explore possibilities for the future that extend this thesis to be more expressive, enabling us to verify more programs. Additionally, we explore some ideas that would generate more properties automatically.

5.2.1 Ghost Statements or Ghost Apply

In previous examples, we saw that Hypra is still limited by compositionality in cases where an additional method call is required (see Figure 4.5) to prove that a property holds. In those cases in the program, we needed to know that a method call was executed previously to prove that a property holds from the apply statement. For example, in Figure 4.2, we needed the exists property in the heuristic method h to be able to prove that all values for r are the same. To solve this issue, there should be a way to apply a specification on a method call that was not actually executed, yet still obtain a property from an apply statement. To achieve this, we can either add ghost statements that do not alter the program's behavior but can still be used to prove properties, for example, ghost calls. Alternatively, we could extend our DSL to prove such behavior without requiring a method call.

5.2.2 Automatically Generating Triggers

We saw in section 2.4.1, where we briefly mentioned triggers in top-level forall quantifiers, that we currently need to specify the triggers manually. We also need to write the correct hints for them manually. However, in many cases, generating the correct triggers and hints is straightforward, as we often want to trigger based on the current value of a variable in the current state. We could automatically create the trigger and the hints for them in simple cases.

This also applies to the case where we have pre-/postconditions, hyperassertions, and invariants, because every quantifier that does not quantify over states but rather other variables currently lacks a mechanism to generate triggers. This could then also be automated or manually specified by the Hypra user.

5.2.3 Automatically Infer Apply

In this thesis, we introduced a method for applying a specification in various ways. However, there is a case to be made that Hypra should be able to

generate the correct apply statement automatically. This may not always be possible, but in certain simple cases, such as in Figure 1.3, we should not need to write the apply statement. This could be done by automatically inferring which apply statement would hold at each point in a program and then automatically generating them in the Viper file.

5.2.4 Optimization of the Generated Viper Statements

As we saw, the size of the generated assertion by an apply statement can sometimes be considerable. For example, when applying associativity on four different method calls, we generate 256 different assertions, which creates a significant load for Viper and the SMT solver. However, many of those assertions do not contribute to anything constructive. For example, in the generated assertion in Figure 2.10, we could remove one obsolete assertion, as the equality operator is symmetric:

$$\forall\langle\sigma_1@\text{call}_1\rangle, \langle\sigma_2@\text{call}_2\rangle.\, \sigma_1(u) = \sigma_2(v)$$
$$\forall\langle\sigma_1@\text{call}_2\rangle, \langle\sigma_2@\text{call}_1\rangle.\, \sigma_1(v) = \sigma_2(u)$$

Both of those assertions state the same property.

Automatically generating the smallest possible assertion could help optimize the apply statement.

5.2.5 Automatic Framing when Leaving a Loop

As previously explained in the loop section, we can use accumulative labels of labels declared within a loop outside of it. However, we currently have no use for it since we do not have a forall-exists clause and framing for those labels, which means that we cannot generate any additional facts for our postcondition by applying those accumulative labels after a loop. Having a strategy for framing would be helpful in this case. However, one would need to be careful because a loop might not have been executed.

5.2.6 Automatic Framing for Loop Entry

In the example in Figure 4.4, we used a `dummyCall()` method with a label, which allowed us to use the disjunctive framing from the label entry. However, using a dummy method is not elegant and does not contribute to the program. To address this, we could allow all statements to have labels. This would solve the issue of the dummy method, but it would not solve the core issue, which is that we should automatically retain as much information about the program when entering a loop. This would include automatically generating a framing that connects the loop entry to the loop body and a framing connecting the end of the loop body to the next iteration. Then the loop

should be verified inline, allowing all the information about the program up to the loop to be retained. This approach could eliminate the need for framing for labels, as it automatically includes all necessary information. Consequently, it could also eliminate the advanced label rules, allowing labels to be used in if statements without introducing unsoundness.

Bibliography

[1] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *2008 21st IEEE Computer Security Foundations Symposium*, 2008, pp. 51–65. DOI: 10.1109/CSF.2008.7.

[2] T. Dardinier and P. Müller, "Hyper hoare logic: (dis-)proving program hyperproperties," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1485–1509, Jun. 2024, ISSN: 2475-1421. DOI: 10.1145/3656437. [Online]. Available: http://dx.doi.org/10.1145/3656437.

[3] R. W. Floyd., "Assigning meanings to programs," in *Proceedings of Symposium in Applied Mathematics (1967), 19-32*, 1967.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[5] T. Dardinier, A. Li, and P. Müller, "Hypra: A deductive program verifier for hyper hoare logic," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 1279–1308, Oct. 2024. DOI: 10.1145/3689756. [Online]. Available: https://dardinier.me/papers/OOPSLA24_Hypra.pdf.

[6] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49122-5.

[7] T. Kleymann, "Hoare logic and auxiliary variables," *Form. Asp. Comput.*, vol. 11, no. 5, pp. 541–566, Dec. 1999, ISSN: 0934-5043. DOI: 10.1007/s001650050057. [Online]. Available: https://doi.org/10.1007/s001650050057.

[8] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 57–69, ISBN: 9781450342612. DOI: 10.1145/2908080.2908092. [Online]. Available: https://doi.org/10.1145/2908080.2908092.

[9] E. D'Osualdo, A. Farzan, and D. Dreyer, "Proving hypersafety compositionally," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 289–314, Oct. 2022, ISSN: 2475-1421. DOI: 10.1145/3563298. [Online]. Available: http://dx.doi.org/10.1145/3563298.

Appendix A

Appendix

Every concept introduced in this thesis has been implemented in Hypra, enabling us to automatically prove all the Hypra programs shown in this thesis's examples, except for the example in Figure 3.5, which can only be proven by Hypra with unsound loop framing enabled.

This thesis utilizes the CADMO thesis template, which can be found at https://cadmo.ethz.ch/education/thesis/template.html.

Generative AI has been used in the following way:

- **Gemini 2.5:** Has been used to clarify the phrasing of this thesis as well as to generate documentation of the newly implemented methods. This has then been checked to be correct.
- **Grammarly:** Has been used to check the grammar and clarify phrasing in this thesis.

All ideas and substantive content remain our own work.

The repository containing the final implementation of this work, as well as the examples, can be found at https://github.com/Blausein/hypra.git.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

- ○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].
- ◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].
- ○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Automated Compositional Verification of Hyperproperties

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

Last name(s):	**First name(s):**
Neugebauer	Patrick

With my signature I confirm the following:

- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich the 12.9.2025	P. Neugebauer

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

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard