ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A General Approach to Formally Verify Viper Front-ends

Master Thesis

Hongyi Ling

April 23, 2024

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

Abstract

Writing correct programs becomes increasingly important given the widespread use of programs in critical domains. This is especially challenging when the program contains concurrency because of the large number of possible interleavings of the execution of parallel threads that need to be considered. Program verification techniques enable proving the absence of bugs in programs, and concurrent separation logic (CSL), one such technique, provides an elegant solution to the challenge of formally proving concurrent programs correct.

However, manually writing CSL proofs is cumbersome and time consuming. It is thus important to automate this process. Viper is one such automatic verification infrastructure that has been used as a foundation to develop various verifiers for real-world programming languages using CSL reasoning principles. This is achieved by developing Viper front-ends that translate input programs and their specifications into Viper programs. Nevertheless, few of these translations have been formally proved sound in an interactive theorem prover. Moreover, there lacks a general approach to take care of the common features of these front-ends and prove sound their translations to Viper once and for all.

In this thesis, we develop a general framework for proving sound Viper front-ends based on CSL. We also instantiate the framework with a concrete front-end, which shows that our framework is of practical use.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors Thibault Dardinier and Gaurav Parthasarathy for not only the new insights about the project they provided me during our meetings but also the many valuable suggestions they gave on presentations and academic writing.

I am also very grateful to Prof. Dr. Peter Müller for the opportunity to do my Master's thesis in his group.

Lastly, I would like to thank my family and friends for always being there and supporting me.

Contents

Chapter 1

Introduction

Program verification techniques enable proving the absence of bugs in programs, which is becoming increasingly important given the widespread use of programs in critical domains. Separation logic (SL) [1] is one such program verification technique for heap-manipulating programs. It provides a way to prove Hoare triples $\{P\}\ c\ \{Q\}$, where P is a precondition, c is the code to be executed and Q is a postcondition. A Hoare triple $\{P\}\ c\ \{Q\}$ holds if whenever P holds before the execution, then Q holds after the execution of c. Concurrent separation logic (CSL) [2] further extends SL to verify concurrent programs. It enables c in the Hoare triple $\{P\}\ c\ \{Q\}$ to contain concurrency, and asserts the triple only if it holds for all possible interleavings in the execution of parallel parts in c.

In practice, it is desirable to automatically verify source code statically. Viper [3] is an automatic verification infrastructure, which includes the Viper intermediate verification language (IVL) and verifiers to automatically check the correctness of Viper programs. The assertion language in the Viper IVL is based on SL. The Viper infrastructure has been used as a foundation to develop various verifiers for real-world programming languages using SL and CSL reasoning principles. This is achieved by developing Viper *front-ends* that translate input programs and their specifications, such as pre- and postconditions, into Viper programs. Examples of Viper front-ends include Gobra [4] for Go, Nagini [5] for Python, Prusti [6] for Rust, and VerCors [7] for Java.

For such a front-end translation to be *sound*, the correctness of the Viper program must imply the correctness of the input program w.r.t. the input specification. In general, the soundness of translations in current Viper front-ends is not formally proved. In recent work, Arlt [8] started addressing this issue, by formally defining a simple Viper front-end that supports concurrency and proving its translation sound once and for all in the Isabelle theorem prover [9]. Arlt's work shows the feasibility of formally proving the

soundness of Viper front-end translations. It deals directly with the operational semantics of the front-end program and that of the translated Viper program. This requires proving low-level steps to bridge the gap between the front-end's and Viper's operational semantics, which is cumbersome. Moreover, Arlt's work is specific to the translation of a particular front-end that has limited control flow structures (e.g., loops are not supported, and assignments of local variables in parallel compositions are neither supported) and assertion language (e.g., heap-dependent expressions and existential quantifiers are not supported). In addition to these restrictions, the front-end in Arlt's work only supports a simple binary ownership model (which only distinguishes between owning a heap location or not owning the heap location), and thus cannot for example verify programs where two threads read the same heap location (which is supported by fractional ownership models where the ownership of heap locations is represented by fractions).

In this thesis, we develop a general framework for proving Viper front-end translations sound once and for all in the Isabelle theorem prover [9]. Instead of directly working with the front-end's and Viper's operational semantics as in Arlt's work [8], we make use of existing CSL of the front-end that has been proved sound w.r.t. the front-end's operational semantics and SL proof representation of verified Viper programs, and prove sound the translation of the front-end to Viper by converting Viper SL proofs of the translated Viper programs to front-end CSL proofs of the front-end programs. Our framework supports front-ends with a larger subset of control structures and assertion language. Specifically, it supports control structures such as loops and assignments of local variables in parallel compositions, and heap-dependent expressions and existential quantifiers in the assertion language, while all of these features are not supported in Arlt's work [8]. Moreover, it supports more general ownership models for the front-end(e.g., fractional ownership models where the ownership of heap locations is represented by rational fractions). To show that our framework is practical, we instantiate our framework with a concrete CSL based front-end in Isabelle.

1.1 Outline

The rest of this thesis is organized as follows. In Chapter 2, we give background knowledge relevant for this thesis. In Chapter 3, we describe the main challenges of this thesis. Chapter 4 provides an overview of the general framework from a user's perspective. Then Chapter 5 formally presents the components of the framework. In this chapter, we also formally prove the soundness of the translation. In Chapter 6, we instantiate our framework with the front-end we describe in Chapter 2. Finally, in Chapter 7, we conclude this thesis by comparing it with Arlt's work [8] and pointing out possible future work.

Chapter 2

Background

In this chapter, we provide the background knowledge relevant to this thesis. We start with some general concepts about separation logic in Section 2.1, including our model of program states and the fractional ownership model in Section 2.1.1, the separation algebra model in Section 2.1.2, the well-typedness notion for program states in Section 2.1.3, and the semantic assertion notion and some useful property of it in Section 2.1.4. Next, in Section 2.2, we describe a concrete front-end that will be used in Chapter 6 to instantiate our framework and the CSL rules for it. Then, in Section 2.3, we give a brief introduction to a subset of Viper that is related to this thesis and Viper separation logic. Finally, in Section 2.4, we give the translation of some typical front-end components into Viper.

2.1 General Separation Logic Concepts

2.1.1 The State Model and Fractional Ownership Model

We model both the front-end and Viper program states to have two components: a store that models the values stored in local variables and a heap with some ownership model that models the shared memory.

Definition 2.1 (states) *A front-end or Viper state s is of the form (σ, h), where the store $\sigma : VN \rightharpoonup V$ is a partial function from variable names VN to values V, and h is the heap.*

A commonly used ownership model is the binary ownership model.

Definition 2.2 (binary ownership model) *A heap with a binary ownership model $h : L \rightharpoonup V$ is a partial function mapping heap locations L to values V. For each $l \in L$, $h(l)$ is defined when h exclusively owns the heap location l. In this case, $h(l)$ is the value stored at l.*

Compared with the binary ownership model, which only distinguishes between owning a heap location or not owning the heap location, the fractional ownership model [10] distinguishes read and write access by allowing splitting and recombining ownership to heap locations.

In this thesis, we consider two fractional ownership models: the rational permission model and the real permission model. In both of these models, the ownership of heap locations is represented by permission amounts in the interval $[0, 1]$ and, splitting and recombining the ownership are implemented by splitting and summing up permission amounts. The rational permission model, which further restricts permission amounts to be rational numbers, is commonly used by concurrent separation logics in the literature. On the other hand, Viper uses the real permission model, where the permission amounts are allowed to be any real numbers in the interval $[0, 1]$, since SMT solvers have built-in support for reals but not for rationals.

A heap with with some ownership model then consists of two parts: a permission mask $m : L \to P$ that records ownership $p \in P$ to each heap location $l \in L$, and a partial heap $h : L \rightharpoonup V$ that records the value $v \in V$ (if exists) stored at each heap location $l \in L$. And the partial heap should contain information for each heap location of which the permission heap has some ownership. That is, if $m(l) \neq 0_m$, then $h(l)$ should be defined, where $0_m \in P$ represents no ownership (in the fractional permission models, 0_m corresponds to the value 0). Specifically, we call a heap with a fractional permission model a *permission heap*.

2.1.2 Separation Algebra for Implicit Dynamic Frames

The separation algebra [11] abstracts the structure of the heaps with all these kinds of ownership models. The definition of a separation algebra for IDF starts with a *partial commutative monoid* (PCM) that abstracts the combination of permission masks and partial heaps. Then the operation of extracting the partial heap information out from the heap is abstracted as the *core* operation. Adding the core operation into a PCM results in a *PCM with core*. Finally, since in Implicit Dynamic Frames (IDF) [12] decouples the assertification of ownership of heap locations and the access to the heap locations, we need to make a distinction between the values at the heap locations that we have ownership of and are able to access, and the heap locations of which we have no ownership. The operation of erasing the values at these inaccessible heap locations is abstracted as the *stabilize* operation. Adding the stabilize operation and a *stable* predicate that decides whether the object contains any values at inaccessible heap locations into a PCM with core eventually results in a *separation algebra for IDF*.

Definition 2.3 (PCM) *A PCM $(P, \oplus)$ consists of a set P and a partial binary*

operator $\oplus : P \times P \rightharpoonup P$. *For* $x, y \in P$, *we say* x *is* compatible *with* y *if* $x \oplus y$ *is defined, and denote it as* $x \# y$. $\oplus$ *satisfies the following properties:*

(Commutativity) *For any* $x, y \in P$, $x \# y$ *if and only if* $y \# x$, *and* $x \oplus y = y \oplus x$ *when* x *is compatible with* y.

(Associativity) *For any* $x, y, z \in P$, $(x \oplus y) \oplus z$ *is defined if and only if* $x \oplus (y \oplus z)$ *is defined, and they have the same value when they are both defined.*

As a concrete example, the partial addition abstracts the compatibility and combination of the permission heaps. Two permission heaps $H_1 = (m_1, h_1)$ and $H_2 = (m_2, h_2)$ are compatible if the sum of their permissions to each heap location does not exceed the full permission amount 1 and they agree on the value at each heap location of which they both have knowledge (i.e., $\forall l \in L$, $m_1(l) + m_2(l) \leq 1$, and if both $h_1(l)$ and $h_2(l)$ are defined, then $h_1(l) = h_2(l)$). Two compatible permission heaps can then be combined by adding up their permission amount at each heap location and combining their knowledge of the partial heap. That is, for $(m_1, h_1) \# (m_2, h_2)$, $(m_1, h_1) \oplus (m_2, h_2) = (m, h)$ where

$$m(l) = m_1(l) + m_2(l),$$
$$h(l) = \begin{cases} h_1(l), & \text{if } h_1(l) \text{ is defined,} \\ h_2(l), & \text{otherwise.} \end{cases}$$

Compatibility guarantees that the permission amount $m(l)$ to each heap location l is still between 0 and 1, and that h agrees on the value with both h_1 and h_2 at each heap location.

Definition 2.4 (PCM with core) *A PCM with core* $(P, \oplus, \text{core})$ *consists of a PCM* $(P, \oplus)$ *and a unary operator* $\text{core} : P \to P$ *on* P *denoted as* $|\cdot|$. *Apart from commutativity and associativity that* $\oplus$ *should satisfy in order for* $(P, \oplus)$ *to be a PCM,* $|\cdot|$ *should satisfy:*

(C1) $x = x \oplus |x|$ *for any* $x \in P$.

(C2) $|x| = |x| \oplus |x|$ *for any* $x \in P$.

(C3) *For any* $x, c \in P$, *if* $x = x \oplus c$, *then there exists* $r \in P$ *such that* $|x| = c \oplus r$.

(C4) *For any* $a, b, c \in P$, *if* $c = a \oplus b$, *then* $|c| = |a| \oplus |b|$.

Intuitively, the core operation of an element x extracts out the duplicable information (e.g., the partial heap of permission heaps) of x. (C1) means that $|x|$ does not contain more information than x. (C2) expresses that the core does not contain any non-duplicable information (e.g., the permission amount). (C3) expresses that $|x|$ extracts out all the duplicable information of x. And (C4) means that $|\cdot|$ preserves the $\oplus$ operation.

In a PCM with core $(P, \oplus, |\cdot|)$, we can define a binary relation $\succeq$: $x \succeq y$ if and only if there exists some z such that $x = y \oplus z$. It is easy to show that $\succeq$ satisfies the three axioms reflexivity, antisymmetry, and transitivity of partial orders. As a result, $\succeq$ is a partial order of the PCM with core.

In the PCM of permission heaps, the core operation of a permission heap removes permissions to all heap locations (which are non-duplicable) but retains the partial heap (which is duplicable). That is, for a permission heap (m, h), $|(m, h)| = (\mathbf{0}_L, h)$, where the zero mask $\mathbf{0}_L$ is defined as $\mathbf{0}_L(l) = 0$ for any heap location $l \in L$.

In the permission heap set as a PCM with core, $H_1 = (m_1, h_1) \succeq H_2 = (m_2, h_2)$ reflects that H_1 has at least as much permission amount to each heap location, and knows at least as much about the partial heap than H_2 and agrees with H_2's partial heap h_2. That is, for any heap location l, $m_1(l) \geq m_2(l)$, and if $h_2(l)$ is defined and has value v, then $h_1(l)$ is also defined and $h_1(l) = v$.

Definition 2.5 (separation algebra for IDF) *A separation algebra for IDF* $(P, \oplus, \text{core}, \text{stabilize}, \text{stable})$ *consists of a PCM with core* $(P, \oplus, \text{core})$*, a unary operator* $\text{stabilize} : P \to P$ *and a predicate* $\text{stable} : P \to \{\text{True}, \text{False}\}$ *on* P*. Apart from axioms of PCM with core that* $(P, \oplus, \text{core})$ *needs to satisfy,* stabilize *and* stable *should satisfy:*

(S1) *For any* $x \in P$, $x = \text{stabilize}(x) \oplus |x|$.

(S2) *For any* $x, y, z \in P$*, if* $x = y \oplus z$ *then* $\text{stabilize}(x) = \text{stabilize}(y) \oplus \text{stabilize}(z)$.

(S3) *The* stabilize *operation always returns a stable result. That is, for any* $x \in P$, $\text{stable}(\text{stabilize}(x))$.

(S4) *For any* $x \in P$*, if* x *is already stable, i.e.,* $\text{stable}(x)$*, then the* stabilize *operation does not change* x*, i.e.,* $\text{stabilize}(x) = x$.

Intuitively, we want to use the separtion algebra for IDF to model the permission heap in a CSL: if the permission heap of one thread holds non-zero permission amount to some heap location, then no other thread has 1 permission, and the value at this heap location cannot be changed by other threads. So this value is *stable* w.r.t. the interference from other threads. On the other hand, 0 permission means that another thread could have 1 permission. The thread with 1 permission to the heap location could modify the value at this heap location, and thus this value is *unstable*. The stabilize operation removes the information that is unstable (e.g., the value at some heap location with no permission in a permission heap), and the stable predicate holds if and only if such information does not exist in a given object. The axioms in Definition 2.5 describes some basic requirements for these two functions. (S1) means that stabilize contains all the non-duplicable information (e.g., the permission amounts in permission heaps) that is removed by $|\cdot|$ and

does not add more information. (S2) expresses that stabilize preserves the $\oplus$ operation. (S3) and (S4) express the relation between stabilize and stable: stabilize indeed makes an object stable, and does not make any other changes to the objects that are already stable.

In the PCM with core of permission heaps, the stable predicate holds if and only if the permission heap does not contain any values at those heap locations to which it does not have permissions. That is, stable(m, h) if and only if for any heap location $l \in L$, if $m(l) > 0$ then $h(l)$ is defined. And the stabilize operation makes a permission heap $H = (m, h)$ stable by removing the heap values at all the heap locations to which H does not have permission, and leaving all other parts (the permission mask m and the values at those heap locations to which H has non-zero permission) unchanged. That is, stabilize$(H) = (m, h')$, where h' is defined as for any heap location $l \in L$, $h'(l)$ equals to $h(l)$ if $m(l) > 0$, and is undefined otherwise.

Although the above intuitions and instantiations for separation algebras for IDF are for the heap, the separation algebra for IDF can also model the store, and then the whole state:

1. Stores are a separation algebra for IDF, where for stores σ_1, σ_2, $\sigma_1 \# \sigma_2$ if and only if $\sigma_1 = \sigma_2$ (i.e., for any variable x, $\sigma_1(x)$ is defined if and only if $\sigma_2(x)$ is defined, and when they are both defined, $\sigma_1(x) = \sigma_2(x)$), and when $\sigma_1 \# \sigma_2$, $\sigma_1 \oplus \sigma_2 = \sigma_1$, core and stabilize are both identity mappings, and stable(σ) is trivially true for any store σ.

2. The Cartesian product of two separation algebras for IDF is a separation algebra for IDF, where $(x_1, y_1) \# (x_2, y_2)$ if and only if $x_1 \# x_2$ and $y_1 \# y_2$, and when $(x_1, y_1) \# (x_2, y_2)$, $(x_1, y_1) \oplus (x_2, y_2) = (x_1 \oplus x_2, y_1 \oplus y_2)$, the core and stabilize operator operates on both components (i.e., $|(x, y)| = (|x|, |y|)$ and stabilize$(x, y) = (\text{stabilize}(x), \text{stabilize}(y))$), and stable$(x, y)$ if and only if stable(x) and stable(y).

   As a result, the state, as the Cartesian product of stores and separation algebra for IDF modeled heaps, forms a separation algebra for IDF.

2.1.3 Type Context and Well-typed States

In this thesis, we study statically typed front-ends. In a statically typed front-end program, each variable has a fixed type. The type information for variables is stored by the *type context*. A type context $T : VN \rightharpoonup VT$ is a partial mapping from variable names VN to types VT that defines each declared variable's type.

In a statically typed front-end or Viper program, all variables should be declared before their first use, and if the program type checks, then the values of the variables will always be of the types in their declarations. The

type context will record each variable's type at the time of their declaration, and we call those states *well-typed* if all variables have their declared types.

Definition 2.6 (well-typedness) *A state ω is well-typed w.r.t. a type context T, if its store σ respects T. Concretely, for any variable $x \in VN$, $\sigma(x)$ is defined if and only if $T(x)$ is defined, and when they are both defined, $\sigma(x)$ is of type $T(x)$.*

In this thesis, we simplify the declare-before-use process for variables by separating their declarations from the program. Concretely, we require the program to declare all the variables at the beginning, and we will then deal with the rest of the program with the type context derived from the variable declarations. As a result, we do not consider variable declarations in this thesis. Instead, we will deal with the programs w.r.t. type contexts as the product of the variable declaration process.

2.1.4 Semantic Assertions

Apart from syntactic assertions, which have a syntactic form and whose semantics are given by an interpretation on its syntax, we will also use *semantic assertions* when we need to describe more properties of states (e.g., in the pre- and postconditions of front-end CSL and Viper SL rules that we will introduce later). A semantic assertion is equivalently a predicate on state, and can be represented by the set of states that satisfy it. Compared with syntactic assertions, semantic assertions capture all syntactic assertions and contain more assertions that cannot be syntactically expressed. They are therefore more expressive.

We will use separating conjunction, normal conjunction, and normal disjunction of semantic assertions. They are defined as follows.

Definition 2.7 (connectives for semantic assertions) *The separating conjunction, normal conjunction, and normal disjunction of semantic assertions A and B are denoted as $A * B$, $A \wedge B$, and $A \vee B$ respectively. Their semantics is defined as follows:*

- *$(A * B)(\omega)$ is true if and only there exist α and β such that $\alpha \oplus \beta = \omega$, and $A(\alpha)$ and $B(\beta)$ are both true.*
- *$(A \wedge B)(\omega)$ is true if and only if both $A(\omega)$ and $B(\omega)$ are true.*
- *$(A \vee B)(\omega)$ is true if and only if $A(\omega)$ or $B(\omega)$ is true.*

The front-ends that we study in this thesis and Viper both use IDF [12], which decouples the assertification of ownership of heap locations and the access to the heap locations. IDF assertions are in general not equivalent to separation logic assertions (where every access to the heap is coupled with certain permission amount). As a result, to define separation logic rules that are correct w.r.t. the operational semantics for front-ends and Viper, we need

to restrict the pre- and postconditions to be *self-framing*, under which IDF is equivalent to separation logic, as shown by Parkinson and Summers [12]. Intuitively, an assertion is self-framing if its truth is independent from other threads. That is, it assertifies enough permissions to fix the values of all heap locations on which it depends. Using the stabilize operator of the separation algebra for IDF, we can express self-framedness as follows:

Definition 2.8 (self-framedness) *A semantic assertion A is self-framing if and only if its satisfaction at any state ω only depends on the stable part of ω, i.e., $A(\omega) = A(\text{stabilize}(\omega))$.*

As an example, $\text{acc}(x.f, 1) * x.f = 7$ that assertifies 1 permission to the heap location $x.f$ and at the same time assertifies that the value at $x.f$ equals 7 is self-framing. However, without the assertification of enough permission amount to $x.f$, the single assertion $x.f = 7$ is not self-framing.

For an expression which is not able to assertify any permissions, it cannot be self-framing. In this case, we need an assertion to provide this assertification. We then say the assertion *frames* the expression.

Definition 2.9 (assertion frames expression) *A semantic assertion A frames an expression e if and only if for any state ω that satisfies A, $e(\omega) = e(\text{stabilize}(\omega))$, where $e(\omega)$ is the evaluation of e at state ω, which is either undefined or equal to some value, and the equality means that the left hand side is defined if and only if the right hand side is defined, and they have the same value when they are both defined.*

Apart from the help of framing, we sometimes also need the assertion to exclude the abnormal states for the expression. For example, when evaluating the division x/y, we want the assertion $y \neq 0$ to exclude the case where the division is not defined.

Definition 2.10 (well definedness) *An expression e is* well defined *w.r.t. a semantic assertion A if and only if for any state ω that satisfies A, $e(\omega)$ is defined.*

Finally, we redefine the concepts of entailment and independence of variable for assertions such that they only consider well-typed states.

Definition 2.11 (entailment for well-typed states) *For semantics assertions A and B, we say A entails B for well-typed states w.r.t. type context T, denoted as $A \Longrightarrow_T B$, if for any state ω that is well-typed w.r.t. T, ω satisfies A implies that ω satisfies B.*

Definition 2.12 (independence of variable) *A semantic assertion A is independent of variable x for well-typed states w.r.t. type context T, denoted* $\text{indep}_T(A, x)$, *if for any state ω that is well-typed w.r.t. T, and any value v of type $T(x)$, $A(\omega) = A(\omega(x \mapsto v))$, where $\omega(x \mapsto v)$ is the state obtained from ω by modifying the value of variable x to v and keeping the other parts unchanged.*

It is straightforward to show that the entailment in Definition 2.11 is weaker than the normal entailment, and it is reflexive and transitive, but not anti-symmetric.

Proposition 2.13 (properties about $\Longrightarrow_T$) *The entailment for well-typed states $\Longrightarrow_T$ is weaker than the normal entailment $\Longrightarrow$, i.e., if $A \Longrightarrow B$, then $A \Longrightarrow_T B$ for any type context T.*

Moreover, $\Longrightarrow_T$ is reflexive and transitive. That is,

- *$A \Longrightarrow_T A$ for any assertion A and any type context T. And*
- *For any assertions A, B, and C, and type context T, if $A \Longrightarrow_T B$ and $B \Longrightarrow_T C$, then $A \Longrightarrow_T C$.*

The notion of indep_T is also weaker than the normal independence notion.

Proposition 2.14 (indep_T is weaker than normal independence) *If variable x is independent to a semantic assertion A, i.e., $A(\omega) \iff A(\omega(x \mapsto v))$ for any v, then $\text{indep}_T(A, x)$.*

2.2 A Simple Viper Front-end

A Viper front-end takes as input a program and its specifications, and translates them into a Viper program, such that if the Viper program is correct, then the front-end program satisfies its specifications. Concurrent separation logic (CSL) [2] is a system for proving the correctness of front-end programs w.r.t. their specifications. A CSL of the front-end consists of several rules, and CSL proofs are constructed by applying these CSL rules finite times.

In this section, we will define a simple front-end that will be used to instantiate our general framework in Chapter 6. We start with its syntax in Section 2.2.1. Then we give an interpretation to the expressions and assertions of the front-end in Section 2.2.2, and define well-formedness requirements for front-end statements in Section 2.2.3. In Section 2.2.3, we also define free variables for front-end expressions, assertions, and statements, and written variables for front-end statements. They are used in defining well-formedness and will also be used in defining the CSL rules later. After these preparations, we give the CSL rules for front-end statements in Section 2.2.4. Since our framework does not deal with operational semantics of front-ends, we do not formally define it for our front-end. Instead, we only give intuitions for each statements, from which it is not hard to derived an operational semantics. Finally, we extend the front-end language with separating implications in assertions and method calls in Section 2.2.5. These extensions are only used to demonstrate the examples in Chapter 3. They are not supported by our framework in Chapter 5.

2.2.1 The Syntax of the Front-end

Our front-end supports integers, rationals, booleans, and references as its basic types.

$$T_f ::= \text{Int} \mid \text{Rat} \mid \text{Bool} \mid \text{Ref} \tag{2.1}$$

They correspond to values denoted as

$$V_f ::= \text{Int } n \mid \text{Rat } r \mid \text{Bool } b \mid \text{Ref } l \tag{2.2}$$

Here $n \in \mathbb{Z}$, $r \in \mathbb{Q}$, $b \in \{\text{True}, \text{False}\}$, and $l \in R \cup \{\text{Null}\}$ is either a non-null reference in R or the constant reference Null. V_f therefore consists of all front-end values.

The expressions of the front-end consist of integer, rational, or boolean literals, the Null-literal and local variables. In terms of operations on these objects, we support unary operators, binary operators, and conditional expressions. Specifically, we support the negation operator for booleans ($\neg$) and for numbers ($-$) as unary operators, and the addition ($+$), subtraction ($-$), multiplication ($\times$), Euclidean division ($\div$), modulo (mod), normal division ($/$), equality ($=$), non-equality ($\neq$), greater ($>$), greater or equal ($\geq$), less ($<$), less or equal ($\leq$), logical conjunction ($\wedge$), logical disjunction ($\vee$), and logical implication ($\Rightarrow$) as binary operators. In addition, as a result of using IDF [12], unlike traditional SL based front-ends that express ownership and reference to heap locations with the points-to operator, we decouple these two functionalities, use the accessibility predicate acc to express ownership to heap locations (introduced later in this section), and support heap access expressions via reference expressions and field identifiers.

$$\begin{aligned} E_f ::= {} & \text{Int } n \mid \text{Rat } r \mid \text{Bool } b \mid \text{Ref Null} \\ & \mid x \\ & \mid \text{uop } E_f \\ & \mid E_f \text{ bop } E_f \\ & \mid E_f \,?\, E_f : E_f \\ & \mid E_f.f \end{aligned} \tag{2.3}$$

Here f is a field identifier, and

$$\begin{aligned} &\text{uop} ::= \neg \mid - \\ &\text{bop} ::= + \mid - \mid \times \mid \div \mid \text{mod} \mid / \mid = \mid \neq \mid > \mid \geq \mid < \mid \leq \mid \wedge \mid \vee \mid \Rightarrow \end{aligned} \tag{2.4}$$

The assertions of the front-end are constructed from primitive assertions using certain connectives. In our front-end, the only primitive assertion is the accessibility predicate.

$$A_p ::= \text{acc}(E_f.f, E_f) \tag{2.5}$$

We support the following constructs from primitive assertions: the separating conjunction ($*$), the normal conjunction ($\wedge$), the normal disjunction ($\vee$), the normal implication ($\longrightarrow$) with expressions as premises, the existential quantifier ($\exists$), and expressions.

$$\begin{aligned} A_f ::= & \; A_p \\ & \mid A_f * A_f \\ & \mid A_f \wedge A_f \\ & \mid A_f \vee A_f \\ & \mid E_f \longrightarrow A_f \\ & \mid \exists x : T_f.\, A_f \\ & \mid E_f \end{aligned} \tag{2.6}$$

Finally, the statements of the front-end are constructed from primitive statements using certain compositions as we show next. In our front-end, the primitive statements include assignments to local variables, assignments to the heap, and allocating new heap locations.

$$\begin{aligned} S_p ::= & \; x \texttt{ := } E_f \\ & \mid E_f.f \texttt{ := } E_f \\ & \mid x \texttt{ := } \textbf{alloc}(f_1, \ldots, f_n) \end{aligned} \tag{2.7}$$

Intuitively, "x := e" evaluates e to some value v, and assigns v to variable x; "$e.f$:= e'" evaluates e to a non-null reference r and e' to some value v, and then assigns v to the heap location $r.f$; "x := **alloc**$(f_1, \ldots, f_n)$" allocates new heap locations $r.f_1, \ldots, r.f_n$ with a fresh reference r, and then assigns r to variable x.

We support the following constructs from primitive statements: the sequential composition, the conditional branching, the loop, and the parallel composition.

$$\begin{aligned} S_f ::= & \; S_p \\ & \mid S_f;\, S_f \\ & \mid \textbf{if } E_f \textbf{ then } S_f \textbf{ else } S_f \\ & \mid \textbf{while } E_f \textbf{ inv } A_f \textbf{ do } S_f \\ & \mid \{A_f\}\, S_f\, \{A_f\} \parallel \{A_f\}\, S_f\, \{A_f\} \end{aligned} \tag{2.8}$$

Intuitively, sequential composition "c_1; c_2" means executing c_2 after the execution of c_1. Conditional branching "**if** e **then** c_1 **else** c_2" executes c_1 if e evaluates to True, and c_2 if e evaluates to False. The loop "**while** e **inv** I **do** c"

executes c as long as e evaluates to True, and exits the loop when e evaluates to False. The loop invariant I indicates the assertion that should always be satisfied every time entering or exiting the loop. It provides a guide on how the program should be verified and translated into Viper, but does not actually affect the execution of the program. Parallel composition "$\{P_1\}\ c_1\ \{Q_1\}\ \|\ \{P_2\}\ c_2\ \{Q_2\}$" means executing c_1 and c_2 in parallel. The assertions P_1 and P_2 indicate the resources that should be given to c_1 and c_2, and Q_1 and Q_2 indicate the resources that will be returned by c_1 and c_2. As the loop invariant in the loop, all of the assertions P_1, P_2, Q_1, and Q_2 only provide a guide on how the program should be verified and translated into Viper, but do not affect the execution of the front-end program.

2.2.2 The Interpretation of Expressions and Assertions

Our front-end uses the rational permission model. As introduced in Section 2.1.1, we can represent its state with $(\sigma, (m, h))$, where $\sigma : VN \rightharpoonup V_f$ is the store, $m : L \to [0, 1] \cap \mathbb{Q}$ is the permission mask, and $h : L \rightharpoonup V_f$ is the partial heap. Furthermore, a heap location $l \in L$ is determined by a non-null reference $r \in R$ and a field identifier f. We denote it by $l = r.f$.

We first define the evaluation functions $\text{eval}_u : \text{uop} \times V_f \rightharpoonup V_f$ and $\text{eval}_b : V_f \times \text{bop} \times V_f \rightharpoonup V_f$ for unary and binary operators. The unary operator $\neg$ operates only on booleans, and $-$ operates on both integers and rationals.

$$\begin{aligned}
&\text{eval}_u(\neg(\text{Bool } b)) \triangleq \text{Bool } (\neg b),\\
&\text{eval}_u(-(\text{Int } n)) \triangleq \text{Int } (-n),\\
&\text{eval}_u(-(\text{Rat } r)) \triangleq \text{Rat } (-r).
\end{aligned}$$

The binary operators $+$, $-$, $\times$, $/$, $=$, $\neq$, $>$, $\geq$, $<$, $\leq$ operate on two integers or two rationals in the way just as their names indicate.

$$\begin{aligned}
&\text{eval}_b(\text{Int } n_1 + \text{Int } n_2) \triangleq \text{Int } (n_1 + n_2),\\
&\text{eval}_b(\text{Int } n_1 - \text{Int } n_2) \triangleq \text{Int } (n_1 - n_2),\\
&\text{eval}_b(\text{Int } n_1 \times \text{Int } n_2) \triangleq \text{Int } (n_1 \cdot n_2),\\
&\text{eval}_b(\text{Int } n_1 / \text{Int } n_2) \triangleq \begin{cases} \text{Rat } (n_1/n_2), & \text{if } n_2 \neq 0,\\ \text{undefined}, & \text{otherwise,} \end{cases}\\
&\text{eval}_b(\text{Int } n_1 = \text{Int } n_2) \triangleq \text{Bool } (n_1 = n_2),\\
&\text{eval}_b(\text{Int } n_1 \neq \text{Int } n_2) \triangleq \text{Bool } (n_1 \neq n_2),\\
&\text{eval}_b(\text{Int } n_1 > \text{Int } n_2) \triangleq \text{Bool } (n_1 > n_2),\\
&\text{eval}_b(\text{Int } n_1 \geq \text{Int } n_2) \triangleq \text{Bool } (n_1 \geq n_2),\\
&\text{eval}_b(\text{Int } n_1 < \text{Int } n_2) \triangleq \text{Bool } (n_1 < n_2),
\end{aligned}$$

$$
\begin{aligned}
&\text{eval}_b(\text{Int } n_1 \leq \text{Int } n_2) \triangleq \text{Bool } (n_1 \leq n_2),\\
&\text{eval}_b(\text{Rat } r_1 + \text{Rat } r_2) \triangleq \text{Rat } (r_1 + r_2),\\
&\text{eval}_b(\text{Rat } r_1 - \text{Rat } r_2) \triangleq \text{Rat } (r_1 - r_2),\\
&\text{eval}_b(\text{Rat } r_1 \times \text{Rat } r_2) \triangleq \text{Rat } (r_1 \cdot r_2),\\
&\text{eval}_b(\text{Rat } r_1 / \text{Rat } r_2) \triangleq \begin{cases} \text{Rat } (r_1/r_2), & \text{if } r_2 \neq 0,\\ \text{undefined}, & \text{otherwise,} \end{cases}\\
&\text{eval}_b(\text{Rat } r_1 = \text{Rat } r_2) \triangleq \text{Bool } (r_1 = r_2),\\
&\text{eval}_b(\text{Rat } r_1 \neq \text{Rat } r_2) \triangleq \text{Bool } (r_1 \neq r_2),\\
&\text{eval}_b(\text{Rat } r_1 > \text{Rat } r_2) \triangleq \text{Bool } (r_1 > r_2),\\
&\text{eval}_b(\text{Rat } r_1 \geq \text{Rat } r_2) \triangleq \text{Bool } (r_1 \geq r_2),\\
&\text{eval}_b(\text{Rat } r_1 < \text{Rat } r_2) \triangleq \text{Bool } (r_1 < r_2),\\
&\text{eval}_b(\text{Rat } r_1 \leq \text{Rat } r_2) \triangleq \text{Bool } (r_1 \leq r_2).
\end{aligned}
$$

Moreover, we allow the two operands of $\times$ and $/$ to be an integer and a rational.

$$
\begin{aligned}
&\text{eval}_b(\text{Int } n \times \text{Rat } r) \triangleq \text{Rat } n \cdot r,\\
&\text{eval}_b(\text{Rat } r \times \text{Int } n) \triangleq \text{Rat } r \cdot n,\\
&\text{eval}_b(\text{Int } n / \text{Rat } r) \triangleq \begin{cases} \text{Rat } (n/r), & \text{if } r \neq 0,\\ \text{undefined}, & \text{otherwise,} \end{cases}\\
&\text{eval}_b(\text{Rat } r / \text{Int } n) \triangleq \begin{cases} \text{Rat } (r/n), & \text{if } n \neq 0,\\ \text{undefined}, & \text{otherwise.} \end{cases}
\end{aligned}
$$

The Euclidean division $\div$ and modulo mod operate on two integers n_1 and n_2 and evaluate to the quotient and the remainder of n_1 divided by n_2 respectively. That is, when $n_2 \neq 0$, take the unique q and r satisfying $n_1 = qn_2 + r$ and $0 \leq r < |n_2|$. Then

$$
\begin{aligned}
&\text{eval}_b(\text{Int } n_1 \div \text{Int } n_2) \triangleq \text{Int } q,\\
&\text{eval}_b((\text{Int } n_1) \text{ mod } (\text{Int } n_2)) \triangleq \text{Int } r.
\end{aligned}
$$

If $n_2 = 0$, then both $\text{eval}_b(\text{Int } n_1 \div \text{Int } n_2)$ and $\text{eval}_b((\text{Int } n_1) \text{ mod } (\text{Int } n_2))$ are undefined.

Finally, the logical binary operators $\wedge$, $\vee$, and $\Rightarrow$, and the equality ($=$) and non-equality ($\neq$) operators operate on two boolean values in the standard way.

$$
\text{eval}_b(\text{Bool } b_1 = \text{Bool } b_2) \triangleq \text{Bool } (b_1 = b_2),
$$

$$\begin{aligned}
&\mathrm{eval}_b(\text{Bool } b_1 \neq \text{Bool } b_2) \triangleq \text{Bool } (b_1 \neq b_2),\\
&\mathrm{eval}_b(\text{Bool } b_1 \wedge \text{Bool } b_2) \triangleq \text{Bool } (b_1 \wedge b_2),\\
&\mathrm{eval}_b(\text{Bool } b_1 \vee \text{Bool } b_2) \triangleq \text{Bool } (b_1 \vee b_2),\\
&\mathrm{eval}_b(\text{Bool } b_1 \Rightarrow \text{Bool } b_2) \triangleq \text{Bool } (b_1 \longrightarrow b_2).
\end{aligned}$$

Now we define the interpretation of expressions.

Definition 2.15 (interpretation of expressions) *Given a front-end expression e, a front-end state ω, and a front-end value $v \in V_f$, we say that e is interpreted as v in ω if and only if*

$$\langle e, \omega\rangle \Downarrow_f v.$$

The interpretation is inductively defined as shown in Figure 2.1.

$$\frac{}{\langle \text{Int } n, \omega\rangle \Downarrow_f \text{Int } n}\ (\text{Int}) \qquad \frac{}{\langle \text{Rat } r, \omega\rangle \Downarrow_f \text{Rat } r}\ (\text{Rat})$$

$$\frac{}{\langle \text{Bool } b, \omega\rangle \Downarrow_f \text{Bool } b}\ (\text{Bool}) \qquad \frac{}{\langle \text{Ref Null}, \omega\rangle \Downarrow_f \text{Ref Null}}\ (\text{Null})$$

$$\frac{\langle e, \omega\rangle \Downarrow_f v \quad \mathrm{eval}_u(\text{uop } v) = v'}{\langle \text{uop } e, \omega\rangle \Downarrow_f v'}\ (\text{Uop}) \qquad \frac{\langle e_1, \omega\rangle \Downarrow_f v_1 \quad \langle e_2, \omega\rangle \Downarrow_f v_2 \quad \mathrm{eval}_b(v_1 \text{ bop } v_2) = v}{\langle e_1 \text{ bop } e_2, \omega\rangle \Downarrow_f v}\ (\text{Bop})$$

$$\frac{\sigma(x) \text{ is defined}}{\langle x, (\sigma, H)\rangle \Downarrow_f \sigma(x)}\ (\text{Var}) \qquad \frac{\langle e_1, \omega\rangle \Downarrow_f \text{Bool False}}{\langle e_1 \wedge e_2, \omega\rangle \Downarrow_f \text{Bool False}}\ (\text{AndLazy})$$

$$\frac{\langle e_1, \omega\rangle \Downarrow_f \text{Bool True}}{\langle e_1 \vee e_2, \omega\rangle \Downarrow_f \text{Bool True}}\ (\text{OrLazy}) \qquad \frac{\langle e_1, \omega\rangle \Downarrow_f \text{Bool False}}{\langle e_1 \Rightarrow e_2, \omega\rangle \Downarrow_f \text{Bool True}}\ (\text{ImpLazy})$$

$$\frac{\langle e_1, \omega\rangle \Downarrow_f \text{Bool True} \quad \langle e_2, \omega\rangle \Downarrow_f v}{\langle e_1 \,?\, e_2 : e_3, \omega\rangle \Downarrow_f v}\ (\text{CondTrue}) \qquad \frac{\langle e_1, \omega\rangle \Downarrow_f \text{Bool False} \quad \langle e_3, \omega\rangle \Downarrow_f v}{\langle e_1 \,?\, e_2 : e_3, \omega\rangle \Downarrow_f v}\ (\text{CondFalse})$$

$$\frac{\langle e, (\sigma, (m, h))\rangle \Downarrow_f \text{Ref } r \qquad r \neq \text{Null} \qquad h(r.f) \text{ is defined}}{\langle e.f, (\sigma, (m, h))\rangle \Downarrow_f h(r.f)}\ (\text{HeapAcc})$$

Figure 2.1: The interpretation of front-end expressions.

For front-end assertions, we first interpret primitive assertions, and then inductively interpret the composite assertions. In Chapter 5, our framework will take the interpretation of primitive assertions as input, and make requirements on the inductive interpretation of the compositions of assertions. As

an instantiation of the framework, the inductive interpretation of composite assertions of our front-end needs to satisfy the requirements made by the framework.

Definition 2.16 (interpretation of assertions) *Given a front-end assertion A and a front-end state ω, we say that ω satisfies A if and only if*

$$\omega \vDash_f A.$$

Figure 2.2 gives interpretation rules for primitive assertions. Figure 2.3 defines the inductive interpretation rules for composite assertions.

$$\frac{\langle e, (\sigma, (m, h))\rangle \Downarrow_f \text{Ref } r \qquad r \neq \text{Null} \qquad \langle p, (\sigma, (m, h))\rangle \Downarrow_f \text{Rat } v \qquad v > 0 \qquad m(r.f) \geq v}{(\sigma, (m, h)) \vDash_f \text{acc}(e.f, p)} \text{ (AccPos)}$$

$$\frac{\langle e, (\sigma, (m, h))\rangle \Downarrow_f \text{Ref } r \qquad \langle p, (\sigma, (m, h))\rangle \Downarrow_f \text{Rat } 0}{(\sigma, (m, h)) \vDash_f \text{acc}(e.f, p)} \text{ (AccZero)}$$

Figure 2.2: Interpretation rules for front-end primitive assertions.

$$\frac{\alpha \vDash_f A \qquad \beta \vDash_f B \qquad \alpha \oplus \beta = \omega}{\omega \vDash_f A * B} \text{ (Star)} \qquad \frac{\omega \vDash_f A \qquad \omega \vDash_f B}{\omega \vDash_f A \wedge B} \text{ (Conj)}$$

$$\frac{\omega \vDash_f A}{\omega \vDash_f A \vee B} \text{ (DisjL)} \qquad \frac{\omega \vDash_f B}{\omega \vDash_f A \vee B} \text{ (DisjR)}$$

$$\frac{\langle e, \omega\rangle \Downarrow_f \text{Bool } b \qquad b = \text{True} \implies \omega \vDash_f A}{\omega \vDash_f e \longrightarrow A} \text{ (Imp)} \qquad \frac{\langle e, \omega\rangle \Downarrow_f \text{Bool True}}{\omega \vDash_f e} \text{ (Pure)}$$

$$\frac{v \text{ is of type } t \qquad \omega(x \mapsto v) \vDash_f A}{\omega \vDash_f \exists x : t.\, A} \text{ (Exist)}$$

Figure 2.3: Inductive interpretation for front-end composite assertions.

2.2.3 Well-formed Statements

The well-formedness predicate on front-end statements imposes additional requirements on the statements apart from the syntax. It additionally takes a front-end type context as an argument, which gives the variable declaration

context of the program and allows it to perform part of type checks for our front-end.

We first define the set of free variables for expressions, assertions, and statements, and the written variable set for statements. They will be used in defining the well-formedness predicate.

The free variables of an expression, an assertion, or a statement are the variables that appear syntactically, and thus might have an influence on the semantics.

Definition 2.17 (free variables of expressions) *The free variable set* fv_e *for expressions is inductively defined as follows:*

$$\begin{aligned}
&\mathrm{fv}_e(\text{Int } n) = \mathrm{fv}_e(\text{Rat } r) = \mathrm{fv}_e(\text{Bool } b) = \mathrm{fv}_e(\text{Ref Null}) = \varnothing,\\
&\mathrm{fv}_e(x) = \{x\},\\
&\mathrm{fv}_e(\text{uop } e) = \mathrm{fv}_e(e),\\
&\mathrm{fv}_e(e_1 \text{ bop } e_2) = \mathrm{fv}_e(e_1) \cup \mathrm{fv}_e(e_2),\\
&\mathrm{fv}_e(e_1 \,?\, e_2 : e_3) = \mathrm{fv}_e(e_1) \cup \mathrm{fv}_e(e_2) \cup \mathrm{fv}_e(e_3),\\
&\mathrm{fv}_e(e.f) = \mathrm{fv}_e(e).
\end{aligned}$$

It is straightforward to show that for any front-end expression e, $\mathrm{fv}_e(e)$ is a finite set, and that $\mathrm{fv}_e(e)$ does contain all the variables that might affect the interpretation of e in the sense that for any variable $x \notin \mathrm{fv}_e(e)$, the evaluation of e is independent to the value of x (i.e., $\langle e, \omega\rangle \Downarrow_f v \iff \langle e, \omega(x \mapsto v')\rangle \Downarrow_f v$ for any v').

Definition 2.18 (free variables of assertions) *The free variable set* fv_a *for assertions is defined inductively on the structure of assertions. For the base case* $A \triangleq \mathrm{acc}(e.f, p)$,

$$\mathrm{fv}_a(A) = \mathrm{fv}_e(e) \cup \mathrm{fv}_e(p). \tag{2.9}$$

The inductive cases are listed in Equation 2.10.

$$\begin{aligned}
&\mathrm{fv}_a(A * B) = \mathrm{fv}_a(A) \cup \mathrm{fv}_a(B),\\
&\mathrm{fv}_a(A \wedge B) = \mathrm{fv}_a(A) \cup \mathrm{fv}_a(B),\\
&\mathrm{fv}_a(A \vee B) = \mathrm{fv}_a(A) \cup \mathrm{fv}_a(B),\\
&\mathrm{fv}_a(e \longrightarrow A) = \mathrm{fv}_e(e) \cup \mathrm{fv}_a(A),\\
&\mathrm{fv}_a(\exists x : t.\, A) = \mathrm{fv}_a(A) \backslash \{x\},\\
&\mathrm{fv}_a(e) = \mathrm{fv}_e(e).
\end{aligned} \tag{2.10}$$

It is straightforward to show that for any front-end assertion A, $\mathrm{fv}_a(A)$ is a finite set, and that $\mathrm{fv}_a(A)$ does contain all the variables that might affect the interpretation of A in the sense that for any variable $x \notin \mathrm{fv}_a(A)$, the

satisfaction of A is independent to the value of x (i.e., $A \vDash_f \omega \iff A \vDash_f \omega(x \mapsto v)$ for any v).

Definition 2.19 (free variables of statements) *The free variable set* fv_c *for statements is defined inductively on the structure of statements. The three base cases are defined as follows:*

$$\begin{aligned} &\mathrm{fv}_c(x \texttt{ := } e) = \{x\} \cup \mathrm{fv}_e(e), \\ &\mathrm{fv}_c(e.f \texttt{ := } e') = \mathrm{fv}_e(e) \cup \mathrm{fv}_e(e'), \\ &\mathrm{fv}_c(x \texttt{ := alloc}(f_1, \ldots, f_n)) = \{x\}. \end{aligned} \tag{2.11}$$

The inductive cases are defined as follows:

$$\begin{aligned} &\mathrm{fv}_c(c_1; c_2) = \mathrm{fv}_c(c_1) \cup \mathrm{fv}_c(c_2), \\ &\mathrm{fv}_c(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) = \mathrm{fv}_e(e) \cup \mathrm{fv}_c(c_1) \cup \mathrm{fv}_c(c_2), \\ &\mathrm{fv}_c(\texttt{while } e \texttt{ inv } I \texttt{ do } c) = \mathrm{fv}_e(e) \cup \mathrm{fv}_c(c), \\ &\mathrm{fv}_c(\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}) = \mathrm{fv}_c(c_1) \cup \mathrm{fv}_c(c_2). \end{aligned} \tag{2.12}$$

It is straightforward to show that for any front-end statement c, $\mathrm{fv}_c(c)$ is a finite set. And by the intuition of the primitive statements and the compositions in Section 2.2.1, $\mathrm{fv}_c(c)$ indeed includes all the variables that might affect the semantics of c, in the sense that for any variable $x \notin \mathrm{fv}_c(c)$, the execution of c is independent to the value of x and does not change the value of x either. Specifically, the invariant of the loop "$\texttt{while } e \texttt{ inv } I \texttt{ do } c$" and the pre- and postconditions of the parallel composition "$\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$" do not participate in the inductive computation of free variables. This is because they are only used for guiding the verification and translation, but do not affect the execution of the statement. As we will show in Section 2.2.4, they do not affect the CSL rules either.

We then define the written variable set for statements. Intuitively, it contains all the variables that might be modified by executing the statement.

Definition 2.20 (written variable set for statements) *The written variable set* wr *for statements is defined inductively on the structure of statements. The three base cases are defined as follows:*

$$\begin{aligned} &\mathrm{wr}(x \texttt{ := } e) = \{x\}, \\ &\mathrm{wr}(e.f \texttt{ := } e') = \varnothing, \\ &\mathrm{wr}(x \texttt{ := alloc}(f_1, \ldots, f_n)) = \{x\}. \end{aligned} \tag{2.13}$$

The inductive cases are defined as follows:

$$\begin{aligned} &\mathrm{wr}(c_1; c_2) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2), \\ &\mathrm{wr}(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2), \\ &\mathrm{wr}(\texttt{while } e \texttt{ inv } I \texttt{ do } c) = \mathrm{wr}(c), \\ &\mathrm{wr}(\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2). \end{aligned} \tag{2.14}$$

Note that in the conditional branching "**if** e **then** c_1 **else** c_2" and the loop "**while** e **inv** I **do** c", the variables in the condition expression e appears in the set of free variables but not the set of written variables. This is because the expressions in our front-end do not have any side effects such as modifying the state.

We finally define the well-formedness predicate for front-end statements.

Definition 2.21 (well-formed statements) *We define well-formedness of statements w.r.t. type context T (denoted as wf_T) inductively on the structure of statements. The three base cases are defined as follows:*

- $\mathrm{wf}_T(x := e)$ *if and only if* $T(x)$ *is defined.*
- $\mathrm{wf}_T(x := \mathbf{alloc}(f_1, \ldots, f_n))$ *if and only if* $T(x)$ *is defined.*
- $\mathrm{wf}_T(e.f := e')$ *always holds.*

The inductive cases are defined as follows:

- $\mathrm{wf}_T(c_1; c_2)$ *if and only if* $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
- $\mathrm{wf}_T(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)$ *if and only if* $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
- $\mathrm{wf}_T(\texttt{while}\ b\ \texttt{inv}\ I\ \texttt{do}\ c)$ *if and only if* $\mathrm{wf}_T(c)$, *I is self-framing, and I frames b.*
- $\mathrm{wf}_T(\{P_1\}\ c_1\ \{Q_1\} \parallel \{P_2\}\ c_2\ \{Q_2\})$ *requires the following four conditions:*
  1. $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
  2. *All the pre- and postconditions P_1, P_2, Q_1, and Q_2 are self-framing.*
  3. $\mathrm{wr}(c_1)$ *is disjoint from* $\mathrm{fv}(c_2)$, $\mathrm{fv}(P_2)$, *and* $\mathrm{fv}(Q_2)$.
  4. $\mathrm{wr}(c_2)$ *is disjoint from* $\mathrm{fv}(c_1)$, $\mathrm{fv}(P_1)$, *and* $\mathrm{fv}(Q_1)$.

2.2.4 The CSL Rules for Statements

Vafeiadis [13] defines a simple concurrency-supporting language with a semantics and a set of CSL rules for the language, and proves the soundness of the CSL rules w.r.t. the operational semantics of the language. We will adjust the rules to define the CSL rules for our front-end statements. For each adjustment, we will argue for its correctness.

We assume that the type check of the front-end programs can be performed separately, and we will only deal with well-typed programs. For a well-typed program, the type context T corresponding to the variable declarations at the beginning of the program will not change throughout the execution of the program. That is, starting from any state well-typed w.r.t. T and executing the program for an arbitrary number of steps, the resulting state is still well-typed w.r.t. T. As a result, our CSL rules for the front-end may only be

sound w.r.t. its operational semantics when executing the program from a state that is well-typed w.r.t. the type context specified by the program.

We classify the rules into three classes: a class of rules for primitive statements, a class of rules for composite statements, and the other class of rules (called structural rules) for refining pre- and postconditions of existing CSL proofs.

Definition 2.22 (CSL rules) *We use $T \vdash_{\text{FE}} \{P\}\ c\ \{Q\}$ to denote that the triple $\{P\}\, c\, \{Q\}$ is CSL-provable w.r.t. front-end type context T, where P and Q are front-end semantic assertions and c is a front-end statement.*

The inference rules of $\vdash_{\text{FE}}$ for primitive statements are given in Figure 2.4, where we use $\omega(r.f \mapsto v)$ to denote the state obtained from ω by modifying the value at heap location $r.f$ of the partial heap of ω to v and keeping the other parts unchanged. The semantic assertions $A[x \mapsto e]$, $e.f \overset{1}{\mapsto} \_$, and $P[e.f \mapsto e']$ are defined as follows:

- *ω satisfies $A[x \mapsto e]$ if and only if there exist some state ω_0 and value v such that $A(\omega_0)$ holds, $\langle e, \omega_0\rangle \Downarrow_f v$, and $\omega = \omega_0(x \mapsto v)$.*
- *ω satisfies $e.f \overset{1}{\mapsto} \_$ if and only if there exist some non-null reference r such that $\langle e, \omega\rangle \Downarrow_f \text{Ref}\ r$, and ω has exactly 1 permission to the heap location $r.f$.*
- *ω satisfies $P[e.f \mapsto e']$ if and only if there exist some state ω_0, non-null reference r, and value v, such that $P(\omega_0)$ holds, $\langle e, \omega_0\rangle \Downarrow_f \text{Ref}\ r$, $\langle e', \omega_0\rangle \Downarrow_f v$, and $\omega = \omega_0(r.f \mapsto v)$.*

And $\circledast$ denotes the separating conjunction of a list of assertions:

$$\overset{n}{\underset{i=1}{\circledast}} A_i = A_1 * \cdots * A_n.$$

The rules for composite statements are given in Figure 2.5, and the structural rules for refining pre- and postconditions of existing CSL proofs are given in Figure 2.6.

$$\frac{A \text{ is self-framing} \qquad e \text{ is well defined w.r.t. } A \qquad A \text{ frames } e}{T \vdash_{\text{FE}} \{A\}\ x := e\ \{A[x \mapsto e]\}}\ \text{(AssignVar)}$$

$$\frac{\begin{array}{c} P = A * e.f \overset{1}{\mapsto} \_ \qquad P \text{ is self-framing} \\ e' \text{ is well defined w.r.t. } P \qquad P \text{ frames } e' \end{array}}{T \vdash_{\text{FE}} \{P\}\ e.f := e'\ \{P[e.f \mapsto e']\}}\ \text{(AssignField)}$$

$$\frac{}{T \vdash_{\text{FE}} \{\text{True}\}\ x := \texttt{alloc}(f_1, \ldots, f_n)\ \left\{ \overset{n}{\underset{i=1}{\circledast}} \text{acc}(x.f_i, 1) \right\}}\ \text{(Alloc)}$$

Figure 2.4: The inference CSL rules for primitive statements.

$$\frac{T \vdash_{\text{FE}} \{P\}\ c_1\ \{R\} \qquad T \vdash_{\text{FE}} \{R\}\ c_2\ \{Q\}}{T \vdash_{\text{FE}} \{P\}\ c_1; c_2\ \{Q\}}\ \text{(SeqFE)}$$

$$\frac{\begin{array}{c}e \text{ is well defined w.r.t. } P\\ T \vdash_{\text{FE}} \{P \wedge e\}\ c_1\ \{Q_1\} \qquad T \vdash_{\text{FE}} \{P \wedge \neg e\}\ c_2\ \{Q_2\}\end{array}}{T \vdash_{\text{FE}} \{P\}\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q_1 \vee Q_2\}}\ \text{(IfFE)}$$

$$\frac{\begin{array}{c}T \vdash_{\text{FE}} \{P \wedge e\}\ c\ \{P\}\\ P \text{ frames } e \qquad e \text{ is well defined w.r.t. } P\end{array}}{T \vdash_{\text{FE}} \{P\}\ \textbf{while}\ e\ \textbf{inv}\ I\ \textbf{do}\ c\ \{P \wedge \neg e\}}\ \text{(WhileFE)}$$

$$\frac{\begin{array}{c}T \vdash_{\text{FE}} \{A_1\}\ c_1\ \{B_1\} \qquad T \vdash_{\text{FE}} \{A_2\}\ c_2\ \{B_2\}\\ \text{fv}(c_1) \cap \text{wr}(c_2) = \varnothing \qquad \text{fv}(c_2) \cap \text{wr}(c_1) = \varnothing\\ \forall x \in \text{wr}(c_2).\, \text{indep}_T(A_1, x) \text{ and } \text{indep}_T(B_1, x)\\ \forall x \in \text{wr}(c_1).\, \text{indep}_T(A_2, x) \text{ and } \text{indep}_T(B_2, x)\end{array}}{T \vdash_{\text{FE}} \{A_1 * A_2\}\ \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}\ \{B_1 * B_2\}}\ \text{(ParFE)}$$

Figure 2.5: The inference CSL rules for compositions of statements.

$$\frac{\begin{array}{c}T \vdash_{\text{FE}} \{P\}\ c\ \{Q\}\\ R \text{ is self-framing} \qquad \forall x \in \text{wr}(c).\, \text{indep}_T(R, x)\end{array}}{T \vdash_{\text{FE}} \{P * R\}\ c\ \{Q * R\}}\ \text{(FrameFE)}$$

$$\frac{T \vdash_{\text{FE}} \{P\}\ c\ \{Q\} \qquad P' \text{ is self-framing} \qquad P' \Longrightarrow_T P}{T \vdash_{\text{FE}} \{P'\}\ c\ \{Q\}}\ \text{(ConseqPreFE)}$$

$$\frac{T \vdash_{\text{FE}} \{P\}\ c\ \{Q\} \qquad Q' \text{ is self-framing} \qquad Q \Longrightarrow_T Q'}{T \vdash_{\text{FE}} \{P\}\ c\ \{Q'\}}\ \text{(ConseqPostFE)}$$

Figure 2.6: The inference rules for refining pre- and postconditions of existing CSL proofs.

In the rules (IfFE) and (WhileFE), we use e to represent both the expression and its *boolean interpretation*[1]. A boolean interpretation of e is defined as: for any state ω, $e(\omega)$ is defined if and only if there exists some boolean b such that $\langle e, \omega\rangle \Downarrow_f \text{Bool}\ b$, and in this case $e(\omega) = b$. In all other cases, $e(\omega)$ is undefined. In $P \wedge e$ and $P \wedge \neg e$, we further use e (and $\neg e$) to represent the assertion derived from the boolean interpretation of e (and $\neg e$ respectively). Under the condition that e is well defined, this does not cause any issues,

[1]Another approach to avoid abnormal cases where e evaluates to non-boolean values and ensure the correctness of the rules (IfFE) and (WhileFE) other than using boolean interpretations is to ensure that e only evaluates to boolean values by a static type check. Then for well-typed programs, these rules are correct with the interpretation of expressions defined in Definition 2.15.

since e (and $\neg e$) both as the boolean interpretation of expression and as the assertion are total functions of states. However, without this condition, we should be careful about the case where e is not defined. Formally, the assertion e is true in state ω if and only if $e(\omega)$ is defined and $e(\omega) = \text{True}$. The negation of the boolean interpretation of the expression e negates its value when it is defined, but remains undefined when $e(\omega)$ is undefined. As a result, the assertion $\neg e$ is true in state ω if and only if $e(\omega)$ is defined and $e(\omega)$ is false. We can use the assertion $e \vee \neg e$ to denote that e is defined. This technique will be used when we define the translation of the front-end in the framework in Section 5.2.

Let us discuss more about the CSL rules in Figures 2.4, 2.5, and 2.6.

First, we note that the pre- and postconditions in each rule in Figure 2.4 are self-framing. Moreover, self-framedness of pre- and postconditions is preserved by each of the rules in Figures 2.5 and 2.6. As a result, the pre- and postconditions in any CSL proof are self-framing. The fact that the rules in Figures 2.5 and 2.6 preserve the self-framedness of pre- and postconditions is obvious from the premises for all these rules except the rules (IfFE), (WhileFE), (ParFE), and (FrameFE). We now analyze the four cases respectively.

- In the rule (IfFE), the condition that e is well defined w.r.t. P ensures $P = (P \wedge e) \vee (P \wedge \neg e)$. Since the normal disjunction preserves self-framedness (i.e., if A and B are both self-framing, then $A \vee B$ is also self-framing), the self-framedness of the pre- and postconditions of the conclusion can be inferred from that of the premises.
- In the rule (WhileFE), the premise "P frames e" ensures that the self-framedness of P implies the self-framedness of $P \wedge e$ and $P \wedge \neg e$. As a result, the self-framedness of the pre- and postconditions of the conclusion of the rule (WhileFE) can also be inferred from that of the premises.
- To show that the rules (ParFE) and (FrameFE) preserve the self-framedness of the pre- and postconditions, we only need to note that the separating conjunction preserves the self-framedness (i.e., if A and B are both self-framing, then $A * B$ is also self-framing).

Next, we point out that the premise "e is well defined w.r.t. P" in the rule (WhileFE) is necessary for the rule to be sound w.r.t. the front-end's operational semantics. Example 2.23 demonstrates the consequence of removing this premise.

Example 2.23 *Let x and y be two variables of type* Int*. Consider the following code snippet.*

```
1 while (x / y > 0)
2     inv True
```

```
3 do {
4     y := y − 1
5 }
```

The condition expression e corresponds to $x/y > 0$*, and the code c inside the loop is "*`y := y − 1`*". Let P be the trivial assertion* True*. Then the premise* $T \vdash_{\text{FE}} \{P \wedge e\}\ c\ \{P\}$ *trivially holds. Moreover, since e does not refer to the heap, the premise "P frames e" also holds. Therefore, if we remove the premise "e is well defined w.r.t. P" from the rule (WhileFE), then we have collected enough premises to apply the rule to this code snippet and get a CSL proof of* $T \vdash_{\text{FE}}$ $\{\text{True}\}$ **while** e **inv** True **do** $c\ \{\neg e\}$.

However, with y set to 1 *initially, after a single round execution of the loop, y will be decreased to* 0*. The next time when the program tries to check the loop condition, it will run into a divide-by-zero error. Such an execution error indicates that the modified (WhileFE)-rule is not sound any more.*

Although we express the rules (ParFE) and (FrameFE) in a different way from that presented by Vafeiadis [13], they are equivalent if we restrict them to well-typed states: we change all the requirements of the form "free variables of assertion A are disjoint from written variables of statement c" in the rules presented by Vafeiadis [13] to the form "$\forall x \in c.\, \text{indep}_T(A, x)$", but for finite written variable sets, which is the case of our front-end, the two ways of expressing the premise are equivalent when we only consider well-typed states.

Finally, compared with the consequence rule in Vafeiadis [13], in the rules (ConseqPreFE) and (ConseqPostFE), we weaken the entailment premises to only require the entailment for well-typed states. This weakening is necessary for the instantiation of the framework with our front-end in Chapter 6. As argued at the beginning of this section (Section 2.2.4), the weakening is not problematic, since we consider only the soundness of the CSL rules w.r.t. the operational semantics for the transitions of well-typed states.

2.2.5 Separating Implication and Method Call Extensions

In order to demonstrate some challenges in translating front-end programs into Viper programs in Chapter 3, we extend our front-end with method calls, and we extend its assertion language with the separating implication (also called magic wand).

A separating implication is of the form $A \mathrel{-\!\!*} B$, where A and B are two front-end assertions. The interpretation rule for $A \mathrel{-\!\!*} B$ is:

$$\frac{\forall \alpha.(\alpha \vDash_f A \wedge \alpha \oplus \omega \text{ is defined } \implies \alpha \oplus \omega \vDash_f B)}{\omega \vDash_f A \mathrel{-\!\!*} B}\ \text{(Wand)}$$

Intuitively, $\omega \vDash_f A \mathrel{-\!\!*} B$ means that adding any compatible state α that satisfies A results in the sum $\alpha \oplus \omega$ satisfying B.

A method is of the form

```
method m(x_1 : t_1, ..., x_n : t_n) returns (r : t)
    requires P
    ensures Q
    c
```

Here m is the name of the method, $x_1, \ldots, x_n$ are its arguments, with x_i having type $t_i \in T_f$ for each $i = 1, \ldots, n$. r of type $t \in T_f$ is the returned value of the method. The two front-end assertions P and Q are the pre- and postconditions of the method, respectively. Finally, the body of the method consists of a front-end statement c.

A method call is of the form $m(e_1, \ldots, e_n)$, passing the evaluation results of expressions $e_1, \ldots, e_n$ as arguments $x_1, \ldots, x_n$ to the method m and executing m's body c. For those methods that have return values, the extended front-end also supports storing the return value at some local variable, using $y := m(e_1, \ldots, e_n)$, where y does not appear in any e_i. The method guarantees that if the precondition P is satisfied when starting executing c, then the postcondition Q will be satisfied when the method call returns.

2.3 Viper

In this section, we introduce the subset of Viper that is relevant to this thesis and Viper SL in Sections 2.3.1 and 2.3.2 respectively.

2.3.1 A Subset of Viper

Unlike our front-end, Viper uses the real permission model. The basic types, values, expressions, and assertions in the subset of Viper with which this thesis is concerned are the same as those of our front-end in Section 2.2, except that all the rationals are replaced by reals. That is, the basic types of the subset of Viper are

$$T_V ::= \text{Int} \mid \text{Real} \mid \text{Bool} \mid \text{Ref}$$

They correspond to values denoted as

$$V_V ::= \text{Int } n \mid \text{Real } r \mid \text{Bool } b \mid \text{Ref } l$$

Here $n \in \mathbb{Z}$, $r \in \mathbb{R}$, $b \in \{\text{True}, \text{False}\}$, and l is either a non-null reference or the constant reference Null.

Expressions of the subset of Viper are

$$\begin{aligned} E_V ::= &\text{ Int } n \mid \text{Real } r \mid \text{Bool } b \mid \text{Ref Null} \\ &\mid x \\ &\mid \text{uop } E_V \\ &\mid E_V \text{ bop } E_V \\ &\mid E_V \,?\, E_V : E_V \\ &\mid E_V.f \end{aligned}$$

Here f is a field identifier, and the set of uop and bop supported are the same as the ones supported by the front-end in Equation (2.4).

The evaluation of these unary and binary operators are the same as in the front-end in Section 2.2.2, except that all the rationals become reals.

Similar to Definition 2.15, we use $\langle e, \omega \rangle \Downarrow_V v$ to denote that a Viper expression e is interpreted as some Viper value $v \in V_V$ in a Viper state ω. The interpretation rules of Viper expressions are the same as those of the front-end expressions in Figure 2.1, with the only exception that the rule 2.2.2 is replaced by the following rule (Real):

$$\frac{}{\langle \text{Real } r, \omega \rangle \Downarrow_V \text{ Real } r} \text{ (Real)}$$

Assertions of the subset of Viper are the same as the ones of our front-end in Equations (2.5) and (2.6), except that now the expressions and types are from E_V and T_V.

$$\begin{aligned} A_V ::= &\text{ acc}(E_V.f, E_V) \\ &\mid A_V * A_V \\ &\mid A_V \wedge A_V \\ &\mid A_V \vee A_V \\ &\mid E_V \longrightarrow A_V \\ &\mid \exists x : T_V.\, A_V \\ &\mid E_V \end{aligned}$$

Similar to Definition 2.16, we use $\omega \vDash_V A$ to denote that a Viper state ω satisfies a Viper assertion A. The rules for $\vDash_V$ are similar to those for front-end assertions in Figures 2.2 and 2.3, except that the interpretations of expressions are of Viper, and all the rationals become reals.

We include the local variable assignment "x `:=` E_V" and the heap assignment "$E_V.f$ `:=` E_V" in the subset of Viper we will use. Our subset of Viper does not support allocation of new heap locations in the form "x `:=` **`alloc`**$(f_1, \ldots, f_n)$".

Instead, as shown in Section 6.1, we will translate the front-end allocation statement "x `:=` **`alloc`**$(f_1, \ldots, f_n)$" to a sequential composition of **`havoc`** and **`inhale`** statements.

Apart from the two assignment statements, our subset of Viper also includes **`havoc`**, **`inhale`**, and **`exhale`** statements. Intuitively, statement "**`havoc`** x" for a variable x non-deterministically assigns a fresh value to x; statement "**`inhale`** A" for a Viper assertion A adds some state ω that satisfies A into the current state; statement "**`exhale`** A" for a Viper assertion A subtracts a state ω satisfying A from the current state.

For the compositions of statements, our subset of Viper only supports the sequential composition and the conditional branching.

To conclude, the subset of Viper in this thesis has the following statement constructs:

$$
\begin{aligned}
S_V ::= {} & x \texttt{ := } E_V \\
\mid {} & E_V.f \texttt{ := } E_V \\
\mid {} & \textbf{\texttt{havoc}}\ x \\
\mid {} & \textbf{\texttt{inhale}}\ A_V \\
\mid {} & \textbf{\texttt{exhale}}\ A_V \\
\mid {} & S_V;\ S_V \\
\mid {} & \textbf{\texttt{if}}\ E_V\ \textbf{\texttt{then}}\ S_V\ \textbf{\texttt{else}}\ S_V
\end{aligned}
$$

2.3.2 Viper SL

Viper has an operational semantics, which determines whether a Viper program is correct or not. The correctness of a Viper program can also be shown by a separation logic proof. As the front-end CSL, we have a set of pre-defined Viper SL rules for all primitive or composite Viper statements, and Viper SL proofs are constructed by applying these Viper SL rules finite times.

We use $T \vdash_{\text{VPR}} \{P\}\ c\ \{Q\}$ to denote that the triple $\{P\}\ c\ \{Q\}$ of Viper semantic assertions P and Q and Viper statement c is provable w.r.t. Viper type context T using Viper SL rules. Here Viper SL rules for the two assignments "x `:=` e" and "$e.f$ `:=` e'", the sequential composition "c_1; c_2", and the conditional branching "**`if`** e **`then`** c_1 **`else`** c_2" are the same as the rules (AssignVar), (AssignField), (SeqFE), and (IfFE) in the front-end, except that the evaluation of expressions is in Viper. The rules for **`havoc`**, **`inhale`**, and **`exhale`** are

shown in Figure 2.7[2,3]. In Viper, there are no structural rules as (FrameFE), (ConseqPreFE), and (ConseqPostFE) to refine the pre- and postconditions. Instead, a Viper SL proof for a Viper statement is constructed by deriving Hoare triples for each primitive statement (in our subset of Viper, they are the two assignments "x `:=` e" and "$e.f$ `:=` e'", "**`havoc`** x", "**`inhale`** A", and "**`exhale`** A") using corresponding rules for the statement, and connecting them using the rules for sequential compositions or conditional branchings. Note that for each primitive or composite statement, there is a unique Viper SL rule. This not only simplifies the construction of Viper SL proofs since the rule to apply at each step is unique, but also enables the following inversion rule for each primitive or composite Viper statement:

Lemma 2.24 (the inversion rule) *If there is a Viper SL proof for some (primitive or composite) Viper statement, then the components in the rule corresponding to this statement exist, and all the premises of this rule hold.*

As an example, if $T \vdash_{\text{VPR}} \{A\}$ **`inhale`** P $\{B\}$, then all the premises in the rule (Inhale) hold. That is, A is self-framing, A frames P, and $B = A * P$. If $T \vdash_{\text{VPR}} \{A\}$ **`if`** e **`then`** c_1 **`else`** c_2 $\{B\}$, then e is well defined w.r.t. A, and there exist Q_1 and Q_2 such that $T \vdash_{\text{VPR}} \{A \wedge e\}\ c_1\ \{Q_1\}$, $T \vdash_{\text{VPR}} \{A \wedge \neg e\}\ c_2\ \{Q_2\}$, and $B = Q_1 \vee Q_2$.

$$\frac{A \text{ is self-framing}}{T \vdash_{\text{VPR}} \{A\}\ \textbf{havoc}\ x\ \{\exists x : T(x).\ A\}}\ \text{(Havoc)}$$

$$\frac{A \text{ is self-framing} \qquad A \text{ frames } P}{T \vdash_{\text{VPR}} \{A\}\ \textbf{inhale}\ P\ \{A * P\}}\ \text{(Inhale)}$$

$$\frac{A \text{ and } B \text{ are self-framing} \qquad B \implies A * P}{T \vdash_{\text{VPR}} \{B\}\ \textbf{exhale}\ P\ \{A\}}\ \text{(Exhale)}$$

Figure 2.7: Viper SL rules for **`havoc`**, **`inhale`**, and **`exhale`**.

Remark 2.25 *As the front-end CSL rules, Viper SL rules are also parametrized by specific Viper type contexts, and reflect the Viper's operational semantics w.r.t. the same type context. As an example, consider the rule (Havoc). Let A be "*$\text{acc}(x.f, 1) \vee x = 2$*", and T be a type context where $T(x) =$ Ref. Then if we execute "*$\textbf{havoc}\ x$*"*

[2]The symbol P is overloaded in the postcondition of the rule (Inhale) and the premise "$B \implies A * P$" of the rule (Exhale). In these places, the symbol P actually represents the semantic assertion derived from the interpretation of the syntactic assertion P.

[3]The premise "A frames P" for assertions A and P in the rule (Inhale) is to ensure the self-framedness of the postcondition $A * P$. We do not use the precise definition of "A frames P" in this thesis. Instead, we use its implication that the postcondition in the rule (Inhale) is self-framing. A more general fact for Viper SL proofs is Fact 2.27.

from some state ω in which the value of x is 2 (thus $\omega \vDash_V A$, but ω is not well-typed w.r.t. the type context T), the resulting state does not satisfy the postcondition.

It has been proved that Viper SL proofs correctly and completely reflect Viper's operational semantics.

Fact 2.26 (correct Viper programs have Viper SL proofs) *If Viper program c is correct w.r.t. Viper's operational semantics for precondition P in Viper type context T, then there exists some postcondition Q such that*

$$T \vdash_{\text{VPR}} \{P\}\ c\ \{Q\}\,.$$

For Viper SL proofs, we further have the following fact:

Fact 2.27 (self-framedness of assertions in Viper SL proof) *In any Viper SL proof $T \vdash_{\text{VPR}} \{P\}\ c_v\ \{Q\}$, P and Q are self-framing.*

2.4 Example of Front-ends Translations into Viper

In this section, we describe the translations of front-end method calls and parallel compositions. They will be used in the examples in Chapter 3 to demonstrate some challenges in translating front-end programs into Viper programs.

Each front-end method is translated and verified separately in Viper. The translation translates the precondition to an **`inhale`** statement and the postcondition to an **`exhale`** statement, and puts the translation of the body statement of the front-end method between the **`inhale`** and **`exhale`** statements, as shown by the translation of the method `m1` in Figure 2.8, where tr_t, tr_a, and tr_c represent the translation of front-end types, assertions, and statements respectively.

For the method call $m(e_1, \ldots, e_n)$, the translation exhales the translated precondition $\text{tr}_\text{a}(P)$ with x_i set to e_i for $i = 1, \ldots, n$, and then inhales the translated postcondition $\text{tr}_\text{a}(Q)$ with x_i set to e_i for $i = 1, \ldots, n$, as shown in the translation of the method `c2` in Figure 2.8, where tr_e represents the translation of front-end expressions, and $A[x/e]$ for an assertion A represents the assertion obtained by substituting each unbound occurrence of x with e in A.

For y `:=` $m(e_1, \ldots, e_n)$ where y does not appear in any e_i, the translation further havocs y between the **`exhale`** and **`inhale`** statements, as shown in the translation of the method `c1` in Figure 2.8.[4]

[4]Another change compared with the translation of method `c2` is that when inhaling the postcondition in Line 10, the unbound occurrences of return variable r in the postcondition are all substituted by the local variable y.

```
1  method m1(x : t_x) returns (r : t_r)
2      requires P
3      ensures Q
4      c
5
6  method c1() {
7      y := m1(e)
8  }
9
10 method m2(x : t'_x)
11     requires P'
12     ensures Q'
13
14 method c2() {
15     m2(e')
16 }
```

```
1  method m1(x : tr_t(t_x)) returns (r : tr_t(t_r)) {
2      inhale tr_a(P);
3      tr_c(c);
4      exhale tr_a(Q)
5  }
6
7  method c1() {
8      exhale tr_a(P)[x/tr_e(e)];
9      havoc y;
10     inhale tr_a(Q)[x/tr_e(e), r/y]
11 }
12
13 method c2() {
14     exhale tr_a(P')[x/tr_e(e')];
15     inhale tr_a(Q')[x/tr_e(e')]
16 }
```

Figure 2.8: Front-end method and method calls (at the top) and their translation (at the bottom).

The parallel composition $\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$ is another typical composition of front-end statements. When Viper translates and verifies $\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$, it views c_1 and c_2 as two separate methods, and translates and verifies each c_i with pre- and postconditions P_i and Q_i for $i = 1, 2$. Finally, in the main program, it views the parallel composition block as two simultaneous method calls to c_1 and c_2, with the only difference being that the threads c_1 and c_2 in the parallel composition may modify local variables of the main program, while in a method call, the callee method has its own local variables. Therefore, the translation havocs the written variables of c_1 and c_2 between exhaling their preconditions and inhaling their postconditions. To summarize, Viper translates the parallel composition $\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$ to the following programs:

```
1 inhale tr_a(P_1);
2 tr_c(c_1);
3 exhale tr_a(Q_1)
```

```
1 inhale tr_a(P_2);
2 tr_c(c_2);
3 exhale tr_a(Q_2)
```

```
1 exhale tr_a(P_1 * P_2);
2 havoc wr(c_1) ∪ wr(c_2);
3 inhale tr_a(Q_1 * Q_2)
```

Figure 2.9: The translation of the front-end parallel composition $\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$.

Here "**havoc** S" for a finite set S of variables is an abbreviation of a sequence of "**havoc** x" for each $x \in S$. Using the inversion rule (Lemma 2.24) for **havoc**, it is straightforward to prove Proposition 2.28 by induction on the variables in S as a finite list. The proposition can be seen as the inversion rule for havocing a finite list of variables.

Proposition 2.28 (inversion rule for havocing a finite variable list) *Let S be a finite set of variables. If $T \vdash_{\text{VPR}} \{A\}$ **havoc** S $\{B\}$, then $B = \exists S : T(S).\, A$, where we abbreviate a list of consecutive existential quantifiers for the finite set S as $\exists S$, and use "$S : T(S)$" to denote that each variable $x \in S$ takes values of type $T(x)$.*

Chapter 3

Challenges

This chapter describes the two main challenges in developing the framework: the gap between the operational semantics of the front-end and Viper and the mismatches between front-end and Viper states. They will be discussed in Section 3.1 and Section 3.2 respectively.

3.1 Gap between Operational Semantics

We aim to develop a general framework for proving Viper front-end translations sound. To show the soundness of the translation from a front-end program to a Viper program, we need to prove that if the translated Viper program is correct w.r.t. Viper's operational semantics, then the front-end program is also correct w.r.t. to the front-end's operational semantics. As an example, consider the front-end program in Example 3.1.

Example 3.1 (a front-end program and its translation) *Consider the example front-end program in Figure 3.1. As introduced in Section 2.4, it is translated into three Viper programs in Figure 3.2.*

The three Viper programs all verify. As a result, each Viper program is correct w.r.t. Viper's operational semantics. Using this as the premise, what we want to show is that the front-end program in Figure 3.1 is correct w.r.t. the front-end's operational

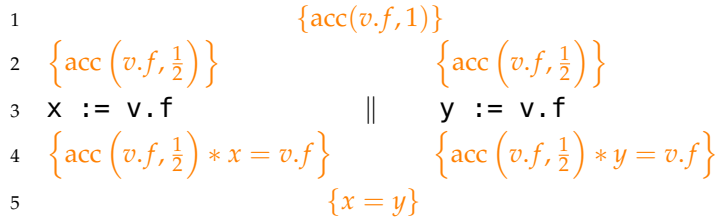

Figure 3.1: The example front-end program.

```
1 inhale acc(v.f, 1/2);
2 x := v.f;
3 exhale acc(v.f, 1/2) * x = v.f
```

```
1 inhale acc(v.f, 1/2);
2 y := v.f;
3 exhale acc(v.f, 1/2) * y = v.f
```

```
1 inhale acc(v.f, 1);
2 exhale acc(v.f, 1/2) * acc(v.f, 1/2);
3 havoc x, y;
4 inhale (acc(v.f, 1/2) * x = v.f) * (acc(v.f, 1/2) * y = v.f);
5 exhale x = y
```

Figure 3.2: The translation of the front-end program in Figure 3.1.

semantics. Concretely, we need to show that starting from any state satisfying the precondition $\mathrm{acc}(v.f, 1)$ *(i.e., holding full permission to the heap location* $v.f$*) and executing the parallel composition, the resulting state should satisfy the postcondition* $x = y$*.*

In Arlt's work [8], the soundness of the translation of the front-end is proved by directly connecting its operational semantics to Viper's operational semantics and relating the execution in Viper with that in the front-end. Despite its effectiveness, this approach is cumbersome, because the proof needs to consider and deal with lots of low-level details of the operational semantics of the two languages. Example 3.2 shows how these details might complicate the soundness proof of the translation.

Example 3.2 (low-level details complicate the proof) *Consider the front-end program (which is the right hand side thread of the front-end program in Figure 3.1) and its translation in Figure 3.3.*

Since the code verifies in Viper, there is some execution in Viper that doesn't cause any errors. In order to prove sound the translation from such a front-end to Viper, we then need to find some execution in the front-end that doesn't cause any errors either. As a result, we have to find an execution in Viper that can be converted to

```
1  {acc(v.f, 1/2)}
2  y := v.f
3  {acc(v.f, 1/2) * y = v.f}
```

```
1  inhale acc(v.f, 1/2);
2  y := v.f;
3  exhale acc(v.f, 1/2) * y = v.f
```

Figure 3.3: The front-end code (on the left) and its translation (on the right) showing how low-level details complicate the proof.

some execution in the front-end.

*However, there may be many feasible executions in Viper, and not all of them can be converted to executions in the front-end. Some executions in Viper don't correspond to any executions in the front-end because they run into some intermediate states that are not expressible in the front-end. For example, the statement "***inhale** $\mathrm{acc}(v.f, 1/2)$*" in Line 1 may add more resources, and result in some state that does not correspond to any front-end states. Specifically, starting from a state ω with no permission to $v.f$, Viper with a real permission model may add $\frac{\sqrt{2}}{2}$ permission to $v.f$, but the resulting state which has $\frac{\sqrt{2}}{2}$ permission to $v.f$ is not expressible in a front-end with a rational permission model.*

SL rules provide a way to hide these low-level details. An SL proof of a program can be constructed by syntactically applying SL rules without the need of considering semantic interpretations of these rules or the operational semantics of the programming language of the program. We can then prove sound the translation in Example 3.2 in a more elegant and much simpler way as follows:

Example 3.3 (a simpler solution to Example 3.2) *Since the Viper program in Figure 3.3 (denoted as c_v) verifies, there exists a Viper SL proof*

$$T \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ c_v\ \{\mathrm{True}\}$$

for some type context T. From this Viper SL proof for c_v, we can easily derive the following SL proof for the Viper program

$$T \vdash_{\mathrm{VPR}} \{\mathrm{acc}(v.f, 1/2)\}\ y := v.f\ \{\mathrm{acc}(v.f, 1/2) * y = v.f\}$$

for the variable assignment in Line 2. This proof can further be converted into a CSL proof of the front-end Hoare triple in Figure 3.3[1] *:*

$$T \vdash_{\mathrm{FE}} \{\mathrm{acc}(v.f, 1/2)\}\ y := v.f\ \{\mathrm{acc}(v.f, 1/2) * y = v.f\}\,.$$

[1] In this example, we ignore possible type mismatches between the front-end and Viper and assume that both the front-end program and its Viper translation program have the same type context T. The type mismatch issue will be discussed in Section 3.2.2.

Although the assertions at pre- and postconditions may have different semantics in the front-end and Viper, the conversion of Viper SL proofs to the front-end's CSL proofs is straightforward.

As a new approach, with the help of SL rules of Viper and CSL rules of front-ends, the soundness proof for front-ends' translations will then be constructed in three steps:

1. First, if the Viper program is correct w.r.t. its operational semantics, then there exists a valid SL proof for it. The existence has already been proved formally.
2. Second, convert the SL proof of a Viper program into a CSL proof of the front-end program. A general approach for dealing with this step is one of the main contributions of this thesis.
3. The soundness of a front-ends' CSL guarantees that a CSL proof of a front-end program implies the correctness of the front-end program w.r.t. the front-ends' operational semantics. We will use the CSL adjusted from Vafeiadis [13], whose soundness, as discussed in Section 2.2.4, is already proved. The focus of this thesis is therefore not on proving the CSL soundness.

Figure 3.4 shows the proof approach by Arlt [8] and our new approach.

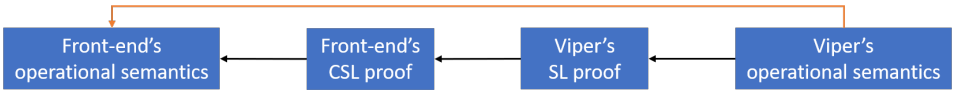

Figure 3.4: The different proof approaches by Arlt [8] and us. The orange arrow represents Arlt's approach, and the black arrows represent our three-step approach.

As an example, using the new approach, the goal in Example 3.1 then becomes:

Example (the new goal for Example 3.1) *If all of the three Viper programs in Figure 3.2 have Viper SL proofs, then the front-end triple in Figure 3.1 is provable under CSL.*

3.2 Mismatch between Front-ends' and Viper's States

Viper front-ends typically have a different state model than Viper. The difference usually appears in two aspects:

- The front-end and Viper use different permission models. Permissions are usually simpler in the front-end than in Viper, in the sense that the permission model of the front-end is a strict subset of that of Viper. For

example, as is standard in the literature, a front-end represents fractional ownership by rational permission values, while Viper represents fractional ownership by reals, as discussed in Section 2.1.1. Dealing with this gap is nontrivial, because Viper may be reasoning with permission amounts that cannot be represented in the front-end state, and while we are trying to connect the states of the front-end and Viper, it's sometimes unclear how to convert back such permission amounts into the front-end.

- Types are smaller in the front-end than in Viper. For example, a front-end with rational permission modeled states naturally has support for rational types in order to express and operate on permission values, while Viper does not have rational types, and rationals in the front-end are translated to reals in Viper.

In order for our framework to work for different front-ends, it must be able to handle these possible mismatches in a general way. Sections 3.2.1 and 3.2.2 will discuss these two mismatches in more details.

3.2.1 Permission Models Mismatch

Arlt [8] already addressed and solved this mismatch problem in the case where the front-end uses a binary ownership model for its state, while the Viper semantics supports fractional ownership of heap locations represented by rationals. The solution is to round down the fractional permissions to zero permissions for the states that are to satisfy some assertions, and round up the fractional permissions to full write permissions for the frame that is supposed to be preserved during the execution in Viper. The limited front-end assertion language ensures that there exists an execution of the translated program in Viper with the rounded-up frame and the proof can be continued by converting this execution back to an execution in the front-end. However, the solution relies on the fact that the choices for rounding up and rounding down are both unique. This makes it specific to the particular mismatch case between binary ownership and fractional ownership, and to the limited assertion language supported by the front-end. Specifically, it does not work for the mismatch between rationals and reals, where the choices for rounding up and rounding down are not unique anymore. In this project, we aim to develop a general approach, which in particular works for rationals and reals.

Example 3.4 shows possible consequences of permission mismatch between rationals and reals.[2]

[2]We demonstrate the issue using the translation of method calls in Example 3.4, but the same issue also occurs for the tranlation of parallel compositions, and the argument for the issue in parallel compositions is similar to that in method calls (which we demonstrate in Example 3.4).

Example 3.4 *Consider the front-end code template and the translation of the* `main` *method in Figure 3.5. For simplicity, we omit the "***inhale** True*" and "***exhale** True*" statements in the translation since they have no effects on the program.*

```
1  method m1(x : Ref)
2      requires P1
3      ensures True
4
5  method m2(x : Ref)
6      requires P2
7      ensures True
8
9  method main()
10     requires acc(x.f, 1)
11     ensures True
12 {
13     m1(x);
14     m2(x)
15 }
```

```
1  method main()
2  {
3      inhale acc(x.f, 1);
4      exhale tr_a(P1);
5      exhale tr_a(P2)
6  }
```

Figure 3.5: The front-end code template (on the left) and the translation of the `main` method (on the right).

Here $\mathrm{tr_a}(P_i)$ *is the translation of assertion* P_i *for* $i = 1, 2$ *and the translation of assertion* $\mathrm{acc}(x.f, 1)$ *is itself. Viper translates the precondition* $\mathrm{acc}(x.f, 1)$ *and the postcondition* True *of the* `main` *method of the front-end program as "***inhale** $\mathrm{acc}(x.f, 1)$*" and "***exhale** True*" (the latter is omitted). Between these two statements, the two method calls in the* `main` *method of the front-end program are translated into exhaling the precondition and then inhaling the postcondition (and the "***inhale** True*" statements are omitted), as we introduce in Section 2.4.*

In the following, we will construct an example revealing the unsoundness of the translation using this template.

First, assume that we pick P_1 *and* P_2 *that are equivalent to* $\mathrm{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$ *and* $\mathrm{acc}\left(x.f, 1 - \sqrt{\frac{1}{2}}\right)$ *respectively (we will construct concrete* P_1 *and* P_2 *in next step), and that* $\mathrm{tr_a}(P_i)$ *has the same form as* P_i *for* $i = 1, 2$*. In this case, Viper, which has a real permission model, is able to split the full permission to* $x.f$ *into two parts equal to* $\sqrt{\frac{1}{2}}$ *and* $1 - \sqrt{\frac{1}{2}}$ *respectively, and use the first part to execute Line 4 and*

the second to execute Line 5. Therefore, the translated program verifies. However, since $\sqrt{\frac{1}{2}}$ *is not an expressible permission amount in the front-end with a rational permission model, in order to execute the method* `m1`*, the front-end will need to give out strictly more than that amount of permission, i.e.,* $\sqrt{\frac{1}{2}} + \varepsilon$ *amount of permission, for some* $\varepsilon > 0$*. As a result of this giving-out, there is* $1 - \left(\sqrt{\frac{1}{2}} + \varepsilon\right) < 1 - \sqrt{\frac{1}{2}}$ *amount of permission to* $x.f$ *left, which is not enough for executing the method* `m2`*. Therefore, the front-end program should not verify.*

Next, we construct the assertions P_1 *and* P_2 *that are equivalent to* $\text{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$ *and* $\text{acc}\left(x.f, 1 - \sqrt{\frac{1}{2}}\right)$ *respectively and that* $\text{tr}_\text{a}(P_i)$ *has the same form as* P_i *for* $i = 1, 2$*. Since the front-end with a rational permission model may not support irrational expressions as permission amounts, we cannot directly use* $\text{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$ *and* $\text{acc}\left(x.f, 1 - \sqrt{\frac{1}{2}}\right)$ *as* P_1 *and* P_2*. Nevertheless, we can construct* P_1 *and* P_2 *that are equivalent to these two assertions, using rational expressions only, as follows:*

$$P_1 = \left(\text{acc}(v.p, 1) * v.p \geq 0 * v.p^2 < \frac{1}{2}\right) \mathrel{-\!\!*} \text{acc}(x.f, v.p),$$
$$P_2 = \left(\text{acc}(v.p, 1) * 0 \leq v.p \leq 1 * (1 - v.p)^2 > \frac{1}{2}\right) \mathrel{-\!\!*} \text{acc}(x.f, v.p).$$

Here $v.p$ *is a fresh heap location that stores rational values. Let us first explain why* P_1 *is equivalent to* $\text{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$*: Let* ω *be a state such that* $\omega \vDash_f P_1$*, then by the rule (Wand), for any* α *that is compatible with* ω *and* $\alpha \vDash_f \text{acc}(v.p, 1) * v.p \geq 0 * v.p^2 < \frac{1}{2}$*, it holds that* $\alpha \oplus \omega \vDash_f \text{acc}(x.f, v.p)$*. Note that* $\text{acc}(v.p, 1)$ *at the left hand side of "*$-\!\!*$*" requires full permission to* $v.p$ *for* α*. As a result,* α *has full control of the value of* $v.p$ *in* $\alpha \oplus \omega$*. Specifically, by the two expressions* $v.p \geq 0$ *and* $v.p^2 < \frac{1}{2}$ *connected with* $\text{acc}(v.p, 1)$ *by "*$*$*",* $v.p$ *can be any rational value in the interval* $\left[0, \sqrt{\frac{1}{2}}\right)$*, and* $\alpha \oplus \omega$ *needs to have at least* $v.p$ *amount of permission to* $x.f$ *for any such* $v.p$*. Since* α *doesn't guarantee any permission amount to* $x.f$*, the permission to* $x.f$ *must all come from* ω*. As a result, to satisfy the whole "*$-\!\!*$*",* ω *must have at least* $\sqrt{\frac{1}{2}}$ *amount of ownership to* $x.f$*. That is,* P_1 *is indeed equivalent to* $\text{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$*. Similarly,* P_2 *is equivalent to* $\text{acc}\left(x.f, 1 - \sqrt{\frac{1}{2}}\right)$*. Moreover, since "*$*$*" and "*$-\!\!*$*" are translated identically into Viper,* P_1 *and* P_2 *will be translated identically into Viper. That is,* $\text{tr}_\text{a}(P_i)$ *indeed equals* P_i *for* $i = 1, 2$*.*

The unsoundness in Example 3.4 stems from the fact that the front-end state needs more resources than the Viper state to satisfy P_1. As a result, there is not enough resource left for the front-end to satisfy P_2. We characterized this property of an assertion without which the translation will be unsound as in the above example, and named it the *backward satisfiability* property. Informally, an assertion A is backward satisfiable if for any Viper state ω_V that satisfies A, there always exists a front-end state ω_F that is smaller than or equal to ω_V and also satisfies A. Here smaller means that it holds less heap resources, in terms of permissions to each heap location and information about values at each heap location.

3.2.2 Type Mismatch

Front-ends and Viper having different state models, it is also common that they may support different types. For example, a front-end with rational permissions naturally has support for rational types in order to express and operate on permission values, while Viper does not have rational types, and directly uses reals to express rationals. As a result, the translation will translate rationals in the front-end into reals in Viper. Moreover, some front-ends may only support natural number or bounded integer types (e.g., each integer type in C/C++ has a fixed-length bit representation, and therefore only represents integers in a fixed range) and they will all be translated to the unbounded integer type of Viper. Example 3.5 and 3.6 show how the type mismatch between front-end's rationals and Viper's reals may cause incompleteness or unsoundness issues of the translation.

Example 3.5 (incompleteness) *Assume that x and y are two local variables of type rational in the front-end. As a result, in the Viper program, they will become local variables of type real. Now consider the front-end code and its translation in Figure 3.6, where the pre- and postcondtions of method* `m` *are both the trivial assertion* True, *and we omit the statements "***exhale** True*" and "***inhale** True*" in the translation since they have no effects on the program.*

```
1  {y = x}
2  x := m()
3  {x² ≠ 2}
```

```
1  inhale y = x;
2  havoc x;
3  exhale x² ≠ 2
```

Figure 3.6: The front-end code (on the left) and its translation (on the right) showing the incompleteness issue caused by type mismatches.

*Knowing that x is a rational number, the front-end program trivially verifies. However, in the translation, the statement "***havoc** x" changes x to an arbitrary real number. Without more information given about x, Viper is not able to execute*

*the statement "***exhale** $x^2 \neq 2$*". Therefore, the translated program doesn't verify, showing the incompleteness of the translation caused by type mismatches.*

Besides erasing the knowledge that the new x is a rational value in the state after *the* **havoc**, *the statement "***havoc** x*" also erases the knowledge that x is a rational value in the state* before *the* **havoc**. *As a result, in the front-end, throughout the program, we know that y equals some rational value, while in the Viper translation, we only know that y equals some real value, and lose the information that y is actually rational.*

Example 3.6 (unsoundness) *The assertion "$\exists x : \text{Rat}.\, x^2 = 2$" should always be false, but if we directly translate it into Viper without adding any constraints on the $\exists$-bounded x, we will get "$\exists x : \text{Real}.\, x^2 = 2$", which becomes true.*

*Such translation changes the semantics of the assertion, and may render the translation unsound. For example, consider the simple front-end program and its translation in Figure 3.7, where the "***assert** *A" statement in the front-end verifies whether current state satisfies A, and is translated to "***assert** $\text{tr}_a(A)$*" in Viper with $\text{tr}_a(A)$ being the translation of assertion A.*

```
1 {True}
2 assert ∃x : Rat. x^2 = 2
3 {True}
```

```
1 inhale True;
2 assert ∃x : Real. x^2 = 2;
3 exhale True
```

Figure 3.7: The front-end code (on the left) and its translation (on the right) showing the unsoundness issue caused by type mismatches.

*The front-end program does not verify since the statement "***assert** $\exists x : \text{Rat}.\, x^2 = 2$*" fails. But the Viper statement "***assert** $\exists x : \text{Real}.\, x^2 = 2$*" trivially succeeds and the translation of the front-end program in Viper verifies.*

Although the incompleteness issue in Example 3.5 doesn't affect the soundness of the translation, we still need to define the translation carefully to avoid the unsoundness issue in Example 3.6, and deal with the other effect in Example 3.5 that "**havoc** x" erases the knowledge of the front-end type of x in the state before the **havoc**.

Chapter 4

Framework Overview

This chapter provides a high-level introduction of the overall structure and the components of the general framework for proving CSL-based Viper front-end translations sound from the perspective of a user of the framework.

We start with the introduction to the framework from its overall structure, and its inputs and outputs in Section 4.1. In the next three sections, we will introduce the framework via different building blocks, where each building block relies on the previously introduced building block. In each building block, we will introduce the components of the framework that are included in this block. Specifically, Section 4.2 introduces types, values, and states, Section 4.3 introduces expressions and assertions, and Section 4.4 introduces statements.

We will describe the components inside the framework and formalize the whole framework in Chapter 5, in which we will also formally prove sound the translation of front-ends defined by the framework.

4.1 Structure, Inputs, and Outputs of the Framework

There are three kinds of components that need to be translated from a front-end to Viper: (1) the states (and as a result, the types and values), (2) the expressions and assertions, and (3) the statements. Instead of defining the framework in one step that captures the translation of each of these components, we define a separate building block for each of them, with each building block relying on the previously defined one. That is, the building block for (2) uses the building block defined for (1), and the building block for (3) uses the building block for (2).

Each of the building blocks takes inputs required to define the corresponding translation, and as the output provides the actual translation and other aspects related to the translation (such as a soundness proof).

For each building block, we additionally require a set of conditions that must be ensured for the inputs such that the building block can guarantee certain outputs (such as the soundness of the translation of statements for the building block of statements).

Concretely, the three building blocks are as follows:

1. The first building block consists of the translation for front-end types, values, and states. As shown in Figure 4.1, the building block takes as inputs the definitions of the front-end's types, values, and heaps (which is an abstraction of the permission heaps of our front-end in Section 2.2), and the translation of these components into Viper. It also takes as input a function `have_inv` in order to solve the type mismatch issue introduced in Section 3.2.2. The intuition of this function will be introduced in Section 4.2. With these inputs, the building block outputs the translation function $\mathrm{tr_s}$ for front-end states.

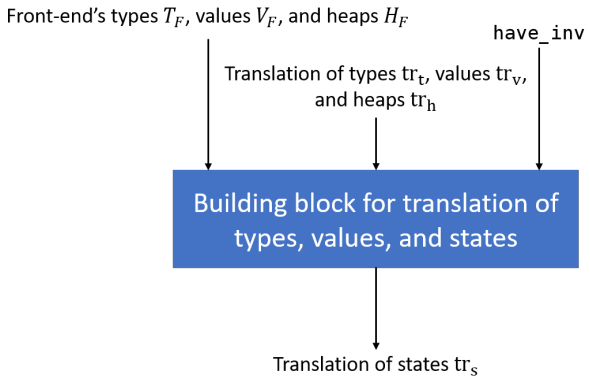

Figure 4.1: The first building block.

2. Using the building block we constructed in last step, the second building block consists of the translation for front-end expressions and assertions, as shown in Figure 4.2. Apart from the inputs described in the previous building block, this building block additionally takes as inputs the interpretation of front-end expressions[1] and primitive assertions, and the translation of these components into Viper. The building block has following constructs of assertions from primitive assertions: the separating conjunction ($*$), the normal conjunction ($\wedge$), the normal disjunction ($\vee$), the normal implication ($\longrightarrow$) with expressions as premises, the existential quantifier ($\exists$), and expressions. It will

[1] In case that the front-end types may not support booleans, the formalization of this building block actually makes a distinction between boolean expressions and general expressions, and requires the user to input the interpretation and translation for both boolean and general expressions.

inductively give the interpretation of these constructs and a translation of them into Viper.

Apart from the above basic inputs, the building block also takes as inputs a function that gives a finite free variable set for each expression and primitive assertion. The building block will compute the free variable set for any front-end assertion using this input, which will further be used to express well-formedness of front-end statements. The design of requiring the user to input the function that computes free variable sets for expressions and assertions rather than directly computing from their interpretation has two advantages: (1) it is easier to require finiteness of the computed free variable set. With user provided input function to compute free variables, we can directly add conditions on the input function requiring finiteness of the computed free variable set. However, if we remove this input and let the building block to compute the free variables from the interpretations, then it will be difficult to avoid the case where some assertions might have infinitely many free variables, which might cause unsoundness issues in the CSL rules used by the framework, and (2) it also gives flexibility to the user to overestimate free variables of expressions and assertions.

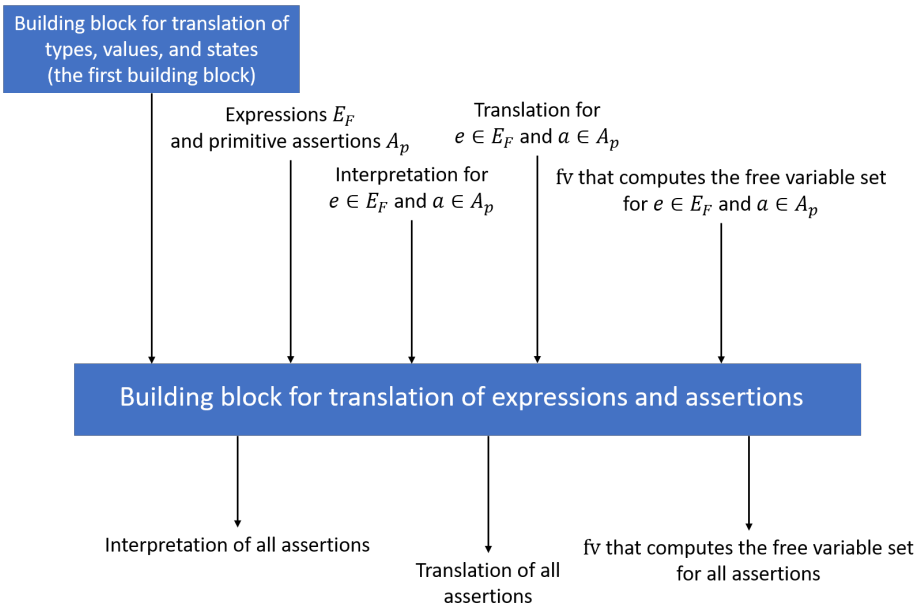

Figure 4.2: The second building block.

3. Finally, using the building block we constructed in the second step, we construct the last building block (which will be our framework) that translates the front-end statements, as shown in Figure 4.3. It

additionally takes as inputs the front-end's primitive statements, the CSL rules for them, and a translation of them.

Apart from the above basic inputs, the framework also takes as inputs two functions fv and wr that compute finite free variable sets and finite written variable set for primitive statements respectively, and a well-formedness predicate wf of primitive statements.

The well-formedness predicate wf provides an interface for the user to encode additional requirements on the front-end's primitive statements. wf additionally takes a type context as its input, which gives the variable declaration context of the program to the well-formedness check of statements. The type context input will be used in Section 4.4 when we discuss conditions for the input primitive statements.

The functions fv and wr that compute free variable sets and written variable sets for primitive statements will be used to compute the free variable set and the written variable set for each composite statement. The written variable computation is used for the translation of loops, which nondeterministically assigns values to the written variables of the loop body via **`havoc`** statements.

The framework has following constructs of statements from primitive statements: the sequential composition, the loop, the conditional branching, and the parallel composition. With all the above inputs, the framework finally outputs CSL rules and a well-formedness predicate wf for all front-end statements, a translation $\mathrm{tr_c}$ of the front-end's composite statements into Viper, and a proof of the soundness of $\mathrm{tr_c}$.

In each step, there are certain conditions that the inputs need to satisfy. We will introduce them and discuss their necessity in proving the soundness of the whole translation in the following sections. With these conditions satisfied, the framework will be able to define a translation of front-end statements to Viper, and prove the soundness of this translation.

Moreover, we require that the input translations of expressions, primitive assertions, and primitive statements are executable. As a result of this requirement and the finiteness of written variable sets for all front-end statements (which can be derived from the finiteness requirement for wr of primitive statements and the fact that the computation of written variable sets for composite statements is a finite-step induction and only includes executable operations), the translation of all all front-end statements to Viper defined by the framework will be executable.

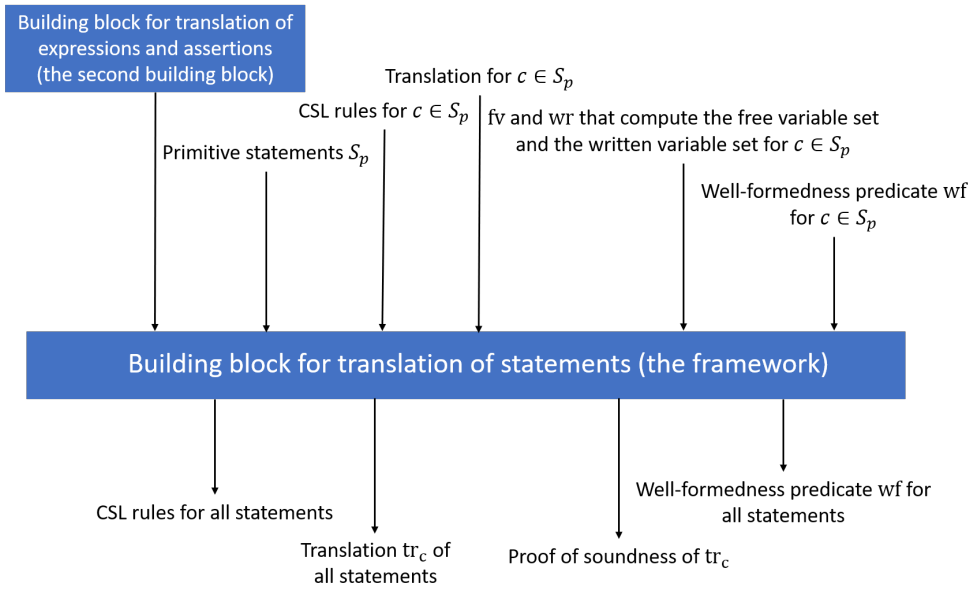

Figure 4.3: The framework built in the last step.

4.2 Types, Values, and States

As shown in Figure 4.1, in this step, the translation of front-end types, values, and states are constructed in the framework, given the definitions of the front-end's types, values, and heaps, the translation of them into Viper, and a partial function `have_inv` from the front-end's types and variable names to Viper expressions.

We will split the introduction of the conditions into two parts: (1) the ones for the translation of types and values in Section 4.2.1, and (2) the ones for the translation of heaps in Section 4.2.2.

4.2.1 Conditions for Translation of Types and Values

We use $\mathrm{tr_v}$ and $\mathrm{tr_t}$ to denote the translation functions of the front-end's values and types respectively. A basic condition for $\mathrm{tr_v}$ and $\mathrm{tr_t}$ is the correctness. Concretely,

(A1) If value v has type t in the front-end, then after the translation, the Viper value $\mathrm{tr_v}(v)$ should have Viper type $\mathrm{tr_t}(t)$.

As a simple example, type rational in our front-end in Section 2.2 is translated into the real type in Viper. Therefore, the translation of any rational value in the front-end should be of type real. Additionally, in order for the translation to not lose any type information of the front-end, the type translation and value translation inputs are required to be injective. That is,

(A2) For any front-end types t_1, t_2, $\text{tr}_\text{t}(t_1) = \text{tr}_\text{t}(t_2)$ implies $t_1 = t_2$.

(A3) For any front-end values v_1, v_2, $\text{tr}_\text{v}(v_1) = \text{tr}_\text{v}(v_2)$ implies $v_1 = v_2$.

For those front-end types of which the translation of the front-end values is not surjective, the `have_inv` provides a Viper expression assertifying whether a Viper value is a translation image of some front-end value of this front-end type. That is,

(A4) If $\texttt{have\_inv}(t)$ is defined for some front-end type t, and gives a Viper expression e for some variable x (i.e., $\texttt{have\_inv}(t)(x) = e$), then for any Viper state ω^V, $\langle e, \omega^V \rangle \Downarrow_V$ Bool True if and only if there exists some front-end value v of type t, such that the value of x in ω^V is exactly $\text{tr}_\text{v}(v)$. Otherwise, if $\texttt{have\_inv}(t)$ is not defined, then tr_v restricted to the values of type t is a bijection.

As an example, in our front-end in Section 2.2, the translation of the values of types Int, Bool, and Ref is the identity, which is bijective. But rationals are translated to reals. For this case,

$$\texttt{have\_inv}(\text{Rat})(x) \triangleq \exists a, b : \text{Int}.\, b \neq 0 \wedge x = a/b.$$

Satisfying the expression $\texttt{have\_inv}(\text{Rat})(x)$ means that the value of `x` in the current Viper state is actually rational. With the help of the expression provided by `have_inv`, the assertion "$\exists x : \text{Rat}.\, x^2 = 2$" in Example 3.6 will now be translated into "$\exists x : \text{Real}.\, (\exists a, b : \text{Int}.\, b \neq 0 \wedge x = a/b) \wedge x^2 = 2$", which preserves the semantics of the original assertion.

4.2.2 Conditions for Translation of Heaps

The heap translation function tr_h maps front-end heaps to Viper permission heaps, where both of them are modeled as separation algebras for IDF (introduced in Section 2.1.2). We can then construct the state translation function tr_s that translates front-end states to Viper states. tr_s translates the store of the front-end state by translating the value of each variable, and translates the heap of the front-end state using tr_h.

In order to preserve addition and stabilize structure and not to lose information of the input front-end's heap, tr_h is required to be homomorphic (i.e., preserving $\oplus$) and injective, and must preserve the stabilize operation. That is,

(A5) For any front-end heaps h_1, h_2, $\text{tr}_\text{h}(h_1) \# \text{tr}_\text{h}(h_2)$ if and only if $h_1 \# h_2$. When $h_1 \# h_2$, $\text{tr}_\text{h}(h_1) \oplus \text{tr}_\text{h}(h_2) = \text{tr}_\text{h}(h_1 \oplus h_2)$.

(A6) For any front-end heaps h_1, h_2, $\text{tr}_\text{h}(h_1) = \text{tr}_\text{h}(h_2)$ implies $h_1 = h_2$.

(A7) For any front-end heap h, $\text{stabilize}(\text{tr}_\text{h}(h)) = \text{tr}_\text{h}(\text{stabilize}(h))$.

In order to prove that the translation of the separating conjunction of assertions preserves the semantics given by the front-end, we require the following condition:

(A8) For any front-end heaps h_1, h_2 and Viper heap H, if $\mathrm{tr_h}(h_1) = H \oplus \mathrm{tr_h}(h_2)$, then H is an image of $\mathrm{tr_h}$. That is, there exists some front-end heap h, such that $H = \mathrm{tr_h}(h)$.[2]

Remark 4.1 *One might want to express (A8) in a simpler way as follows:*

(A8') *For any front-end heaps* h_1 *and* h_2, $\mathrm{tr_h}(h_1) \succeq \mathrm{tr_h}(h_2)$ *implies* $h_1 \succeq h_2$.

However, this is weaker than (A8). The reason is that the $\oplus$ *of a separation algebra for IDF is not like the addition of numbers. In the latter, fixing any two numbers of the equation* $a = b + c$*, the remaining number will be uniquely determined, while in the former, fixing* x *and* z *in the equation* $x = y \oplus z$ *does not uniquely determine* y*. When assuming* $\mathrm{tr_h}(h_1) = H \oplus \mathrm{tr_h}(h_2)$*, (A8) guarantees that* H *is the image of the translation of some front-end heap. On the other hand, if we substitute (A8) with (A8'), then we only get the following implication:*

$$\begin{aligned} & \mathrm{tr_h}(h_1) = H \oplus \mathrm{tr_h}(h_2) \implies \mathrm{tr_h}(h_1) \succeq \mathrm{tr_h}(h_2) \\ \overset{(A8')}{\implies} \; & h_1 \succeq h_2 \implies h_1 = h \oplus h_2 \text{ for some } h \\ \overset{(A5)}{\implies} \; & \mathrm{tr_h}(h_1) = \mathrm{tr_h}(h') \oplus \mathrm{tr_h}(h_2). \end{aligned}$$

The desired property would follow from this chained implication if $H = \mathrm{tr_h}(h')$ *holds. However, as discussed above,* $H = \mathrm{tr_h}(h')$ *is not derivable from* $\mathrm{tr_h}(h_1) = H \oplus \mathrm{tr_h}(h_2)$ *and* $\mathrm{tr_h}(h_1) = \mathrm{tr_h}(h') \oplus \mathrm{tr_h}(h_2)$.

Moreover, the following conditions are required in order to prove that normal conjunction ($\wedge$) and normal implication ($\longrightarrow$) preserve the backward satisfiability property introduced in Section 3.2.1 respectively:

(A9) For any front-end heaps h_1, h_2 and Viper heap H, if $H \succeq \mathrm{tr_h}(h_1)$ and $H \succeq \mathrm{tr_h}(h_2)$, then there exists some front-end heap h such that $h \succeq h_i$ for $i = 1, 2$ and $H \succeq \mathrm{tr_h}(h)$.

(A10) For any Viper heap H, if it is upper bounded by the translation of some front-end heap h_b, i.e., $\mathrm{tr_h}(h_b) \succeq H$, then its core is the translation of some front-end heap. That is, there exists some front-end heap h such that $\mathrm{tr_h}(h) = |H|$.

Intuitively, (A9) expresses that the front-end heaps are closed under the operation of taking supremum. (A10) is weaker than requiring that $\mathrm{tr_h}$ preserves $|\cdot|$ (which will further require h being the core of some front-end heap), but is sufficient for our use.

[2]This condition cannot be implied from the homomorphism condition (A5). The reason is that in (A5), both heap arguments for the Viper addition are translated from front-end, while this is not guaranteed by the premise $\mathrm{tr_h}(h_1) = H \oplus \mathrm{tr_h}(h_2)$ of the condition (A8).

4.3 Expressions and Assertions

As shown in Figure 4.2, in this step, the interpretation, the translation, and a free variable set computing function of front-end expressions and primitive assertions are given, and an interpretation, a translation, and a free variable set computing function for all front-end assertions are output.

Let $\mathrm{tr_e}$ and $\mathrm{tr_a}$ represent the translations of expressions and assertions respectively, I_e^F and I_a^F represent the interpretations of expressions and assertions in the front-end respectively, and I_e^V and I_a^V represent the interpretations of expressions and assertions in Viper respectively. The interpretation of expressions is a partial function mapping from states to values. The interpretation of assertions is a total function from states to boolean values.

The basic property we want for $\mathrm{tr_e}$ and $\mathrm{tr_a}$ is that the semantics of the expressions and assertions in the front-end is preserved in Viper. We express this semantics-preserving property for $\mathrm{tr_e}$ as the following condition:

(A11) For any front-end expression e and state ω^F, $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega^F))$ is defined if and only if $I_e^F(e, \omega^F)$ is defined. When they are both defined, $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega^F)) = \mathrm{tr_v}(I_e^F(e, \omega^F))$.

In order to get the semantics-preserving property for $\mathrm{tr_a}$, we will conduct an induction on the construction of the front-end assertions. The base case of the induction is the following condition:

(A12) For any front-end primitive assertion A and state ω^F, $\mathrm{tr_a}(A)$ is satisfied at $\mathrm{tr_s}(\omega^F)$ if and only if A is satisfied at ω^F, i.e., $I_a^V(\mathrm{tr_a}(A), \mathrm{tr_s}(\omega^F)) = I_a^F(A, \omega^F)$.

As discussed in Section 3.2.1, we require the backward satisfiability of assertions for the translation to be sound. We will prove this property for all front-end assertions inductively on the construction of the assertion. The base case of the induction is the following condition:

(A13) All front-end primitive assertions are backward satisfiable.

In the induction step for separating conjunctions in proving the semantics-preserving property of $\mathrm{tr_a}$ (Lemma 5.24), we additionally require monotonicity of the translated assertions. Informally, an assertion A is monotone, if ω satisfies A implies that any state larger than ω also satisfies A.

The proof of monotonicity of translated assertions is also inductive, with the base case expressed as the following condition:

(A14) For any front-end primitive assertion A, $\mathrm{tr_a}(A)$ is monotone.

The induction step for normal implication additionally requires that the translation of all front-end expressions should have a monotone interpretation. Concretely,

(A15) For any front-end expression e and Viper state ω^V, if $I_e^V(\mathrm{tr_e}(e), \omega^V)$ is defined and evaluates to some value v, then $I_e^V(\mathrm{tr_e}(e), \varphi^V)$ is defined and evaluates to the same v for any Viper state φ^V for which $\varphi^V \succeq \omega^V$.

Moreover, the following condition that the translation of any front-end expression is pure (i.e., the evaluation of the translated expression only depends on the core of the state) is required for the induction step of normal implication in proving backward satisfiability of front-end assertions:

(A16) For any front-end expression e and Viper state ω^V, $I_e^V(\mathrm{tr_e}(e), \omega^V)$ is defined if and only if $I_e^V(\mathrm{tr_e}(e), |\omega^V|)$ is defined. When they are both defined, their values should be equal.

Remark 4.2 *When $I_e^V(\mathrm{tr_e}(e), |\omega^V|)$ is defined, using the order relation $\omega^V \succeq |\omega^V|$ from (C1), (A16) becomes a special case of (A15). However, when $I_e^V(\mathrm{tr_e}(e), |\omega^V|)$ is not defined, we cannot derive that $I_e^V(\mathrm{tr_e}(e), \omega^V)$ is neither defined from (A15) and $\omega^V \succeq |\omega^V|$, which is why we need the condition (A16).*

Finally, for the user-provided function that computes free variable sets of expressions and primitive assertions, we require that it indeed characterizes (or overestimates) the free variables of an expression or a primitive assertion. That is, for any variable x that is not in the set, changing its value should not affect the evaluation of the expression or the assertion:

(A17) If x is not a free variable of an expression e, then for any front-end state ω^F and any front-end value v, $I_e^F(e, \omega^F) = I_e^F(e, \omega^F(x \mapsto v))$, where the notation $\omega^F(x \mapsto v)$ is defined in Definition 2.12.

(A18) If x is not a free variable of an primitive assertion A, then for any front-end state ω^F and any front-end value v, $I_a^F(A, \omega^F) = I_a^F(A, \omega^F(x \mapsto v))$.

4.4 The Statements

As shown in Figure 4.3, in the last step of the construction, the framework takes as inputs a well-formedness predicate wf, a free variable set computing function fv, a written variable set computing function wr, a translation function $\mathrm{tr_c}$, and a set of CSL rules for front-end primitive statements. It then outputs a well-formedness predicate wf and CSL rules for all front-end statements, as well as a translation function $\mathrm{tr_c}$ for all front-end statements and a proof of soundness of $\mathrm{tr_c}$.

We use $T \vdash_p \{P\}\ c\ \{Q\}$ to denote a CSL rule input by the user for primitive statement c and pre- and postconditions P and Q w.r.t. type context T. A basic condition for the translation of primitive statements is its correctness w.r.t. front-end CSL rules and Viper SL rules. Concretely, for a specific front-end type context T, let $\mathrm{tr_{tc}}(T)$ be its translation to Viper (the type context translation function $\mathrm{tr_{tc}}$ simply translates the type of each variable

to Viper), P and Q be two Viper semantic assertions, and c be a front-end primitive statement that is well-formed w.r.t. T (denoted as $\mathrm{wf}_T(c)$), if $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr}_{\mathrm{c}}(c)\ \{Q\}$, then $T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$, where btr is the natural backward translation of semantic Viper assertions. Concretely, for a semantic Viper assertion R, $\mathrm{btr}(R)$ is a semantic front-end's assertion whose representation set consists of the front-end states whose translation images satisfy R. The actual condition we make in the framework is weaker than the one above, in the sense that it only requires correctness in the case where P and Q are monotone:

(A19) For any front-end type context T, front-end primitive statement c, and any monotone semantic Viper assertions P and Q, if $\mathrm{wf}_T(c)$ and $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr}_{\mathrm{c}}(c)\ \{Q\}$, then $T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$.

The reason why we only consider the case where the pre- and post-conditions are monotone is that we will only deal with monotone assertions throughout the soundness proof of the translation in the framework.

For the free variable set computing function fv and the written variable set computing function wr, since we only use them in describing syntactic side conditions of CSL's parallel rule, the framework does not have any requirements for them that are related to the operational semantics of front-end statements, with the only exception that the well-formedness predicate needs to check the consistency of the written variables with the type context. Concretely, they have to be declared at the beginning of the program. The following condition shows this requirement in the view of type contexts:

(A20) For any primitive statement c that is well-formed w.r.t. type context T (i.e., $\mathrm{wf}_T(c)$) and any variable $x \in \mathrm{wr}(c)$, $T(x)$ should be defined.

Chapter 5

Formalization of the Framework

In this chapter, we will formalize the whole framework that we introduced in Chapter 4. Since the input of the framework only contains part of the front-end, the framework will first make requirements of the remaining part of the front-end. We describe these requirements in Section 5.1. We then define the executable translation and the well-formedness requirements for front-end statements in Sections 5.2 and 5.3, respectively. Before formally stating and proving the soundness of the translation, we extend the conditions on heaps in Section 4.2.2 to those on states in Section 5.4. Finally, we prove the soundness of the translation in Section 5.5.

5.1 The Requirements of the Front-end

As shown in Figure 4.2, the input of the framework provides a set of expressions E_F and primitive assertions A_p. Then the framework has following constructs of assertions from primitive assertions: the separating conjunction ($*$), the normal conjunction ($\wedge$), the normal disjunction ($\vee$), the normal implication ($\longrightarrow$) with expressions as premises, the existential quantifier ($\exists$), and expressions. Formally, the composite assertions in the framework have the following form, which is exactly the same as the compositions of assertions shown in Equation (2.6) of our front-end in Section 2.2.

$$
\begin{aligned}
A_F ::= {} & A_p \\
\mid {} & A_F * A_F \\
\mid {} & A_F \wedge A_F \\
\mid {} & A_F \vee A_F \\
\mid {} & E_F \longrightarrow A_F \\
\mid {} & \exists x : T_F.\, A_F \\
\mid {} & E_F
\end{aligned}
$$

Here T_F denotes the front-end types, as shown in Figure 4.1.

With the input of the framework providing the set of primitive statements S_p as shown in Figure 4.3, the framework has following constructs of statements from the primitive statements: the sequential composition, the conditional branching, the loop, and the parallel composition. Formally, the composite statements in the framework have the following form, which is exactly tthe same as the compositions of statements shown in Equation (2.8) of our front-end in Section 2.2.

$$\begin{aligned} S_F ::= & \; S_p \\ & \mid S_F; S_F \\ & \mid \textbf{if } E_F \textbf{ then } S_F \textbf{ else } S_F \\ & \mid \textbf{while } E_F \textbf{ inv } A_F \textbf{ do } S_F \\ & \mid \{A_F\}\, S_F\, \{A_F\} \parallel \{A_F\}\, S_F\, \{A_F\} \end{aligned}$$

We use $\omega \vDash_F A$ to denote that front-end state ω satisfies front-end assertion A in our framework. The interpretation of expressions and primitive assertions are input into the framework as I_e^F for general expressions and I_a^F for primitive assertions. Therefore, for primitive assertion A, $\omega \vDash_F A$ if and only if $I_a^F(A, \omega) = \text{True}$. The framework then requires the front-end to interpret the compositions of assertions inductively as follows:

- $\omega \vDash_F A * B$ if and only if there exist front-end states α and β such that $\alpha \oplus \beta = \omega$, $\alpha \vDash_F A$, and $\beta \vDash_F B$.
- $\omega \vDash_F A \wedge B$ if and only if $\omega \vDash_F A$ and $\omega \vDash_F B$.
- $\omega \vDash_F A \vee B$ if and only if $\omega \vDash_F A$ or $\omega \vDash_F B$.
- $\omega \vDash_F e \longrightarrow A$ if and only if $I_e^F(e, \omega)$ is defined and equals some boolean value b, and if $b = \text{True}$ then $\omega \vDash_F A$.
- $\omega \vDash_F \exists x : t.\, A$ if and only if there exists some front-end value v of type t, such that $\omega(x \mapsto v) \vDash_F A$, where the notation $\omega(x \mapsto v)$ is defined in Definition 2.12.
- For expression e, $\omega \vDash_F e$ if and only if $I_e^F(e, \omega)$ is defined and equals boolean True.

Note that these requirements are equivalent to the inductive interpretation rules in Figure 2.3 of our front-end in Section 2.2.

As shown in Figures 4.2 and 4.3, our framework takes as input a function fv to give the free variables of expressions and primitive statements, and also a function wr for written variables of primitive statements. Given these functions, the framework computes free variables and written variables for composite statements as follows, which will be used in defining the CSL rules.

Definition 5.1 (free variables and written variables) *The free variable set* fv *of front-end statements is computed inductively as follows:*

- *For primitive statement* c, $\mathrm{fv}(c)$ *is given by the input of the framework.*
- $\mathrm{fv}(c_1; c_2) = \mathrm{fv}(c_1) \cup \mathrm{fv}(c_2)$.
- $\mathrm{fv}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) = \mathrm{fv}(e) \cup \mathrm{fv}(c_1) \cup \mathrm{fv}(c_2)$.
- $\mathrm{fv}(\textbf{while } e \textbf{ inv } I \textbf{ do } c) = \mathrm{fv}(e) \cup \mathrm{fv}(c)$.
- $\mathrm{fv}(\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}) = \mathrm{fv}(c_1) \cup \mathrm{fv}(c_2)$.

The written variable set wr *of front-end statements is computed inductively as follows:*

- *For primitive statement* c, $\mathrm{wr}(c)$ *is given by the input of the framework.*
- $\mathrm{wr}(c_1; c_2) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2)$.
- $\mathrm{wr}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2)$.
- $\mathrm{wr}(\textbf{while } e \textbf{ inv } I \textbf{ do } c) = \mathrm{wr}(c)$.
- $\mathrm{wr}(\{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}) = \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2)$.

Note that the inductive computation of fv and wr above is the same as the inductive computation of free variables and written variables of composite statements in Equations (2.12) and (2.14) for our front-end in Section 2.2.

Finally, taking as input a set of CSL rules for primitive statements (as shown in Figure 4.3), the framework defines CSL rules as follows:

Definition 5.2 (CSL rules) *We use* $T \vdash_{\mathrm{CSL}} \{P\}\ c\ \{Q\}$ *to denote that the triple* $\{P\}\ c\ \{Q\}$ *is CSL-provable w.r.t. front-end type context* T*, where* P *and* Q *are front-end semantic assertions and* c *is a front-end statement.*

The inference rules of $\vdash_{\mathrm{CSL}}$ *are given in Figure 5.1.*

As we did in the rules (IfFE) and (WhileFE) of our front-end in Section 2.2.4, here in the rules (If) and (While) we also use e to represent both the expression and its boolean interpretation[1]. Then except the rule (Prim) which is for deriving CSL proofs for primitive statements from the input of the framework, the remaining rules are the same as the corresponding rules in Figures 2.5 and 2.6 of our front-end in Section 2.2.

[1] In the Isabelle formalization of our framework, since we make a distinction between general expressions and boolean expressions, the condition expression e in "**if** e **then** c_1 **else** c_2" and "**while** e **inv** I **do** c" is required to be a boolean expression, and the symbol e appeared at the positions of pre- and postconditions in the rules (If) and (While) represents the interpretation of the boolean expression e (i.e., a partial function mapping states to boolean values).

$$\frac{T \vdash_p \{P\}\ c\ \{Q\}}{T \vdash_{\text{CSL}} \{P\}\ c\ \{Q\}}\ \text{(Prim)}$$

$$\frac{T \vdash_{\text{CSL}} \{P\}\ c_1\ \{R\} \qquad T \vdash_{\text{CSL}} \{R\}\ c_2\ \{Q\}}{T \vdash_{\text{CSL}} \{P\}\ c_1; c_2\ \{Q\}}\ \text{(Seq)}$$

$$\frac{\begin{array}{c}e \text{ is well defined w.r.t. } P \\ T \vdash_{\text{CSL}} \{P \wedge e\}\ c_1\ \{Q_1\} \qquad T \vdash_{\text{CSL}} \{P \wedge \neg e\}\ c_2\ \{Q_2\}\end{array}}{T \vdash_{\text{CSL}} \{P\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q_1 \vee Q_2\}}\ \text{(If)}$$

$$\frac{T \vdash_{\text{CSL}} \{P \wedge e\}\ c\ \{P\} \qquad P \text{ frames } e \qquad e \text{ is well defined w.r.t. } P}{T \vdash_{\text{CSL}} \{P\}\ \texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c\ \{P \wedge \neg e\}}\ \text{(While)}$$

$$\frac{\begin{array}{c}T \vdash_{\text{CSL}} \{A_1\}\ c_1\ \{B_1\} \qquad T \vdash_{\text{CSL}} \{A_2\}\ c_2\ \{B_2\} \\ \text{fv}(c_1) \cap \text{wr}(c_2) = \varnothing \qquad \text{fv}(c_2) \cap \text{wr}(c_1) = \varnothing \\ \forall x \in \text{wr}(c_2).\, \text{indep}_T(A_1, x) \text{ and } \text{indep}_T(B_1, x) \\ \forall x \in \text{wr}(c_1).\, \text{indep}_T(A_2, x) \text{ and } \text{indep}_T(B_2, x)\end{array}}{T \vdash_{\text{CSL}} \{A_1 * A_2\}\ \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}\ \{B_1 * B_2\}}\ \text{(Par)}$$

$$\frac{\begin{array}{c}T \vdash_{\text{CSL}} \{P\}\ c\ \{Q\} \\ R \text{ is self-framing} \qquad \forall x \in \text{wr}(c).\, \text{indep}_T(R, x)\end{array}}{T \vdash_{\text{CSL}} \{P * R\}\ c\ \{Q * R\}}\ \text{(Frame)}$$

$$\frac{T \vdash_{\text{CSL}} \{P\}\ c\ \{Q\} \qquad P' \text{ is self-framing} \qquad P' \Longrightarrow_T P}{T \vdash_{\text{CSL}} \{P'\}\ c\ \{Q\}}\ \text{(ConseqPre)}$$

$$\frac{T \vdash_{\text{CSL}} \{P\}\ c\ \{Q\} \qquad Q' \text{ is self-framing} \qquad Q \Longrightarrow_T Q'}{T \vdash_{\text{CSL}} \{P\}\ c\ \{Q'\}}\ \text{(ConseqPost)}$$

Figure 5.1: CSL rules of the front-end.

5.2 The Executable Translation

We now formally define the translation from the front-end to Viper. The translation of expressions and primitive assertions is given by the input of the framework as tr_e and tr_a. So we are left to define the translation of assertions and statements.

Definition 5.3 (translation of assertions) *The translation of assertions* tr_a *is defined inductively as follows:*

- *For primitive assertion* A, $\text{tr}_\text{a}(A)$ *is provided by the input.*
- $\text{tr}_\text{a}(A * B) = \text{tr}_\text{a}(A) * \text{tr}_\text{a}(B)$.

- $\mathrm{tr_a}(A \wedge B) = \mathrm{tr_a}(A) \wedge \mathrm{tr_a}(B)$.
- $\mathrm{tr_a}(A \vee B) = \mathrm{tr_a}(A) \vee \mathrm{tr_a}(B)$.
- $\mathrm{tr_a}(e \longrightarrow A) = \mathrm{tr_e}(e) \longrightarrow \mathrm{tr_a}(A)$.
- *In order to translate* $\exists x : t.\, A$*, we need to consider two cases:*
  1. *The translation of the values of type* t *is a bijection, i.e.,* $\texttt{have\_inv}(t)$ *is undefined. Then* $\mathrm{tr_a}(\exists x : t.\, A) = \exists x : \mathrm{tr_t}(t).\, \mathrm{tr_a}(A)$.
  2. *The translation of the values of type* t *is not surjective, i.e., when* $\texttt{have\_inv}(t)$ *is defined. Let* $\texttt{have\_inv}(t)(x) = e$ *for variable* x*, then* $\mathrm{tr_a}(\exists x : t.\, A) = \exists x : \mathrm{tr_t}(t).\, (e \wedge \mathrm{tr_a}(A))$.
- *For front-end expression* e, $\mathrm{tr_a}(e) = \mathrm{tr_e}(e)$.

Let's discuss in more details about the translation of $\exists x : t.\, A$.

We first show that if $\omega \vDash_V \mathrm{tr_a}(\exists x : t.\, A)$, then there exists some front-end value v of type t such that $\omega(x \mapsto \mathrm{tr_v}(v)) \vDash_V \mathrm{tr_a}(A)$:

- In the case where the translation of values of type t is a bijection, if $\omega \vDash_V \mathrm{tr_a}(\exists x : t.\, A)$, then by the interpretation of $\exists$ in Viper (which is the same as the rule (Exist) in our front-end in Section 2.2), there exists some Viper value v' of type $\mathrm{tr_t}(t)$, such that $\omega(x \mapsto v') \vDash_V \mathrm{tr_a}(A)$. Moreover, the bijectivity ensures that there exists some front-end value v of type t such that $\mathrm{tr_v}(v) = v'$.
- In the case where the translation of the values of type t is not surjective, if $\omega \vDash_V \mathrm{tr_a}(\exists x : t.\, A)$, then there exists some Viper value v' of type $\mathrm{tr_t}(t)$, such that $\omega(x \mapsto v') \vDash_V e \wedge \mathrm{tr_a}(A)$, where $e = \texttt{have\_inv}(t)(x)$. That is, $\omega(x \mapsto v') \vDash_V \mathrm{tr_a}(A)$, and $\langle e, \omega(x \mapsto v') \rangle \Downarrow_V$ Bool True. From the framework condition (A4) in Section 4.2.1, the value of variable x in state $\omega(x \mapsto v')$, which is exactly v', is the image of some front-end value v of type t (i.e., $v' = \mathrm{tr_v}(v)$).

Conversely, if there exists some front-end value v of type t such that $\omega(x \mapsto \mathrm{tr_v}(v)) \vDash_V \mathrm{tr_a}(A)$, then since $\mathrm{tr_v}(v)$ has type $\mathrm{tr_t}(t)$ by the framework condition (A1) in Section 4.2.1, no matter if the translation of the values of type t is bijective, it is straightforward to show that ω satisfies $\mathrm{tr_a}(\exists x : t.\, A)$.

To summarize, we have the following result:

Lemma 5.4 (satisfaction of $\mathrm{tr_a}(\exists x : t.\, A)$) *Viper state* ω *satisfies* $\mathrm{tr_a}(\exists x : t.\, A)$ *if and only if there exists some front-end value* v *of type* t *such that* $\omega(x \mapsto \mathrm{tr_v}(v))$ *satisfies* $\mathrm{tr_a}(A)$.

The translation function for statements consists of two parts. The first part $\mathrm{tr_c^M}$ maps a front-end statement to a main Viper program. The second part

tr_c^E maps a front-end statement to a finite set of Viper programs. Both parts are computed inductively, as shown in Definition 5.5.

Definition 5.5 (translation of statements) tr_c^M *is computed inductively as follows:*

- *For primitive statement* c, $\mathrm{tr}_c^M(c)$ *is provided by the input.*
- $\mathrm{tr}_c^M(c_1;\, c_2) = \mathrm{tr}_c^M(c_1);\, \mathrm{tr}_c^M(c_2)$.
- $\mathrm{tr}_c^M(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) = \texttt{if}\ \mathrm{tr}_e(e)\ \texttt{then}\ \mathrm{tr}_c^M(c_1)\ \texttt{else}\ \mathrm{tr}_c^M(c_2)$.
- *For loop* $c_w = \texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c$,

$$\begin{aligned}\mathrm{tr}_c^M(c_w) = {}& \texttt{exhale}\ \mathrm{tr}_a(I) \wedge (\mathrm{tr}_e(e) \vee \neg \mathrm{tr}_e(e));\\ & \texttt{havoc}\ \mathrm{wr}(c);\\ & \texttt{inhale}\ \mathrm{tr}_a(I) \wedge \neg \mathrm{tr}_e(e).\end{aligned}$$

- *For parallel composition* $c = \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$,

$$\begin{aligned}\mathrm{tr}_c^M(c) = {}& \texttt{exhale}\ \mathrm{tr}_a(P_1 * P_2);\\ & \texttt{havoc}\ \mathrm{wr}(c);\\ & \texttt{inhale}\ \mathrm{tr}_a(Q_1 * Q_2).\end{aligned}$$

tr_c^E *is computed inductively as follows:*

- *For primitive statement* c, $\mathrm{tr}_c^E(c) = \varnothing$.
- $\mathrm{tr}_c^E(c_1;\, c_2) = \mathrm{tr}_c^E(c_1) \cup \mathrm{tr}_c^E(c_2)$.
- $\mathrm{tr}_c^E(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) = \mathrm{tr}_c^E(c_1) \cup \mathrm{tr}_c^E(c_2)$.
- *For loop* $c_w = \texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c$,

$$\begin{aligned}\mathrm{tr}_c^E(c_w) = {}& \mathrm{tr}_c^E(c) \cup\\ & \{\texttt{inhale}\ \mathrm{tr}_a(I) \wedge \mathrm{tr}_e(e);\\ & \ \mathrm{tr}_c^M(c);\\ & \ \texttt{exhale}\ \mathrm{tr}_a(I) \wedge (\mathrm{tr}_e(e) \vee \neg \mathrm{tr}_e(e))\}.\end{aligned}$$

- *For parallel composition* $c = \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$,

$$\begin{aligned}\mathrm{tr}_c^E(c) = {}& \mathrm{tr}_c^E(c_1) \cup \mathrm{tr}_c^E(c_2) \cup\\ & \{\texttt{inhale}\ \mathrm{tr}_a(P_1);\, \mathrm{tr}_c^M(c_1);\, \texttt{exhale}\ \mathrm{tr}_a(Q_1),\\ & \ \texttt{inhale}\ \mathrm{tr}_a(P_2);\, \mathrm{tr}_c^M(c_2);\, \texttt{exhale}\ \mathrm{tr}_a(Q_2)\}\end{aligned}$$

Let's motivate the translation of loops and parallel compositions. In both cases, we split the verification into multiple parts. The main program of the translation tr_c^M of loops and parallel compositions abstracts over the precise execution of the loop body and the threads by using the user-specified assertions. Then, the tr_c^E part of the translation checks that the execution of the loop body and the threads respects the user-specified assertions.

Concretely, tr_c^M abstracts the execution of the loop body and the threads as a process of giving out the resource the excution acquires, choosing fresh values for the variables the execution modifies, and then getting back the resource the execution promises to return. These three steps are implemented as **`exhale`**, **`havoc`**, and **`inhale`** statements, respectively.

The tr_c^E of the loop translates the inner code into a separate program, gives it the resource that must hold to enter the loop as the precondition, and requires the invariant as the postcondition to hold after a single round of execution of the inner code[2]. That is, we verify each round of the loop by verifying tr_c^E of the loop. The tr_c^E of the parallel composition translates each thread with the user-specified pre- and post-conditions into a separate program. As a result, we verify each thread separately by verifying tr_c^E of the parallel composition.

Note that in the translation of loops, we **`exhale`** not only the translation of I, but also $\text{tr}_e(e) \lor \neg\text{tr}_{ee}(e)$ in tr_c^M and tr_c^E. As mentioned and discussed in Section 2.2.4, this is a technique to to check that $tr_e(e)$ is defined. With such technique, the main program of the translation of Example 2.23 will be

```
1 exhale x/y > 0 ∨ ¬(x/y > 0);
2 havoc y;
3 inhale ¬(x/y > 0)
```

Since both $(x/y > 0)$ and $\neg(x/y > 0)$ imply $y \neq 0$, which is not guaranteed, the "**`exhale`** $x/y > 0 \lor \neg(x/y > 0)$" in Line 1 cannot verify.

5.3 Well-formedness of Front-end Programs

Recall that our goal is to show that if the translation of a front-end statement c has a Viper SL proof using Viper SL rules described in Section 2.3, then there is a CSL proof for c using CSL rules described in Figure 5.1 in Section 5.1. However, as shown in Example 5.6, the Viper SL proof is not sufficient to generate a CSL proof. As a result, we need further well-formedness properties on the front-end program.

[2]Since the loop body is translated to a separate Viper program in tr_c^E of the loop, local variables are initialized to arbitrary well-typed values (rather than inherits the values of the variables in the state before entering the loop in the main program) in our verification of this program. As a result, when specifying the invariant, the user needs to include the information of all the local variables that will be used in the loop.

Example 5.6 (Viper SL proof is not sufficient for a CSL proof) *Consider the example front-end statement and its translation (with the "***inhale** True*" and "***exhale** True*" statements omitted since they have no effects on the programs) in Figure 5.2.*

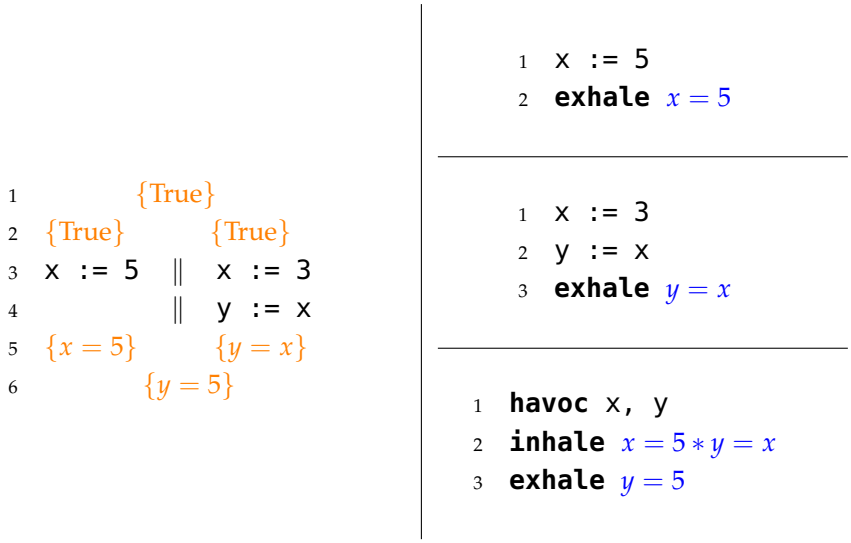

Figure 5.2: The front-end statement (at left) and its translation (at right).

As defined in Definition 5.5, the front-end parallel composite statement is translated into three Viper programs. It is straightforward to see that all the three programs have Viper SL proofs. However, we cannot apply the rule (Par) to the front-end statement because the free variable set of the left hand side thread (which is $\{x\}$*) is not disjoint from the written variable set of the right hand side thread (which is* $\{x, y\}$*). Moreover, the front-end statement does not verify against its postcondition* $y = 5$ *because both threads of the parallel composition assign to variable* x*, which causes a data race.*

We now start to define the well-formedness requirements for front-end statements. Besides the well-formedness requirement of each separate statement, some compositions (specifically, the loop and the parallel composition) also generate extra well-formedness requirements. These extra requirements are required for the premises of the corresponding CSL rules (the rules (While) and (Par)) and cannot be derived from Viper SL proof of the translation. Some of these requirements need to be checked semantically, while the others can be checked syntactically. In order to make more requirements syntactically checkable, we require the user to provide as inputs of the framework a function that computes free variables for expressions and primitive assertions. We use this function to compute free variables of assertions, and further use the function that computes free variables for assertions to define parts of the extra requirements of parallel compositions.

Concretely, with the function fv that computes free variable sets for expressions and primitive assertions provided as the input of the framework, we compute free variable sets for composite front-end assertions inductively as follows (which is the same as the inductive computation of free variables of composite assertions in Equation (2.10) for our front-end in Section 2.2):

- For primitive assertion A, $\mathrm{fv}(A)$ is provided by the input of the framework.
- $\mathrm{fv}(A * B) = \mathrm{fv}(A \wedge B) = \mathrm{fv}(A \vee B) = \mathrm{fv}(A) \cup \mathrm{fv}(B)$.
- $\mathrm{fv}(e \longrightarrow A) = \mathrm{fv}(e) \cup \mathrm{fv}(A)$.
- $\mathrm{fv}(\exists x : t.\, A) = \mathrm{fv}(A) \backslash \{x\}$.
- For expression e, $\mathrm{fv}(e)$ is also provided by the input of the framework.

Using the framework conditions (A17) and (A18) in Section 4.3, it is straightforward to show by structural induction on assertions that the above computation indeeds characterizes free variables for assertions.

Lemma 5.7 (free variables of assertions) *For any front-end type syntactic assertion A, and $x \notin \mathrm{fv}(A)$, $\mathrm{indep}_T(A, x)$ holds for any type context T.*

Now we can define the well-formedness requirements for each front-end composition of statements.

Definition 5.8 (well-formedness of statements) *We define well-formedness of statements w.r.t. type context T (denoted as wf_T) inductively as follows:*

- *For primitive statement c, $\mathrm{wf}_T(c)$ is provided by the input.*
- $\mathrm{wf}_T(c_1; c_2)$ *if and only if* $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
- $\mathrm{wf}_T(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)$ *if and only if* $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
- $\mathrm{wf}_T(\texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c)$ *if and only if* $\mathrm{wf}_T(c)$, *I is self-framing, and I frames e.*
- $\mathrm{wf}_T(\{P_1\}\ c_1\ \{Q_1\} \parallel \{P_2\}\ c_2\ \{Q_2\})$ *requires the following four conditions:*
  1. $\mathrm{wf}_T(c_1)$ *and* $\mathrm{wf}_T(c_2)$.
  2. *All the pre- and post-conditions P_1, P_2, Q_1, and Q_2 are self-framing.*
  3. $\mathrm{wr}(c_1)$ *is disjoint from* $\mathrm{fv}(c_2)$, $\mathrm{fv}(P_2)$, *and* $\mathrm{fv}(Q_2)$.
  4. $\mathrm{wr}(c_2)$ *is disjoint from* $\mathrm{fv}(c_1)$, $\mathrm{fv}(P_1)$, *and* $\mathrm{fv}(Q_1)$.

Note that the inductive definition of wf above is the same as the inductive definition of well-formedness of composite statements in Definition 2.21 for our front-end in Section 2.2.

Finally, we note that that in Definition 5.8, the extra well-formedness requirements for the loop (i.e., I is self-framing and I frames e) need to be checked semantically. The self-framedness requirement of P_1, Q_1, P_2, and Q_2 of the extra well-formedness requirements for the parallel composition need to be checked semantically as well. However, with the free variables of assertions and statements and the written variables of statements computable, the last two extra requirements (i.e., $\mathrm{wr}(c_1)$ is disjoint from $\mathrm{fv}(c_2)$, $\mathrm{fv}(P_2)$, and $\mathrm{fv}(Q_2)$, and $\mathrm{wr}(c_2)$ is disjoint from $\mathrm{fv}(c_1)$, $\mathrm{fv}(P_1)$, and $\mathrm{fv}(Q_1)$) are syntactically checkable.

5.4 Translation of Front-end States

As described in Section 4.2.2, the translation of states $\mathrm{tr_s}$ operates separately on the store and the heap. For the store part, we translate the value of each variable. For the heap part, we use the heap translation function $\mathrm{tr_h}$ that is the input of the framework.

With the instantiation of states as separation algebras for IDF in Section 2.1.2, it is straightforward to derive the following properties of the translation of states using the conditions on the translation of heaps in Section 4.2.2.

Proposition 5.9 (properties of $\mathrm{tr_s}$) *The following results hold:*

(L1) *For any front-end states ω_1, ω_2, $\mathrm{tr_s}(\omega_1) \# \mathrm{tr_s}(\omega_2)$ if and only if $\omega_1 \# \omega_2$. When $\omega_1 \# \omega_2$, $\mathrm{tr_s}(\omega_1) \oplus \mathrm{tr_s}(\omega_2) = \mathrm{tr_s}(\omega_1 \oplus \omega_2)$.*

(L2) *For any front-end states ω_1, ω_2, $\mathrm{tr_s}(\omega_1) = \mathrm{tr_s}(\omega_2)$ implies $\omega_1 = \omega_2$.*

(L3) *For any front-end state ω, $\mathrm{stabilize}(\mathrm{tr_s}(\omega)) = \mathrm{tr_s}(\mathrm{stabilize}(\omega))$.*

(L4) *For any front-end states ω_1, ω_2 and Viper state ω_v, if $\mathrm{tr_s}(\omega_1) = \omega_v \oplus \mathrm{tr_s}(\omega_2)$, then ω_v is an image of $\mathrm{tr_s}$. That is, there exists some front-end heap ω_f, such that $\omega_v = \mathrm{tr_s}(\omega_f)$.*

(L5) *For any front-end states ω_1, ω_2 and Viper state ω_v, if $\omega_v \succeq \mathrm{tr_s}(\omega_1)$ and $\omega_v \succeq \mathrm{tr_s}(\omega_2)$, then there exists some front-end state ω_f such that $\omega_f \succeq \omega_i$ for $i = 1, 2$ and $\omega_v \succeq \mathrm{tr_s}(\omega_f)$.*

(L6) *For any Viper state ω_v, if it is upper bounded by the translation of some front-end state ω_b, i.e., $\mathrm{tr_s}(\omega_b) \succeq \omega_v$, then its core is the translation of some front-end state. That is, there exists some front-end state ω_f such that $\mathrm{tr_s}(\omega_f) = |\omega_v|$.*

Proof Note that all the operators on states ($\mathrm{tr_s}$, $\#$, $\oplus$, $\succeq$, core, and stabilize) operate on the store and the heap selarately. As a result, we only need to prove the results for the store part and then combine them with the results for the heap part that come from the conditions in Section 4.2.2.

Concretely, for the stores, because core and stabilize are identity mappings, # and $\succeq$ are both equivalent to equality, and their translation tr_σ is injective (implied by the condition (A3) that the translation of each variable's value is injective), the following results are trivial:

(R1) For any front-end stores σ_1, σ_2, $\mathrm{tr}_\sigma(\sigma_1) \# \mathrm{tr}_\sigma(\sigma_2)$ if and only if $\sigma_1 \# \sigma_2$. When $\sigma_1 \# \sigma_2$, $\mathrm{tr}_\sigma(\sigma_1) \oplus \mathrm{tr}_\sigma(\sigma_2) = \mathrm{tr}_\sigma(\sigma_1 \oplus \sigma_2)$.

(R2) For any front-end stores σ_1, σ_2, $\mathrm{tr}_\sigma(\sigma_1) = \mathrm{tr}_\sigma(\sigma_2)$ implies $\sigma_1 = \sigma_2$.

(R3) For any front-end store σ, $\mathrm{stabilize}(\mathrm{tr}_\sigma(\sigma)) = \mathrm{tr}_\sigma(\mathrm{stabilize}(\sigma))$.

(R4) For any front-end stores σ_1, σ_2 and Viper store σ_v, if $\mathrm{tr}_\sigma(\sigma_1) = \sigma_v \oplus \mathrm{tr}_\sigma(\sigma_2)$, then σ_v is an image of tr_σ. That is, there exists some front-end heap σ_f, such that $\sigma_v = \mathrm{tr}_\sigma(\sigma_f)$.

(R5) For any front-end stores σ_1, σ_2 and Viper store σ_v, if $\sigma_v \succeq \mathrm{tr}_\sigma(\sigma_1)$ and $\sigma_v \succeq \mathrm{tr}_\sigma(\sigma_2)$, then there exists some front-end store σ_f such that $\sigma_f \succeq \sigma_i$ for $i = 1, 2$ and $\sigma_v \succeq \mathrm{tr}_\sigma(\sigma_f)$.

(R6) For any Viper store σ_v, if it is upper bounded by the translation of some front-end store σ_b, i.e., $\mathrm{tr}_\sigma(\sigma_b) \succeq \sigma_v$, then its core is the translation of some front-end store. That is, there exists some front-end store σ_f such that $\mathrm{tr}_\sigma(\sigma_f) = |\sigma_v|$.

We then prove the results in the proposition one by one.

(L1): Let the store and the heap of state ω_i be σ_i and h_i respectively for $i = 1, 2$. Then $\mathrm{tr_s}(\omega_1) \# \mathrm{tr_s}(\omega_2) \iff \mathrm{tr}_\sigma(\sigma_1) \# \mathrm{tr}_\sigma(\sigma_2)$ and $\mathrm{tr_h}(h_1) \# \mathrm{tr_h}(h_2) \iff \sigma_1 \# \sigma_2$ and $h_1 \# h_2 \iff \omega_1 \# \omega_2$. When $\omega_1 \# \omega_2$,

$$
\begin{aligned}
& \mathrm{tr_s}(\omega_1) \oplus \mathrm{tr_s}(\omega_2) \\
= {} & (\mathrm{tr}_\sigma(\sigma_1), \mathrm{tr_h}(h_1)) \oplus (\mathrm{tr}_\sigma(\sigma_2), \mathrm{tr_h}(h_2)) \\
= {} & (\mathrm{tr}_\sigma(\sigma_1) \oplus \mathrm{tr}_\sigma(\sigma_2), \mathrm{tr_h}(h_1) \oplus \mathrm{tr_h}(h_2)) \\
= {} & (\mathrm{tr}_\sigma(\sigma_1 \oplus \sigma_2), \mathrm{tr_h}(h_1 \oplus h_2)) \qquad \text{(by (R1) and (A5))} \\
= {} & \mathrm{tr_s}(\sigma_1 \oplus \sigma_2, h_1 \oplus h_2) \\
= {} & \mathrm{tr_s}((\sigma_1, h_1) \oplus (\sigma_2, h_2)) \\
= {} & \mathrm{tr_s}(\omega_1 \oplus \omega_2).
\end{aligned}
$$

(L2): Let $\omega_i = (\sigma_i, h_i)$ for $i = 1, 2$. Then

$$
\begin{aligned}
& \mathrm{tr_s}(\omega_1) = \mathrm{tr_s}(\omega_2) \\
\implies {} & \mathrm{tr}_\sigma(\sigma_1) = \mathrm{tr}_\sigma(\sigma_2) \wedge \mathrm{tr_h}(h_1) = \mathrm{tr_h}(h_2) \\
\implies {} & \sigma_1 = \sigma_2 \wedge h_1 = h_2 \qquad \text{(by (R2) and (A6))} \\
\implies {} & \omega_1 = \omega_2.
\end{aligned}
$$

(L3): Let $\omega = (\sigma, h)$. Then

$$\begin{aligned}
&\text{stabilize}(\text{tr}_\text{s}(\omega)) \\
&= \text{stabilize}(\text{tr}_\sigma(\sigma), \text{tr}_\text{h}(h)) \\
&= (\text{stabilize}(\text{tr}_\sigma(\sigma)), \text{stabilize}(\text{tr}_\text{h}(h))) \\
&= (\text{tr}_\sigma(\text{stabilize}(\sigma)), \text{tr}_\text{h}(\text{stabilize}(h))) \qquad \text{(by (R3) and (A7))} \\
&= \text{tr}_\text{s}(\text{stabilize}(\omega)).
\end{aligned}$$

(L4): Let $\omega_i = (\sigma_i, h_i)$ for $i = 1, 2$ and $\omega_v = (\sigma_v, h_v)$. From $\text{tr}_\text{s}(\omega_1) = \omega_v \oplus \text{tr}_\text{s}(\omega_2)$ we know $\text{tr}_\sigma(\sigma_1) = \sigma_v \oplus \text{tr}_\sigma(\sigma_2)$ and $\text{tr}_\text{h}(h_1) = h_v \oplus \text{tr}_\text{h}(h_2)$. Then we can get σ_f and h_f satisfying $\text{tr}_\sigma(\sigma_f) = \sigma_v$ and $\text{tr}_\text{h}(h_f) = h_v$ using (R4) and (A8), respectively. Combining σ_f and h_f, we get the required $\omega_f = (\sigma_f, h_f)$ that satisfies $\text{tr}_\text{s}(\omega_f) = \omega_v$.

(L5): Let $\omega_i = (\sigma_i, h_i)$ for $i = 1, 2$ and $\omega_v = (\sigma_v, h_v)$. From $\omega_v \succeq \text{tr}_\text{s}(\omega_i)$ we know $\sigma_v \succeq \text{tr}_\sigma(\sigma_i)$ and $h_v \succeq \text{tr}_\text{h}(h_i)$ for $i = 1, 2$. As a result, using (R5) and (A9), we get σ_f and h_f satisfying: $\sigma_v \succeq \text{tr}_\sigma(\sigma_f)$, $\sigma_f \succeq \sigma_i$ for $i = 1, 2$, $h_v \succeq \text{tr}_\text{h}(h_f)$, and $h_f \succeq h_i$ for $i = 1, 2$. Combining σ_f and h_f, we immediately get the $\omega_f = (\sigma_f, h_f)$ that satisfies all the requirements.

(L6): Let $\omega_b = (\sigma_b, h_b)$ and $\omega_v = (\sigma_v, h_v)$. From $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ we know $\text{tr}_\sigma(\sigma_b) \succeq \sigma_v$ and $\text{tr}_\text{h}(h_b) \succeq h_v$. Then using (R6) and (A10), we can get σ_f and h_f with $\text{tr}_\sigma(\sigma_f) = |\sigma_v|$ and $\text{tr}_\text{h}(h_f) = |h_v|$. As a result, the combination $\omega_f = (\sigma_f, h_f)$ satisfies $\text{tr}_\text{s}(\omega_f) = (\text{tr}_\sigma(\sigma_f), \text{tr}_\text{h}(h_f)) = (|\sigma_v|, |h_v|) = |\omega_v|$. □

5.5 Soundness Proof of the Translation

Now that we have described the requirements of front-end and defined the translation, we are getting to the core of the framework: the soundness proof of the translation, which will be formally displayed in this section. As discussed in Section 3.1, the whole proof is constructed in three parts and our framework deals with the second one, i.e., converting the SL proof of the translated Viper program into a CSL proof of the front-end program. To be more concrete, the conversion works for well-formed front-end programs with user-given front-end pre- and post-conditions (these are syntactic assertions). And since the translation consists of multiple Viper programs, we require all of them to have SL proofs, where the main program has pre- and post-conditions translated from the front-end program, and each of the other programs has no precondition and some existentially quantified postcondition. This is formalized in the following definition:

Definition 5.10 (SL proof of translation) *A front-end statement c with front-end syntactic pre- and post-conditions P and Q is* SL provable *in Viper w.r.t. front-end type context T if:*

- $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P_v\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c)\ \{Q_v\}$, *where* P_v *and* Q_v *are the semantic interpretation of* $\mathrm{tr}_{\mathrm{a}}(P)$ *and* $\mathrm{tr}_{\mathrm{a}}(Q)$ *respectively, and*
- *for each Viper program* $c^E \in \mathrm{tr}_{\mathrm{c}}^{E}(c)$*, there exists some semantic Viper assertion* Q^E *as the postcondition such that* $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ c^E\ \{Q^E\}$.

Then our goal can be formalized by the following theorem:

Theorem 5.11 (soundness of translation w.r.t. SL and CSL proofs) *Let c be a front-end statement, T be a front-end type context, and P and Q be front-end syntactic assertions. Let P_f and Q_f be the front-end semantic assertions that are the interpretation of P and Q respectively. If c is well-formed w.r.t. T, and c with precondition P and postcondition Q is SL provable in Viper w.r.t. T, then $T \vdash_{\mathrm{CSL}} \{P_f\}\ c\ \{Q_f\}$.*

The rest of this section is devoted to proving this theorem. We first describe the proof strategy and the main challenge in Section 5.5.1. Next, in Section 5.5.2, we introduce some necessary definitions and proofs for resolving the challenge. After the preparation, in Section 5.5.3, we propose and prove the frame backward translation lemma that resolves the challenge. Finally, we finish the proof of Theorem 5.11 in Section 5.5.4.

5.5.1 The Proof Strategy and Main Challenge

To prove the main result (Theorem 5.11), we want to use structural induction on front-end statements. However, the theorem does not yield a helpful induction hypothesis. For example, in the sequential composition case, the Viper SL proof produces an intermediate Viper semantic assertion R as the postcondition of the first part of the program and the precondition of the second part of the program. Yet the induction hypothesis is not strong enough to make use of R, since it requires that both pre- and postconditions of the Viper SL proof are translated from front-end syntactic assertions. As a result, we need to extend the concept "SL provable" to a wider range of pre- and post-conditions and strengthen the theorem.

Definition 5.12 (SL provability with Viper semantic assertions) *Let c be a front-end statement, P_v and Q_v be two Viper semantic assertions, and T be a front-end type context. c with pre- and postconditions P_v and Q_v is* Viper annotated SL provable *w.r.t. T if:*

- $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P_v\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c)\ \{Q_v\}$, *and*
- *for each Viper program* $c^E \in \mathrm{tr}_{\mathrm{c}}^{E}(c)$*, there exists some semantic Viper assertion* Q^E *as the postcondition such that* $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ c^E\ \{Q^E\}$.

If we replace the premise "c with precondition P and postcondition Q is SL provable in Viper w.r.t. T" with "c with precondition P_v and postcondition Q_v is Viper annotated SL provable w.r.t. T" in Theorem 5.11 where P_v and

Q_v are the semantic interpretation of $\mathrm{tr_a}(P)$ and $\mathrm{tr_a}(Q)$, then the front-end semantic assertions P_f and Q_f in the conclusion "$T \vdash_{\mathrm{CSL}} \{P_f\}\ c\ \{Q_f\}$" will be the backward translation of P_v and Q_v. Here backward translation of Viper semantic assertions is defined as follows:

Definition 5.13 (backward translation of Viper semantic assertions) *Let A be a Viper semantic assertion. The backward translation of A is a front-end semantic assertion, denoted as $\mathrm{btr}(A)$ and defined as: for any front-end state ω, ω satisfies $\mathrm{btr}(A)$ if and only if $\mathrm{tr_s}(\omega)$ satisfies A.*

A possible extension of Theorem 5.11 is then: if c is well-formed, and c with pre- and postconditions P and Q is Viper annotated SL provable w.r.t. T, then $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$. However, this extension which permits all possible Viper semantic assertions as P and Q turns out to be too strong to prove. Luckily, the translation of front-end assertions results in monotone Viper assertions, and we only need to strengthen Theorem 5.11 to Viper semantic pre- and postconditions P and Q that are monotone.

In the following, we define the monotonicity of assertions formally.

Definition 5.14 (monotonicity of assertions) *A semantic assertion (for Viper or a front-end) is monotone if, if a state ω satifies the assertion, then any larger state (w.r.t. the order of the states as a separation algebra for IDF) also satisfies the assertion.*

We can now state a stronger version of the main theorem (Theorem 5.11):

Theorem 5.15 (strengthened soundness theorem) *Let c be a front-end statement, T be a front-end type context, and P_v and Q_v be two Viper semantic assertions. If*

(P1) *P_v and Q_v are monotone.*

(P2) *c is well-formed w.r.t. T.*

(P3) *c with P_v and Q_v is Viper annotated SL provable w.r.t. T.*

Then $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P_v)\}\ c\ \{\mathrm{btr}(Q_v)\}$.

We will prove Theorem 5.15 by induction on the structure of the statement c. Now the induction hypothesis just requires monotonicity of the pre- and postconditions of the Viper SL proof, which makes the proof of the theorem possible (e.g., in the sequential composition case where the Viper SL proof produces an intermediate Viper semantic assertion R, now we only need to make R monotone to satisfy the requirements of the induction hypothesis).

In the induction proof of Theorem 5.15, the main challenge lies in dealing with the loop and the parallel composition cases. In particular, we need to fill the gap of pre- and postconditions between the front-end triple derived from the

rule (While) or (Par) and the front-end triple in $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$ for which we want to derive a CSL proof. For the following, we describe the challenge in the parallel composition case. The challenge is analogous for the loop case.

Challenge Let T be a front-end type context and $c = \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$ be a front-end parallel composite statement. For simplicity, we assume that c does not modify any local variables. Therefore, the main program of the translation is given by

$$\text{tr}_{\text{c}}^{M}(c) = (\textbf{exhale}\ \text{tr}_{\text{a}}(P_1 * P_2);\ \textbf{inhale}\ \text{tr}_{\text{a}}(Q_1 * Q_2))\,.$$

We want to prove $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$, where P and Q are two monotone Viper semantic assertions. From the theorem's premises and the induction hypothesis, we can prove the following facts:

- a Viper SL proof of $\text{tr}_{\text{c}}^{M}(c)$ of the form

$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{P\}\ \textbf{exhale}\ \text{tr}_{\text{a}}(P_1 * P_2);\ \textbf{inhale}\ \text{tr}_{\text{a}}(Q_1 * Q_2)\ \{Q\}\,. \quad (5.1)$$

- $T \vdash_{\text{CSL}} \{P_i\}\ c_i\ \{Q_i\}$[3] for $i = 1, 2$ by applying the induction hypothesis and using the Viper SL proofs of the programs in $\text{tr}_{\text{c}}^{E}(c)$ (the detailed derivation will be shown in Section 5.5.4). From these CSL proofs of c_1 and c_2 and using the rule (Par), we can further derive

$$T \vdash_{\text{CSL}} \{P_1 * P_2\}\ c\ \{Q_1 * Q_2\}\,. \quad (5.2)$$

Let's analyze the challenge in this situation from the view of the operational semantics of the front-end and Viper.

From the Viper SL proof in Equation (5.1), we know that the execution of "**exhale** $\text{tr}_{\text{a}}(P_1 * P_2)$; **inhale** $\text{tr}_{\text{a}}(Q_1 * Q_2)$" from any Viper state ω_v satisfying P will terminate at some state satisfying Q without any errors. In order to execute **exhale** $\text{tr}_{\text{a}}(P_1 * P_2)$, Viper needs to split ω_v into three parts: one to satisfy $\text{tr}_{\text{a}}(P_1)$, one to satisfy $\text{tr}_{\text{a}}(P_2)$ (recall that $\text{tr}_{\text{a}}(P_1 * P_2) = \text{tr}_{\text{a}}(P_1) * \text{tr}_{\text{a}}(P_2)$), and the other remains as the frame after the **exhale** statement.

On the other hand, by the soundness of the CSL rules w.r.t. the front-end's operational semantics, if we could prove $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$, then starting from any front-end state ω_f satisfying $\text{btr}(P)$ and executing the front-end statement c would terminate at some state satisfying $\text{btr}(Q)$ without any errors. From the CSL proof in Equation (5.2) we know that executing c

[3]Since the pre- and postconditions of a CSL proof are semantic assertions, the symbols P_i and Q_i in the pre- and postconditions of this CSL proof actually represent the interpretation of the user-specified syntactic assertions P_i and Q_i.

requires $P_1 * P_2$. Therefore, the front-end needs a similar split of ω_f as the above split of ω_v in Viper. Concretely, the front-end needs to split ω_f into three parts: one to satisfy P_1, one to satisfy P_2, and the other remains as the frame that will not be modified throughout the execution of the parallel composition c.

However, the existence of the split of the state in Viper does not imply that in the front-end. As shown in Example 3.4 in Section 3.2.1, with the permission model mismatch between the front-end and Viper, it is possible that Viper is able to split the state while the front-end is not. This issue makes the desired $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$ unprovable, and as shown in Example 3.4, it indeed causes the translation to be unsound.

The next two sections will be devoted to the definitions and proofs that resolve this issue.

5.5.2 Preparation for Proving the Frame Backward Translation Lemma

As introduced in Section 3.2.1, the backward satisfiability is the key property that we require to avoid the unsoundness issue that appears because of the permission mismatch in the challenge in Section 5.5.1.

Definition 5.16 (backward satisfiability) *A front-end syntactic assertion A is* backward satisfiable *if: for any Viper state ω_v which is not larger than the translation of some front-end state ω_b, if $\omega_v \vDash_V \text{tr}_\text{a}(A)$, then there exists some front-end state ω_f, such that $\omega_f \vDash_F A$ and $\omega_v \succeq \text{tr}_\text{s}(\omega_f)$.*

The backward satisfiability of the assertion $P_1 * P_2$ will be the key to resolve the challenge in Section 5.5.1. Since it can be any syntactic assertion in the front-end, we need to prove that all the front-end syntactic assertions are backward satisfiable. We will prove by structural induction on the assertions. Since the normal conjunction and normal implication cases additionally require monotonicity of the assertions, we first prove that all the front-end syntactic assertions have a monotone interpretation.

Lemma 5.17 (monotone assertion interpretation) *All front-end syntactic assertions have a monotone interpretation.*

Proof We again use structural induction on the front-end syntactic assertion A.

Base Case A_p: A is a primitive assertion.

> Assume that $\omega \vDash_F A$ and $\omega' \succeq \omega$. Then by (A12), $\text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A)$. By (L1) and the definition of $\succeq$, $\text{tr}_\text{s}(\omega') \succeq \text{tr}_\text{s}(\omega)$. Using (A14), we know that $\text{tr}_\text{s}(\omega') \vDash_V \text{tr}_\text{a}(A)$ as well. Finally, using (A12) again gives us the fact that $\omega' \vDash_F A$.

Base Case E_F: A is an expression e.

Assume that $\omega \vDash_F e$ and $\omega' \succeq \omega$. This means that $I_e^F(e, \omega)$ is defined and evaluates to boolean value True. Then by (A11), $\mathrm{tr_e}(e)$ also evaluates to True at state $\mathrm{tr_s}(\omega)$[4]. Moreover, by (L1) and the definition of $\succeq$, $\mathrm{tr_s}(\omega') \succeq \mathrm{tr_s}(\omega)$. As a result of (A15), we know that $\mathrm{tr_e}(e)$ evaluates to True at $\mathrm{tr_s}(\omega')$ as well. Finally, by (A11) again, we get that $I_e^F(e, \omega') = \text{True}$, which further implies that $\omega' \vDash_F e$ as well.

Inductive Case $*$: $A = A_1 * A_2$, with the induction hypothesis being that A_1 and A_2 both have monotone interpretations.

Assume that $\omega \vDash_F A_1 * A_2$ and $\omega' \succeq \omega$. Then there exist α and β such that $\alpha \oplus \beta = \omega$, $\alpha \vDash_F A_1$, and $\beta \vDash_F A_2$. By the definition of $\succeq$, we can find δ such that $\omega' = \omega \oplus \delta$. By associativity of $\oplus$, we get that $\omega' = (\alpha \oplus \beta) \oplus \delta = \alpha \oplus (\beta \oplus \delta)$. Since $\beta \oplus \delta \succeq \beta$, by monotonicity of A_2's interpretation and $\beta \vDash_F A_2$, we know that $\beta \oplus \delta \vDash_F A_2$. Combining with the facts that $\alpha \vDash_F A_1$ and that $\omega' = \alpha \oplus (\beta \oplus \delta)$, we immediately know that $\omega' \vDash_F A_1 * A_2$.

Remark 5.18 *In the proof of this case, we only use the monotonicity of the interpretation of A_2. Actually, a monotone interpretation of either A_1 or A_2 is sufficient to prove this case.*

Inductive Case $\wedge$: $A = A_1 \wedge A_2$, with the induction hypothesis being that A_1 and A_2 both have monotone interpretations.

Assume that $\omega \vDash_F A_1 \wedge A_2$ and $\omega' \succeq \omega$. Then ω satisfies both A_1 and A_2. By the induction hypothesis, ω' also satisfies both A_1 and A_2. As a result, $\omega' \vDash_F A_1 \wedge A_2$.

Inductive Case $\vee$: $A = A_1 \vee A_2$, with the induction hypothesis being that A_1 and A_2 both have monotone interpretations.

Assume that $\omega \vDash_F A_1 \vee A_2$ and $\omega' \succeq \omega$. Then ω satisfies either A_1 or A_2. Without loss of generality, assume $\omega \vDash_F A_1$. Then by the induction hypothesis that A_1 has a monotone interpretation, $\omega' \vDash_F A_1$. Therefore, $\omega' \vDash_F A_1 \vee A_2$.

Inductive Case $\longrightarrow$: $A = e \longrightarrow A_0$, with the induction hypothesis being that A_0 has a monotone interpretation.

[4]Here we assume that front-end boolean value b is translated identically to Viper boolean value b. This assumption is not required if we make a distinction between general expressions and boolean expressions. For example, in the Isabelle formalization of our framework, e here is a boolean expression, and the semantics-preserving condition of the framework for boolean expressions requires that front-end boolean expression e_b evaluates to some boolean value b at front-end state ω if and only if $\mathrm{tr_e}(e_b)$ evaluates to the same b at Viper state $\mathrm{tr_s}(\omega)$. As a result, the fact that the boolean expression e here evaluates True at ω immediately implies that $\mathrm{tr_e}(e)$ evaluates to True at $\mathrm{tr_s}(\omega)$.

Assume that $\omega \vDash_F e \longrightarrow A_0$ and $\omega' \succeq \omega$. Then we know that $I_e^F(e, \omega)$ is defined and that if $I_e^F(e, \omega) = \text{True}$ then $\omega \vDash_F A_0$. Using the same argument as in the "**Base Case** E", we can prove that $I_e^F(e, \omega')$ is also defined, and $I_e^F(e, \omega') = I_e^F(e, \omega)$. As a result, if $I_e^F(e, \omega') = \text{True}$, we still know that $\omega \vDash_F A_0$, which further implies $\omega' \vDash_F A_0$, by the monotone interpretation of A_0 guaranteed by the induction hypothesis. Combining these facts, we finally know that $\omega' \vDash_F e \longrightarrow A_0$.

Inductive Case $\exists$**:** $A = \exists x : t.\, A_0$, with the induction hypothesis being that A_0 has a monotone interpretation.

Assume that $\omega \vDash_F \exists x : t.\, A_0$ and $\omega' \succeq \omega$. Then there exists some value v of type t, such that $\omega(x \mapsto v) \vDash_F A_0$. Note that $\omega' \succeq \omega$ implies that they have an equal store, and ω' has a larger heap than ω. Therefore, $\omega'(x \mapsto v)$ has the same store as $\omega(x \mapsto v)$. Since the modification of variable x does not change the heap. The order relation between the heaps of ω' and ω is preserved by $\omega'(x \mapsto v)$ and $\omega(x \mapsto v)$. As a result, $\omega'(x \mapsto v) \succeq \omega(x \mapsto v)$. Since A_0 has a monotone interpretation by the induction hypothesis, $\omega'(x \mapsto v) \vDash_F A_0$. Therefore, $\omega' \vDash_F \exists x : t.\, A_0$. $\square$

Now we prove that all the front-end syntactic assertions are backward satisfiable.

Lemma 5.19 (backward satisfiability) *All front-end syntactic assertions are backward satisfiable.*

Proof Use structural induction on the front-end syntactic assertion A.

Base Case A_p**:** A is a primitive assertion.

The goal is exactly what the condition (A13) states.

Base Case E**:** A is an expression e.

Fix some front-end state ω_b and Viper state ω_v with $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and $I_e^V(\text{tr}_\text{e}(e), \omega_v) = \text{Bool True}$. Then by (L6), we know there exists some front-end state ω_f with $\text{tr}_\text{s}(\omega_f) = |\omega_v|$. By the axiom (C1) of PCM with cores, $\omega_v \succeq \text{tr}_\text{s}(\omega_f)$. Moreover, by the condition (A16), $I_e^V(\text{tr}_\text{e}(e), |\omega_v|) = I_e^V(\text{tr}_\text{e}(e), \omega_v) = \text{Bool True}$. Finally, by the condition (A11), $I_e^F(e, \omega_f) = \text{True}$, which further implies that $\omega_f \vDash_F e$.

Inductive Case $*$**:** $A = A_1 * A_2$, with the induction hypothesis being that A_1 and A_2 are both backward satisfiable. In this case and the "**Inductive Case** $\longrightarrow$", we will also use the following fact about separation algebras for IDF which is not hard to prove.

Fact 5.20 (compatibility for smaller elements) *For a separation algebra for IDF P and $x, y \in P$, if $x' \# y'$ for $x' \succeq x$ and $y' \succeq y$, then $x \# y$ and $x' \oplus y' \succeq x \oplus y$.*

Fix some front-end state ω_b and Viper state ω_v with $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and that $\omega_v \vDash_V \text{tr}_\text{a}(A_1 * A_2) = \text{tr}_\text{a}(A_1) * \text{tr}_\text{a}(A_2)$. Then there exist α_v and β_v such that $\alpha_v \oplus \beta_v = \omega_v$, $\alpha_v \vDash_V \text{tr}_\text{a}(A_1)$, and $\beta_v \vDash_V \text{tr}_\text{a}(A_2)$. Note that $\omega_v \succeq \alpha_v$. By the transitivity of $\succeq$ as a partial order, we know $\text{tr}_\text{s}(\omega_b) \succeq \alpha_v$, which allows us to use the induction hypothesis that A_1 is backward satisfiable to get a front-end state α_f with $\alpha_v \succeq \text{tr}_\text{s}(\alpha_f)$ and $\alpha_f \vDash_F A_1$. Using the same argument, we get β_f with $\beta_v \succeq \text{tr}_\text{s}(\beta_f)$ and $\beta_f \vDash_F A_2$. Then, as a result of Fact 5.20, we get $\text{tr}_\text{s}(\alpha_f) \# \text{tr}_\text{s}(\beta_f)$ from $\alpha_v \# \beta_v$, and $\omega_v = \alpha_v \oplus \beta_v \succeq \text{tr}_\text{s}(\alpha_f) \oplus \text{tr}_\text{s}(\beta_f)$. Therefore, using (L1), we know that $\omega_f = \alpha_f \oplus \beta_f$ satisfies $\omega_v \succeq \text{tr}_\text{s}(\alpha_f) \oplus \text{tr}_\text{s}(\beta_f) = \text{tr}_\text{s}(\omega_f)$ and $\omega_f \vDash_F A_1 * A_2$ (from $\alpha_f \vDash_F A_1$ and $\beta_f \vDash_F A_2$).

Inductive Case $\wedge$: $A = A_1 \wedge A_2$, with the induction hypothesis being that A_1 and A_2 are both backward satisfiable. In this case we will also use the fact that A_1 and A_2 have monotone interpretations.

Fix some front-end state ω_b and Viper state ω_v with $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and that $\omega_v \vDash_V \text{tr}_\text{a}(A_1 \wedge A_2) = \text{tr}_\text{a}(A_1) \wedge \text{tr}_\text{a}(A_2)$. Then ω_v satisfies both $\text{tr}_\text{a}(A_1)$ and $\text{tr}_\text{a}(A_2)$. For either $i = 1, 2$, we can get a front-end state ω_i with $\omega_i \vDash_F A_i$ and $\omega_v \succeq \text{tr}_\text{s}(\omega_i)$ using the induction hypothesis that A_i is backward satisfiable. Then by (L5), we know there exists some ω_f that $\omega_v \succeq \text{tr}_\text{s}(\omega_f)$ and $\omega_f \succeq \omega_i$ for both $i = 1, 2$. Finally, using monotonicity of the interpretation of A_i, we know that $\omega_f \vDash_F A_i$ for both $i = 1, 2$. Therefore, $\omega_f \vDash_F A_1 \wedge A_2$.

Inductive Case $\vee$: $A = A_1 \vee A_2$, with the induction hypothesis being that A_1 and A_2 are both backward satisfiable.

Fix some front-end state ω_b and Viper state ω_v with $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and that $\omega_v \vDash_V \text{tr}_\text{a}(A_1 \vee A_2) = \text{tr}_\text{a}(A_1) \vee \text{tr}_\text{a}(A_2)$. Then ω_v satisfies either $\text{tr}_\text{a}(A_1)$ or $\text{tr}_\text{a}(A_2)$. Without loss of generality, assume $\omega_v \vDash_V \text{tr}_\text{a}(A_1)$, then by the induction hypothesis that A_1 is backward satisfiable, there exists some ω_f such that $\omega_f \vDash_F A_1$ and $\omega_v \succeq \text{tr}_\text{s}(\omega_f)$. Therefore, $\omega_f \vDash_F A_1 \vee A_2$.

Inductive Case $\longrightarrow$: $A = e \longrightarrow A_0$, with the induction hypothesis being that A_0 is backward satisfiable. In this case we will also use Fact 5.20 and the fact that A_0 has a monotone interpretation.

Fix some front-end state ω_b and Viper state ω_v with $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and that $\omega_v \vDash_V \text{tr}_\text{a}(e \longrightarrow A_0) = \text{tr}_\text{e}(e) \longrightarrow \text{tr}_\text{a}(A_0)$. Then we know that $I_e^V(\text{tr}_\text{e}(e), \omega_v)$ is defined, and if $I_e^V(\text{tr}_\text{e}(e), \omega_v) = \text{Bool True}$ then $\omega_v \vDash_V \text{tr}_\text{a}(A_0)$. This case is a bit tricky, in that if $I_e^V(\text{tr}_\text{e}(e), \omega_v) = \text{Bool True}$, although we can find ω_f such that $\omega_f \vDash_F A_0$, we cannot promise that $I_e^F(e, \omega_f)$ is defined. Such ω_f is not enough to satisfy $e \longrightarrow A_0$. To ensure that $I_e^F(e, \omega_f)$ is defined while preserving the

relation $\omega_v \succeq \mathrm{tr_s}(\omega_f)$, we add the backward translation of the core of ω_v to the ω_f that the induction hypothesis about A_0 gives us.

Concretely, we consider the following two cases.

- $I_e^V(\mathrm{tr_e}(e), \omega_v) =$ Bool False: Then this case becomes the same as the "**Base Case** E". Concretely, by the axiom (C1) of PCM with cores, $\omega_v \succeq \mathrm{tr_s}(\omega_f)$. Then by (A16), $I_e^V(\mathrm{tr_e}(e), |\omega_v|) = I_e^V(\mathrm{tr_e}(e), \omega_v) =$ Bool False. Moreover, (L6) guarantees the existence of front-end state ω_f such that $\mathrm{tr_s}(\omega_f) = |\omega_v|$. Then by (A11), $I_e^F(e, \omega_f) =$ False. As a result, $\omega_f \vDash_F e \longrightarrow A_0$.
- $I_e^V(\mathrm{tr_e}(e), \omega_v) =$ Bool True: Now we know that $\omega_v \vDash_V \mathrm{tr_a}(A_0)$. As a result, using backward satisfiability of A_0 provided by the induction hypothesis, we get front-end state ω_f' such that $\omega_v \succeq \mathrm{tr_s}(\omega_f')$ and $\omega_f' \vDash_F A_0$. Moreover, we obtain ω_c such that $\mathrm{tr_s}(\omega_c) = |\omega_v|$ from (L6). Note that $\omega_v \# |\omega_v|$. By Fact 5.20 (and the reflexivity of $\succeq$), we then know $\mathrm{tr_s}(\omega_f') \# \mathrm{tr_s}(\omega_c)$ and $\omega_v = \omega_v \oplus |\omega_v| \succeq \mathrm{tr_s}(\omega_f') \oplus \mathrm{tr_s}(\omega_c)$. Therefore, we find the front-end state ω_f as $\omega_f' \oplus \omega_c$. It satisfies that $\omega_v \succeq \mathrm{tr_s}(\omega_f') \oplus \mathrm{tr_s}(\omega_c) = \mathrm{tr_s}(\omega_f)$. Moreover, by (A16), $I_e^V(\mathrm{tr_e}(e), |\omega_v|) = I_e^V(\mathrm{tr_e}(e), \omega_v)$ is defined. By (A15) and that $\mathrm{tr_s}(\omega_f) = \mathrm{tr_s}(\omega_f') \oplus |\omega_v| \succeq |\omega_v|$, we know that $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega_f)) = I_e^V(\mathrm{tr_e}(e), |\omega_v|)$ is also defined. Finally, by (A11), $I_e^F(e, \omega_f)$ is defined. Combining with the fact that $\omega_f \vDash_F A_0$, we eventually conclude that $\omega_f \vDash_F e \longrightarrow A_0$.

Inductive Case $\exists$: $A = \exists x : t.\, A_0$, with the induction hypothesis being that A_0 is backward satisfiable.

Fix some front-end state ω_b and Viper state ω_v with $\mathrm{tr_s}(\omega_b) \succeq \omega_v$ and that $\omega_v \vDash_V \mathrm{tr_a}(\exists x : t.\, A_0)$. Then by Lemma 5.4, there exists some front-end value v of type t, such that $\omega_v(x \mapsto \mathrm{tr_v}(v)) \vDash_V \mathrm{tr_a}(A_0)$. Note that $\mathrm{tr_s}(\omega_b) \succeq \omega_v$ implies that

$$\mathrm{tr_s}(\omega_b(x \mapsto v)) = (\mathrm{tr_s}(\omega_b))(x \mapsto \mathrm{tr_v}(v)) \succeq \omega_v(x \mapsto \mathrm{tr_v}(v)). \tag{5.3}$$

Then, using the induction hypothesis that A_0 is backward satisfiable, we get ω_f' such that $\omega_f' \vDash_F A_0$ and $\omega_v(x \mapsto \mathrm{tr_v}(v)) \succeq \mathrm{tr_s}(\omega_f')$. This is almost what we want, except that we want the order relation between the front-end state and ω_v rather than $\omega_v(x \mapsto \mathrm{tr_v}(v))$. As a result, we want to change the value of variable x in ω_f'. Luckily, ω_b gives us a good choice of the value of x, as $\mathrm{tr_s}(\omega_b) \succeq \omega_v$ means that the translation of its store is exactly the store of ω_v. Therefore, we pick the store of ω_b and the heap of ω_f' to construct our front-end state ω_f (assume $\omega_b = (\sigma_b, h_b)$ and $\omega_f' = (\sigma_f', h_f')$, then $\omega_f = (\sigma_b, h_f')$). Now we prove that $\omega_v \succeq \mathrm{tr_s}(\omega_f)$ and $\omega_f \vDash_F \exists x : t.\, A_0$:

- Since the heap of ω_f is the same as ω'_f, and the heap of ω_v is the same as $\omega_v(x \mapsto \mathrm{tr_v}(v))$, using the relation $\omega_v(x \mapsto \mathrm{tr_v}(v)) \succeq \mathrm{tr_s}(\omega'_f)$, we know that the heap of ω_v is not smaller than the translation of the heap of ω_f. For the store, note that $\mathrm{tr_s}(\omega_b) \succeq \omega_v$ implies that $\mathrm{tr_s}(\omega_b)$ and ω_v have an equal store. Moreover, ω_f uses the store of ω_b, which implies that its translation has the same store as $\mathrm{tr_s}(\omega_b)$. Therefore, $\mathrm{tr_s}(\omega_f)$ has the same store as ω_v. With the relation of the store part and the heap part of $\mathrm{tr_s}(\omega_f)$ and ω_v, we then conclude that $\omega_v \succeq \mathrm{tr_s}(\omega_f)$.
- Note that $\omega_v(x \mapsto \mathrm{tr_v}(v)) \succeq \mathrm{tr_s}(\omega'_f)$ implies that their stores are equal. Moreover, Equation (5.3) implies that $\mathrm{tr_s}(\omega_b(x \mapsto v))$ also has the same store as $\omega_v(x \mapsto \mathrm{tr_v}(v))$. As a result, the translated states (whose stores are the translation of the stores of the front-end states) of ω'_f and $\omega_b(x \mapsto v)$ have equal stores. By (R2), $\omega_b(x \mapsto v)$ and ω'_f have the same store. Since ω_f inherits ω_b's store, the store of ω'_f is also that of $\omega_f(x \mapsto v)$. Finally, because ω_f inherits the heap of ω'_f, we get that $\omega'_f = \omega_f(x \mapsto v)$. Since $\omega'_f \vDash_F A_0$ and v is of type t, we eventually conclude that $\omega_f \vDash_F \exists x : t.\, A_0$. □

In addition to the backward satisfiability, we also need that the translation of all front-end syntactic assertions to preserve the semantics of their interpretation in the front-end, as defined in Definition 5.21.

Definition 5.21 (semantics-preserving of the translation $\mathrm{tr_a}$) *The translation of a front-end syntactic assertion A preserves its semantics if and only if: for any front-end state ω, $\omega \vDash_F A$ if and only if $\mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A)$.*

The semantics-preserving property isn't as trivial as it seems to be. For example, $A * B$ is translated to $\mathrm{tr_a}(A) * \mathrm{tr_a}(B)$, which is satisfied as long as the translated state can be split into two Viper states that satisfy $\mathrm{tr_a}(A)$ and $\mathrm{tr_a}(B)$ respectively. However, it is not guaranteed that the split exists in the front-end as well, as in the following example.

Example 5.22 (semantics of separating conjunctions changes) *Let the front-end use a rational permission model while assume that Viper uses a real permission model. Let assertions*

$$P_1 = \left(\mathrm{acc}(v.p, 1) * v.p \geq 0 * v.p^2 < \frac{1}{2}\right) \mathrel{-\!\!*} \mathrm{acc}(x.f, v.p),$$

$$P_2 = \left(\mathrm{acc}(v.p, 1) * 0 \leq v.p \leq 1 * (1 - v.p)^2 > \frac{1}{2}\right) \mathrel{-\!\!*} \mathrm{acc}(x.f, v.p)$$

as the two assertions in Example 3.4 in Section 3.2.1. Assume that they are translated to exactly the same form in Viper. As a result, in both front-end and Viper, P_1 and

P_2 have the interpretation the same as $\text{acc}\left(x.f, \sqrt{\frac{1}{2}}\right)$ *and* $\text{acc}\left(x.f, 1-\sqrt{\frac{1}{2}}\right)$*, respectively. Consider the front-end state ω that has 1 permission to the heap location $x.f$. Then it is translated to a Viper state that also has 1 permission to $x.f$. In Viper, $\text{tr}_\text{s}(\omega)$ can easily satisfy $\text{tr}_\text{a}(P_1) * \text{tr}_\text{a}(P_2)$ by splitting its permission to $x.f$. However, this is not possible for ω in the front-end, since $\sqrt{\frac{1}{2}}$ is irrational, and thus is not an expressible permission value. Therefore, ω does not satisfy $P_1 * P_2$.*

In order for tr_a to preserve semantics, we need backward satisfiability of the front-end syntactic assertions, which has been proved in Lemma 5.19, and monotonicity of the translated assertions, which is given by the following lemma:

Lemma 5.23 (monotonicity of translated assertions) *For any front-end syntactic assertion A, $\text{tr}_\text{a}(A)$ is monotone.*

Proof As in Lemma 5.17, we use structural induction on A. The proofs for the inductive cases are the same as in Lemma 5.17, while the two base cases are directly proved from the conditions (A14) and (A15). □

We then prove the semantics-preserving lemma.

Lemma 5.24 (semantics-preserving of tr_a) *For any front-end syntactic assertion A, $\text{tr}_\text{a}(A)$ preserves its semantics. That is, for any front-end state ω, $\omega \vDash_F A \iff \text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A)$.*

Proof Use structural induction on A.

Base Case A_p: A is a primitive assertion.

This case is proved by the condition (A12).

Base Case E: A is an expression e.

This case is proved by the condition (A11).

Inductive Case $*$: $A = A_1 * A_2$, with the induction hypothesis being that $\text{tr}_\text{a}(A_1)$ and $\text{tr}_\text{a}(A_2)$ preserve semantics. In this case we will also use the facts that $\text{tr}_\text{a}(A_1)$ is monotone and that A_2 is backward satisfiable.

Fix some front-end state ω. In order to prove that $\omega \vDash_F A_1 * A_2 \iff \text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A_1 * A_2) = \text{tr}_\text{a}(A_1) * \text{tr}_\text{a}(A_2)$, we need to show that $\omega \vDash_F A_1 * A_2 \implies \text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A_1) * \text{tr}_\text{a}(A_2)$, and that $\text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A_1) * \text{tr}_\text{a}(A_2) \implies \omega \vDash_F A_1 * A_2$.

- Assume $\omega \vDash_F A_1 * A_2$. Then there exist α and β such that $\omega = \alpha \oplus \beta$, $\alpha \vDash_F A_1$, and $\beta \vDash_F A_2$. Then $\text{tr}_\text{s}(\alpha) \vDash_V \text{tr}_\text{a}(A_1)$ and $\text{tr}_\text{s}(\beta) \vDash_V \text{tr}_\text{a}(A_2)$ by the induction hypothesis. Moreover, by (L1), $\text{tr}_\text{s}(\omega) = \text{tr}_\text{s}(\alpha) \oplus \text{tr}_\text{s}(\beta)$. As a result, $\text{tr}_\text{s}(\omega) \vDash_V \text{tr}_\text{a}(A_1) * \text{tr}_\text{a}(A_2)$.

- Assume $\mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) * \mathrm{tr_a}(A_2)$. Then there exist Viper states α_v and β_v such that $\mathrm{tr_s}(\omega) = \alpha_v \oplus \beta_v$, $\alpha_v \vDash_V \mathrm{tr_a}(A_1)$, and $\beta_v \vDash_V \mathrm{tr_a}(A_2)$. To get a split in the front-end, we need to modify α_v and β_v such that they both have preimages in the front-end. The way we achieve this is by increasing α_v using the monotonicity of $\mathrm{tr_a}(A_1)$ and decreasing β_v using the backward satisfiability of A_2. Concretely, since $\mathrm{tr_s}(\omega) \succeq \beta_v$, there exists some front-end state β_f such that $\beta_v \succeq \mathrm{tr_s}(\beta_f)$ and $\beta_f \vDash_F A_2$. Let δ be the difference between β_v and $\mathrm{tr_s}(\beta_f)$ (i.e., $\beta_v = \delta \oplus \mathrm{tr_s}(\beta_f)$). Then by associativity of $\oplus$, $\mathrm{tr_s}(\omega) = \alpha_v \oplus (\delta \oplus \mathrm{tr_s}(\beta_f)) = (\alpha_v \oplus \delta) \oplus \mathrm{tr_s}(\beta_f)$. Moreover, by (L4), there exists some front-end state α_f such that $\mathrm{tr_s}(\alpha_f) = \alpha_v \oplus \delta \succeq \alpha_v$. As a result of monotonicity of $\mathrm{tr_a}(A_1)$, $\mathrm{tr_s}(\alpha_f) \vDash_V \mathrm{tr_a}(A_1)$, which further implies that $\alpha_f \vDash_F A_1$ by the induction hypothesis that $\mathrm{tr_a}(A_1)$ preserves semantics. Since $\mathrm{tr_s}(\omega) = \mathrm{tr_s}(\alpha_f) \oplus \mathrm{tr_s}(\beta_f)$ implies $\omega = \alpha_f \oplus \beta_f$ by (L1), we eventually conclude that $\omega \vDash_F A_1 * A_2$.

  Remark 5.25 *This technique of increasing one part and decreasing the other part when dealing with separating conjunctions will be used again in step 5 of the proof of the frame backward translation lemma (Lemma 5.26), and in the "**Case Heap Assignment**" of the proof of the condition (A19) for the instantiation of the framework in Section 6.2.*

Inductive Case $\wedge$: $A = A_1 \wedge A_2$, with the induction hypothesis being that $\mathrm{tr_a}(A_1)$ and $\mathrm{tr_a}(A_2)$ preserve semantics. Then for any front-end state ω,

$$\begin{aligned} & \omega \vDash_F A_1 \wedge A_2 \\ \iff & \omega \vDash_F A_1 \text{ and } \omega \vDash_F A_2 \\ \iff & \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) \text{ and } \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) \\ \iff & \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) \wedge \mathrm{tr_a}(A_2) = \mathrm{tr_a}(A_1 \wedge A_2). \end{aligned}$$

Inductive Case $\vee$: $A = A_1 \vee A_2$, with the induction hypothesis being that $\mathrm{tr_a}(A_1)$ and $\mathrm{tr_a}(A_2)$ preserve semantics. Then for any front-end state ω,

$$\begin{aligned} & \omega \vDash_F A_1 \vee A_2 \\ \iff & \omega \vDash_F A_1 \text{ or } \omega \vDash_F A_2 \\ \iff & \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) \text{ or } \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_2) \\ \iff & \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_1) \vee \mathrm{tr_a}(A_2) = \mathrm{tr_a}(A_1 \vee A_2). \end{aligned}$$

Inductive Case $\longrightarrow$: $A = e \longrightarrow A_0$, with the induction hypothesis being that $\mathrm{tr_a}(A_0)$ preserves semantics.

Fix any front-end state ω. We prove that $\omega \vDash_F e \longrightarrow A_0 \iff \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(e \longrightarrow A_0) = \mathrm{tr_e}(e) \longrightarrow \mathrm{tr_a}(A_0)$.

- If $\omega \vDash_F e \longrightarrow A_0$, then $I_e^F(e, \omega)$ is defined and $I_e^F(e, \omega) = \text{True} \implies \omega \vDash_F A_0$. By the condition (A11), $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega))$ is also defined, and $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega)) = \text{Bool True}$ if and only if $I_e^F(e, \omega) = \text{True}$. Moreover, $\omega \vDash_F A_0 \iff \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_0)$. Therefore, we have $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega)) = \text{Bool True} \implies \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_0)$, from which we then conclude that $\mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(e \longrightarrow A_0)$.
- If $\mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_e}(e) \longrightarrow \mathrm{tr_a}(A_0)$, then $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega))$ is defined, and $I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega)) = \text{Bool True} \implies \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_0)$. Again by the condition (A11), $I_e^F(e, \omega)$ is defined and $I_e^F(e, \omega) = \text{True} \iff I_e^V(\mathrm{tr_e}(e), \mathrm{tr_s}(\omega)) = \text{Bool True}$. Moreover, $\omega \vDash_F A_0 \iff \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(A_0)$. Therefore, we get $I_e^F(e, \omega) = \text{True} \implies \omega \vDash_F A_0$, from which we conclude that $\omega \vDash_F e \longrightarrow A_0$.

Inductive Case $\exists$: $A = \exists x : t.\, A_0$, with the induction hypothesis being that $\mathrm{tr_a}(A_0)$ preserves semantics. Then for any front-end state ω, by Lemma 5.4,

$$
\begin{aligned}
& \omega \vDash_F \exists x : t.\, A_0 \\
\iff\ & \exists v : t.\, \omega(x \mapsto v) \vDash_F A_0 \\
\iff\ & \exists v : t.\, \mathrm{tr_s}(\omega(x \mapsto v)) = (\mathrm{tr_s}(\omega))(x \mapsto \mathrm{tr_v}(v)) \vDash_V \mathrm{tr_a}(A_0) \\
\iff\ & \mathrm{tr_s}(\omega) \vDash_V \mathrm{tr_a}(\exists x : t.\, A_0).
\end{aligned}
$$

□

5.5.3 The Frame Backward Translation Lemma and the Proof

Now we state the frame backward translation lemma that resolves the challenge in Section 5.5.1.

Lemma 5.26 (frame backward translation) *Let T be a front-end type context, c be a front-end statement, P_0 and Q_0 be two front-end syntactic assertions, and P and Q be two Viper semantic assertions. Assume the following:*

(AS1) $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathbf{exhale}\ \mathrm{tr_a}(P_0);\ \mathbf{havoc}\ \mathrm{wr}(c);\ \mathbf{inhale}\ \mathrm{tr_a}(Q_0)\ \{Q\}$.

(AS2) $T \vdash_{\mathrm{CSL}} \{P_f\}\ c\ \{Q_f\}$, *where P_f and Q_f are the semantic interpretations of P_0 and Q_0 respectively.*

(AS3) *P and Q are monotone.*

Then $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$.

In the proof, we will need to make an assertion monotone by taking the closure of the set of satisfied states w.r.t. $\succeq$.

Definition 5.27 (monotonization) *The monotonization of a semantic assertion A, denoted as $M(A)$, is defined as: for any state ω, ω satisfies $M(A)$ if and only if there exists some state φ with $\omega \succeq \varphi$, such that φ satisfies A.*

It is straightforward to show that the operator M indeed makes the assertion monotone.

Proposition 5.28 *For any assertion A, $M(A)$ is monotone.*

We finally prove the frame backward translation lemma (Lemma 5.26).

Proof We prove the lemma in the following steps:

1. Derive intermediate Viper semantic assertions.

   Using the SL rule for sequential composition, from (AS1) we obtain Viper semantic assertions R_v and S_v such that

   $$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathbf{exhale}\ \mathrm{tr}_{\mathrm{a}}(P_0)\ \{R_v\}\,, \tag{5.4a}$$
   $$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{R_v\}\ \mathbf{havoc}\ \mathrm{wr}(c)\ \{S_v\}\,, \tag{5.4b}$$
   $$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{S_v\}\ \mathbf{inhale}\ \mathrm{tr}_{\mathrm{a}}(Q_0)\ \{Q\}\,. \tag{5.4c}$$

   Moreover, using the inversion rule (Lemma 2.24) for **exhale** and **inhale** and Proposition 2.28 in Section 2.4 about the inversion rule for a sequence of **havoc** statements, we know that $P \implies R_v * \mathrm{tr}_{\mathrm{a}}(P_0)$, $S_v = \exists \mathrm{wr}(c) : \mathrm{tr}_{\mathrm{tc}}(T)(\mathrm{wr}(c)).\, R_v$, and $Q = S_v * \mathrm{tr}_{\mathrm{a}}(Q_0)$.

2. Monotonize R_v and S_v.

   Let $M_R = M(R_v)$ and $M_S = \exists \mathrm{wr}(c) : \mathrm{tr}_{\mathrm{tc}}(T)(\mathrm{wr}(c)).\, M_R$. It is straightforward to show that $M_S = M(S_v)$ (in other words, the monotonize operator M commutes with the existential quantifier $\exists$).

   We now show that the entailment $P \implies R_v * \mathrm{tr}_{\mathrm{a}}(P_0)$ and equality $Q = S_v * \mathrm{tr}_{\mathrm{a}}(Q_0)$ are both preserved. That is, $P \implies M_R * \mathrm{tr}_{\mathrm{a}}(P_0)$ and $Q = M_S * \mathrm{tr}_{\mathrm{a}}(Q_0)$.

   - Fix some state ω and assume ω satisfies P. Then ω satisfies $R_v * \mathrm{tr}_{\mathrm{a}}(P_0)$, which further implies that there exist α and β satisfying $\alpha \oplus \beta = \omega$, α satisfies R_v and β satisfies $\mathrm{tr}_{\mathrm{a}}(P_0)$. By reflexivity of $\succeq$, $\alpha \succeq \alpha$. So α satisfies M_R. Therefore, ω satisfies $M_R * \mathrm{tr}_{\mathrm{a}}(P_0)$. We conclude that $P \implies M_R * \mathrm{tr}_{\mathrm{a}}(P_0)$.
   - Using the same argument as above, we know $Q \implies M_S * \mathrm{tr}_{\mathrm{a}}(Q_0)$. Therefore, to prove $Q = M_S * \mathrm{tr}_{\mathrm{a}}(Q_0)$, we only need to show the other direction of implication (i.e., $M_S * \mathrm{tr}_{\mathrm{a}}(Q_0) \implies Q$), which is as follows. Fix some state ω and assume that it satisfies $M_S * \mathrm{tr}_{\mathrm{a}}(Q_0)$. Then there exist α' and β such that $\omega = \alpha' \oplus \beta$, α' satisfies M_S, and β satisfies $\mathrm{tr}_{\mathrm{a}}(Q_0)$. By the fact $M_S = M(S_v)$,

there exists α satisfying $\alpha' \succeq \alpha$ and α satisfies S_v. As a result of Fact 5.20, we know that $\alpha \# \beta$ and $\omega = \alpha' \oplus \beta \succeq \alpha \oplus \beta$. Moreover, $\alpha \oplus \beta$ satisfies $S_v * \mathrm{tr_a}(Q_0) = Q$. Since Q is monotone by (AS3), we know that ω also satisfies Q. As a result, we conclude that $M_S * \mathrm{tr_a}(Q_0) \implies Q$.

3. Backward translate M_R and M_S.

   Let $R_f = \mathrm{btr}(M_R)$ and $S_f = \exists \mathrm{wr}(c) : T(\mathrm{wr}(c)).\, R_f$. Note that with the type mismatch between the front-end and Viper described in Section 3.2.2, S_f is not exactly the backward translation of M_S. Instead, in the following, we will prove

   $$S_f \implies \mathrm{btr}(M_S). \tag{5.5}$$

   Fix any front-end state ω that satisfies S_f. This means that for all variables $x \in \mathrm{wr}(c)$, there exists some front-end value v_x of type $T(x)$, such that $\omega(x \mapsto v_x \mid x \in \mathrm{wr}(c))$ satisfies $R_f = \mathrm{btr}(M_R)$, where $\omega(x \mapsto v_x \mid x \in \mathrm{wr}(c))$ is obtained by changing the value of each variable $x \in \mathrm{wr}(c)$ to v_x in ω. As a result, $\mathrm{tr_s}(\omega(x \mapsto v_x \mid x \in \mathrm{wr}(c))) = (\mathrm{tr_s}(\omega))(x \mapsto \mathrm{tr_v}(v_x) \mid x \in \mathrm{wr}(c))$ satisfies M_R. Note that $\mathrm{tr_v}(v_x)$ has type $\mathrm{tr_{tc}}(T)(x)$ for each x. Therefore, $\mathrm{tr_s}(\omega)$ satisfies $M_S = \exists \mathrm{wr}(c) : \mathrm{tr_{tc}}(T)(\mathrm{wr}(c)).\, M_R$, which further implies that ω satisfies $\mathrm{btr}(M_S)$ by the definition of btr.

   Remark 5.29 *In the next step of the proof we will use S_f rather than $\mathrm{btr}(M_S)$ as the frame R to use the rule (Frame). The fact "$S_f \implies \mathrm{btr}(M_S)$" then resolves the effect in Example 3.5 in Section 3.2.2 that "*`havoc` *x" erases the knowledge of the front-end type of x in the state before the* `havoc`*: although the type information of x before "*`havoc` *x" is lost in Viper (which is reflected here as that the postcondition S_v (recall $S_v = \exists \mathrm{wr}(c) : \mathrm{tr_{tc}}(T)(\mathrm{wr}(c)).\, R_v$) of "*`havoc` $\mathrm{wr}(c)$*" in Equation* (5.4b) *only guarantees existence of value v_x of type $\mathrm{tr_{tc}}(T)(x)$ for each $x \in \mathrm{wr}(c)$), we can still obtain the front-end type of the existing x back from the backward translation of the precondition (here the precondition is $R_f = \mathrm{btr}(M(R_v))$). As a result, using the stronger $S_f = \exists \mathrm{wr}(c) : T(\mathrm{wr}(c)).\, R_f$ instead of $\mathrm{btr}(M(S_v))$ resolves this effect of type mismatch in Example 3.5.*

4. Use the rule (Frame) to add an assertion describing the frame to the pre- and post-conditions in (AS2).

   In order to satisfy the premises of the rule (Frame), we need to choose a self-framing assertion that is independent of all the written variables in c for well-typed states w.r.t. the type context T. We choose the assertion S_f that we derived in step 3. Note that S_f satisfies the independence requirement, because of its quantification over the written variables in c.

To use the rule (Frame), we still need to show that S_f is self-framing. To prove it, we start with R_v. Fact 2.27 guarantees that R_v, which is the postcondition of the triple in Equation (5.4a), is self-framing.

The monotonize operator M, the backward translate operator btr, and the existential quantifier $\exists$ all preserve self-framedness, as shown in the following lemmas:

Lemma 5.30 (*M* preserves self-framedness) *If A is self-framing, then so is $M(A)$.*

Proof Fix a state ω. We need to show that ω satisfies $M(A)$ if and only if stabilize(ω) satisfies $M(A)$.

If ω satisfies $M(A)$, then there exists some state φ such that $\omega \succeq \varphi$ and φ satisfies A. By the axiom (S2) of separation algebras for IDF, we know that stabilize$(\omega) \succeq$ stabilize(φ). Moreover, by the fact that A is self-framing, stabilize(φ) also satisfies A. Therefore, stabilize(ω) satisfies $M(A)$ by the definition of M.

If stabilize(ω) satisfies $M(A)$, then by monotonicity of $M(A)$ and the fact $\omega \succeq$ stabilize(ω) from the axiom (S1) of separation algebras for IDF, we immediately get that ω also satisfies $M(A)$. □

Lemma 5.31 (btr preserves self-framedness) *If A is self-framing, then so is* btr(A).

Proof Fix any front-end state ω. We need to show that ω satisfies btr(A) if and only if stabilize(ω) satisfies btr(A). In fact, we have,

$$
\begin{aligned}
& \omega \text{ satisfies } \mathrm{btr}(A) \\
\iff\ & \mathrm{tr_s}(\omega) \text{ satisfies } A \\
\iff\ & \text{stabilize}(\mathrm{tr_s}(\omega)) \text{ satisfies } A && \text{(by self-framedness of } A\text{)} \\
\iff\ & \mathrm{tr_s}(\text{stabilize}(\omega)) \text{ satisfies } A && \text{(by (L3))} \\
\iff\ & \text{stabilize}(\omega) \text{ satisfies } \mathrm{btr}(A).
\end{aligned}
$$

This concludes the proof. □

Lemma 5.32 ($\exists$ preserves self-framedness) *If A is self-framing, then so is $\exists x : t.\, A$.*

Proof Fix any front-end state ω. We need to show that ω satisfies $\exists x : t.\, A$ if and only if stabilize(ω) satisfies $\exists x : t.\, A$. Note that the stabilize operator of states works separately on the store and the heap, and its function on the store is the identity mapping. As a result, assume $\omega = (\sigma, h)$, then

$$\text{stabilize}(\omega(x \mapsto v))$$

$$
\begin{aligned}
&= \text{stabilize}(\sigma(x \mapsto v), h) \\
&= (\text{stabilize}(\sigma(x \mapsto v)), \text{stabilize}(h)) \\
&= (\sigma(x \mapsto v), \text{stabilize}(h)) \\
&= (\sigma, \text{stabilize}(h))(x \mapsto v) \\
&= (\text{stabilize}(\omega))(x \mapsto v).
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
& \omega \text{ satisfies } \exists x : t.\, A \\
\iff\ & \exists v : t.\, \omega(x \mapsto v) \text{ satisfies } A \\
\iff\ & \exists v : t.\, \text{stabilize}(\omega(x \mapsto v)) = (\text{stabilize}(\omega))(x \mapsto v) \text{ satisfies } A \\
\iff\ & \text{stabilize}(\omega) \text{ satisfies } \exists x : t.\, A.
\end{aligned}
$$

This concludes the proof. □

As a result, $R_f = \text{btr}(M_R) = \text{btr}(M(R_v))$ is self-framing. Then, using an induction on the finite list $\text{wr}(c)$ and applying Lemma 5.32, we get that $S_f = \exists \text{wr}(c) : T(\text{wr}(c)).\, R_f$ is self-framing as well, and we can finally apply the rule (Frame) to $T \vdash_{\text{CSL}} \{P_f\}\ c\ \{Q_f\}$ in (AS2) and get

$$T \vdash_{\text{CSL}} \{P_f * S_f\}\ c\ \{Q_f * S_f\}. \tag{5.6}$$

5. Use the rules (ConseqPre) and (ConseqPost) to conclude the proof.

   Recall that we want to prove $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$. Since P and Q are self-framing by Fact 2.27 and (AS1), and btr preserves self-framedness by Lemma 5.31, $\text{btr}(P)$ and $\text{btr}(Q)$ are self-framing. In order to get $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c\ \{\text{btr}(Q)\}$ from Equation (5.6) using the rules (ConseqPre) and (ConseqPost), we are left to show $\text{btr}(P) \Longrightarrow_T P_f * S_f$ and $Q_f * S_f \Longrightarrow_T \text{btr}(Q)$. In the following we prove them separately.

   - $\text{btr}(P) \Longrightarrow_T P_f * S_f$: We first use backward satisfiability of P_0 to show $\text{btr}(P) \implies R_f * P_f$:

     Proof Fix some front-end state ω_f that satisfies $\text{btr}(P)$. Then $\text{tr}_s(\omega_f)$ satisfies P. By $P \implies M_R * \text{tr}_a(P_0)$, $\text{tr}_s(\omega_f)$ also satisfies $M_R * \text{tr}_a(P_0)$. As a result, there exist Viper states φ_v and β_v such that $\text{tr}_s(\omega_f) = \varphi_v \oplus \beta_v$, φ_v satisfies M_R, and β_v satisfies $\text{tr}_a(P_0)$.

     Now we use the same technique of increasing one part of the state and decreasing the other part as in the proof of the "**Inductive Case** $*$" of Lemma 5.24. Concretely, by backward satisfiability of P_0 (and $\text{tr}_s(\omega_f) = \varphi_v \oplus \beta_v \succeq \beta_v$), we obtain front-end state β_f such that $\beta_v \succeq \text{tr}_s(\beta_f)$ and $\beta_f \vDash_F P_0$. This means that there

exists some Viper state δ as the difference of β_v and $\mathrm{tr_s}(\beta_f)$, i.e., $\beta_v = \delta \oplus \mathrm{tr_s}(\beta_f)$. Further using associativity of PCMs, we get that $\mathrm{tr_s}(\omega_f) = \varphi_v \oplus (\delta \oplus \mathrm{tr_s}(\beta_f)) = (\varphi_v \oplus \delta) \oplus \mathrm{tr_s}(\beta_f)$. (L4) then guarantees that $\varphi_v \oplus \delta$ is the image of the translation of some front-end state φ_f. As a result, by monotonicity of M_R, φ_v satisfies M_R, and $\mathrm{tr_s}(\varphi_f) = \varphi_v \oplus \delta \succeq \varphi_v$, we get $\mathrm{tr_s}(\varphi_f)$ satisfies M_R, which further implies that φ_f satisfies $\mathrm{btr}(M_R) = R_f$.

To summarize, we get $\omega_f = \varphi_f \oplus \beta_f$ (from $\mathrm{tr_s}(\omega_f) = \mathrm{tr_s}(\varphi_f) \oplus \mathrm{tr_s}(\beta_f)$ and (L1)), φ_f satisfies R_f, and $\beta_f \vDash_F P_0$ (i.e., β_f satisfies P_f). Therefore, ω_f satisfies $R_f * P_f$. This concludes the proof that $\mathrm{btr}(P) \implies R_f * P_f$. □

Then, by Proposition 2.13, we reduce the goal to $R_f * P_f \Longrightarrow_T P_f * S_f$. To be consistent with $R_f * P_f$, we rewrite $P_f * S_f$ to $S_f * P_f$. This rewriting is justified by commutativity of the separating conjunction, which is easy to prove. In order to further remove the separating conjunction and reduce the goal to $R_f \Longrightarrow_T S_f$, we prove the following lemma:

Lemma 5.33 ($\Longrightarrow_T$ of components of separating conjunctions) *If $A \Longrightarrow_T A'$, then $A * B \Longrightarrow_T A' * B$.*

Proof Fix some state ω that is well-typed w.r.t. T and satisfies $A * B$. Then there exist α and β such that $\omega = \alpha \oplus \beta$, α satisfies A, and β satisfies B. As a result, ω and α have the same stores. So α is also well-typed w.r.t. T. By $A \Longrightarrow_T A'$, we then know that α satisfies A' as well. As a result, ω also satisfies $A' * B$. □

The proof also works for the stronger entailment $\Longrightarrow$. That is, we have

Lemma 5.34 *If $A \implies A'$, then $A * B \implies A' * B$.*

We will use Lemma 5.34 later in the proof.

We now prove $R_f \Longrightarrow_T S_f$, where we recall $S_f = \exists \mathrm{wr}(c) : T(\mathrm{wr}(c)).\, R_f$. Fix some state ω that is well-typed w.r.t. T and satisfies R_f. Then for any variable $x \in \mathrm{wr}(c)$, the value of x in ω is of type $T(x)$ ((A20) guarantees that $T(x)$ is defined, which implies that variable x has to have some value in the well-typed state ω). The existence of such value proves that ω satisfies $\exists x : T(x).\, R_f$. We can then prove that ω satisfies S_f by an induction on the finite list $\mathrm{wr}(c)$.

- $Q_f * S_f \Longrightarrow_T \mathrm{btr}(Q)$: We prove the stronger $Q_f * S_f \implies \mathrm{btr}(Q)$.

  Using Equation (5.5) proved in step 3 and Lemma 5.34, we get $Q_f * S_f \implies Q_f * \mathrm{btr}(M_S)$. As a result, we reduce our goal to

proving

$$Q_f * \mathrm{btr}(M_S) \implies \mathrm{btr}(Q). \tag{5.7}$$

Recall $Q = M_S * \mathrm{tr_a}(Q_0) = \mathrm{tr_a}(Q_0) * M_S$ proved in step 2. Note that $\mathrm{tr_a}$ preserving the semantics of Q_0 implies that $\mathrm{btr}(\mathrm{tr_a}(Q_0))$ is exactly the interpretation of Q_0 in the front-end, i.e., Q_f. The three terms in Equation (5.7) are then the backward translation of the three terms in $\mathrm{tr_a}(Q_0) * M_S = Q$, respectively. Therefore, we want to see how the separating conjunction would change in the backward translation of assertions. Concretely, we have the following lemma.

Lemma 5.35 *For Viper semantic assertions A, B, and C, if $A = B * C$, then* $\mathrm{btr}(B) * \mathrm{btr}(C) \implies \mathrm{btr}(A)$.

Proof Fix any front-end state ω that satisfies $\mathrm{btr}(B) * \mathrm{btr}(C)$, then we obtain α and β such that $\omega = \alpha \oplus \beta$, α satisfies $\mathrm{btr}(B)$, and β satisfies $\mathrm{btr}(C)$. By the definition of btr, we get that $\mathrm{tr_s}(\alpha)$ satisfies B and $\mathrm{tr_s}(\beta)$ satisfies C. Moreover, by (L1), $\mathrm{tr_s}(\omega) = \mathrm{tr_s}(\alpha) \oplus \mathrm{tr_s}(\beta)$. So ω satisfies $B * C = A$. Again by the definition of btr, we conclude that ω satisfies $\mathrm{btr}(A)$. □

Remark 5.36 *The other direction of entailment* $\mathrm{btr}(A) \implies \mathrm{btr}(B) * \mathrm{btr}(C)$ *is not guaranteed to hold for general A, B, and C. In order to have this directly of entailment, i.e., to preserve the equality, we would need monotonicity held by one of B and C, and a similar property as backward satisfiability held by the other. The proof is then similar to the "**Inductive Case** $*$" in the proof of Lemma 5.24.*

As a result of Lemma 5.35 and $Q = \mathrm{tr_a}(Q_0) * M_S$ proved in step 2, we immediately show Equation (5.7) by $Q_f * \mathrm{btr}(M_S) = \mathrm{btr}(\mathrm{tr_a}(Q_0)) * \mathrm{btr}(M_S) \implies \mathrm{btr}(Q)$, which concludes the whole proof. □

5.5.4 Proving the Soundness Theorem

Recall Theorem 5.15.

Theorem *Let c be a front-end statement, T be a front-end type context, and P_v and Q_v be two Viper semantic assertions. If*

(P1) *P_v and Q_v are monotone.*

(P2) *c is well-formed w.r.t. T.*

(P3) *c with P_v and Q_v is Viper annotated SL provable w.r.t. T.*

Then $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P_v)\}\ c\ \{\mathrm{btr}(Q_v)\}$.

We are now ready to prove it.

Proof We prove the theorem by structural induction on c.

Base Case S_p: c is a primitive statement.

This is directly proved by the condition (A19).

Inductive Case Sequential: $c = c_1; c_2$. The induction hypothesis is:

(IH) For $i = 1, 2$, for arbitrary monotone Viper semantic assertions P and Q, if c_i is well-formed w.r.t. T, and c_i with P and Q is Viper annotated SL provable w.r.t. T, then $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c_i\ \{\text{btr}(Q)\}$.

From the premise (P3), we get an SL proof for $\text{tr}_{\text{c}}^{M}(c) = \text{tr}_{\text{c}}^{M}(c_1);\ \text{tr}_{\text{c}}^{M}(c_2)$ as

$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{P_v\}\ \text{tr}_{\text{c}}^{M}(c_1);\ \text{tr}_{\text{c}}^{M}(c_2)\ \{Q_v\}\,, \tag{5.8}$$

and proofs for each statement $c_v \in \text{tr}_{\text{c}}^{E}(c) = \text{tr}_{\text{c}}^{E}(c_1) \cup \text{tr}_{\text{c}}^{E}(c_2)$ with True as precondition and some semantic postcondition w.r.t. $\text{tr}_{\text{tc}}(T)$. In order to use (IH) of c_i for $i = 1, 2$, which requires Viper annotated SL provability of c_i, we now lack the SL proof for $\text{tr}_{\text{c}}^{M}(c_i)$. We will then construct it. Using the inversion rule (Lemma 2.24) for sequential composition, from Equation (5.8) we obtain an intermediate Viper semantic assertion R such that

$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{P_v\}\ \text{tr}_{\text{c}}^{M}(c_1)\ \{R\}\,, \tag{5.9a}$$

$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{R\}\ \text{tr}_{\text{c}}^{M}(c_2)\ \{Q_v\}\,. \tag{5.9b}$$

In order to apply (IH), we want that R is monotone. Although this is not promised to be true for the R that is originally generated by the Viper's proof, the following fact allows us to monotonize it.

Fact 5.37 (monotonizing pre- and post-conditions) *For any Viper type context T, Viper semantic assertions P and Q, and Viper statement c, if $T \vdash_{\text{VPR}} \{P\}\ c\ \{Q\}$, then $T \vdash_{\text{VPR}} \{M(P)\}\ c\ \{M(Q)\}$.*

Applying to Equations (5.9a) and (5.9b), and note that monotonizing an already monotone assertion does not change it, we get

$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{P_v\}\ \text{tr}_{\text{c}}^{M}(c_1)\ \{M(R)\}\,,$$
$$\text{tr}_{\text{tc}}(T) \vdash_{\text{VPR}} \{M(R)\}\ \text{tr}_{\text{c}}^{M}(c_2)\ \{Q_v\}\,.$$

We can now conclude that c_1 with P_v and $M(R)$, and c_2 with $M(R)$ and Q_v are both Viper annotated SL provable w.r.t. $\text{tr}_{\text{tc}}(T)$. Since P_v, $M(R)$, and Q_v are all monotone, and well-formedness of c w.r.t. T implies that both c_1 and c_2 are well-formed w.r.t. T, we can now apply (IH) on c_1 and c_2 to get

$$T \vdash_{\text{CSL}} \{\text{btr}(P_v)\}\ c_1\ \{\text{btr}(M(R))\}\,,$$

$$T \vdash_{\text{CSL}} \{\text{btr}(M(R))\}\ c_2\ \{\text{btr}(Q_v)\}\,.$$

Applying the rule (Seq) then gives

$$T \vdash_{\text{CSL}} \{\text{btr}(P_v)\}\ c_1;\ c_2\ \{\text{btr}(Q_v)\}\,.$$

Inductive Case If: $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$. The induction hypothesis in this case is:

(IH) For $i = 1, 2$, for arbitrary monotone Viper semantic assertions P and Q, if c_i is well-formed w.r.t. T, and c_i with P and Q is Viper annotated SL provable w.r.t. T, then $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c_i\ \{\text{btr}(Q)\}$.

From the premise (P3), we get an SL proof for $\text{tr}_\text{c}^M(c)$ as

$$\text{tr}_\text{tc}(T) \vdash_{\text{VPR}} \{P_v\}\ \texttt{if}\ \text{tr}_\text{e}(e)\ \texttt{then}\ \text{tr}_\text{c}^M(c_1)\ \texttt{else}\ \text{tr}_\text{c}^M(c_2)\ \{Q_v\}\,, \quad (5.10)$$

and proofs for each statement $c_v \in \text{tr}_\text{c}^E(c) = \text{tr}_\text{c}^E(c_1) \cup \text{tr}_\text{c}^E(c_2)$ with True as precondition and some semantic postcondition w.r.t. $\text{tr}_\text{tc}(T)$. As in last case, we want to use (IH) of c_i for $i = 1, 2$, which requires Viper annotated SL provability of c_i. With proof for each $c_v \in \text{tr}_\text{c}^E(c_i)$ with True as precondition and some semantic postcondition w.r.t. $\text{tr}_\text{tc}(T)$, we now lack the SL proof for $\text{tr}_\text{c}^M(c_i)$, which We will then construct. Using the inversion rule (Lemma 2.24) for conditional branchings, from Equation (5.10) we obtain B_1 and B_2 such that

(I1) $Q_v = B_1 \vee B_2$,

(I2) $\text{tr}_\text{e}(e)$ is well defined w.r.t. P_v, and

(I3) two proofs

$$\text{tr}_\text{tc}(T) \vdash_{\text{VPR}} \{P_v \wedge \text{tr}_\text{e}(e)\}\ \text{tr}_\text{c}^M(c_1)\ \{B_1\}\,, \quad (5.11a)$$
$$\text{tr}_\text{tc}(T) \vdash_{\text{VPR}} \{P_v \wedge \neg\text{tr}_\text{e}(e)\}\ \text{tr}_\text{c}^M(c_2)\ \{B_2\}\,. \quad (5.11b)$$

In order to apply (IH), we again need to monotonize the pre- and post-conditions by Fact 5.37.

Note that (A15) guarantees that both $\text{tr}_\text{e}(e)$ and $\neg\text{tr}_\text{e}(e)$ are monotone, and in the proof of Lemma 5.17, we actually show that normal conjunction preserves monotonicity. As a result, the preconditions in Equations (5.11a) and (5.11b) are both monotone. Then applying Fact 5.37 on them gives us

$$\text{tr}_\text{tc}(T) \vdash_{\text{VPR}} \{P_v \wedge \text{tr}_\text{e}(e)\}\ \text{tr}_\text{c}^M(c_1)\ \{M(B_1)\}\,,$$
$$\text{tr}_\text{tc}(T) \vdash_{\text{VPR}} \{P_v \wedge \neg\text{tr}_\text{e}(e)\}\ \text{tr}_\text{c}^M(c_2)\ \{M(B_2)\}\,.$$

We can then apply (IH) to get

$$T \vdash_{\text{CSL}} \{\text{btr}(P_v \wedge \text{tr}_e(e))\}\ c_1\ \{\text{btr}(M(B_1))\}\,, \tag{5.12a}$$
$$T \vdash_{\text{CSL}} \{\text{btr}(P_v \wedge \neg\text{tr}_e(e))\}\ c_2\ \{\text{btr}(M(B_2))\}\,. \tag{5.12b}$$

Now we need to connect these backward translated assertions with the front-end assertions for the rule (If). Let's start with the preconditions.

It is straightforward to prove

Proposition 5.38 $\text{btr}(A \wedge B) = \text{btr}(A) \wedge \text{btr}(B)$.

As a result, $\text{btr}(P_v \wedge \text{tr}_e(e)) = \text{btr}(P_v) \wedge \text{btr}(\text{tr}_e(e)) = \text{btr}(P_v) \wedge e$, and $\text{btr}(P_v \wedge \neg\text{tr}_e(e)) = \text{btr}(P_v) \wedge \text{btr}(\neg\text{tr}_e(e)) = \text{btr}(P_v) \wedge \neg e$, where the last equalities in these two equations come from (A11) that the translation of expressions preserves their semantics in the front-end.

For the postconditions, it is straightforward to prove the following two results.

Proposition 5.39 $M(A \vee B) = M(A) \vee M(B)$.

Proposition 5.40 $\text{btr}(A \vee B) = \text{btr}(A) \vee \text{btr}(B)$.

As a result,

$$\begin{aligned}
&\text{btr}(M(B_1)) \vee \text{btr}(M(B_2)) \\
&= \text{btr}(M(B_1) \vee M(B_2)) && \text{(by Proposition 5.40)} \\
&= \text{btr}(M(B_1 \vee B_2)) && \text{(by Proposition 5.39)} \\
&= \text{btr}(M(Q_v)) && (Q_v = B_1 \vee B_2) \\
&= \text{btr}(Q_v) && (M(A) = A \text{ for monotone } A)
\end{aligned}$$

To apply the rule (If), we are now left to prove its premises "e is well defined w.r.t. $\text{btr}(P_v)$". This is not hard to derive from (I2) and the condition (A11) that the translation of expressions preserves their semantics in the front-end. As a result, the rule (If) with Equations (5.12a) and (5.12b) gives

$$T \vdash_{\text{CSL}} \{\text{btr}(P_v)\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\text{btr}(Q_v)\}$$

as required.

Inductive Case While: $c = \texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c_f$. The induction hypothesis in this case is:

(IH) For arbitrary monotone Viper semantic assertions P and Q, if c_f is well-formed w.r.t. T, and c_f with P and Q is Viper annotated SL provable w.r.t. T, then $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c_f\ \{\text{btr}(Q)\}$.

From the premise (P3), we get

(J1) an SL proof for $\mathrm{tr}_{\mathrm{c}}^{M}(c)$ as

$$\begin{aligned}\mathrm{tr}_{\mathrm{tc}}(T) \vdash \{P_v\}\ &\mathbf{exhale}\ \mathrm{tr}_{\mathrm{a}}(I) \wedge (\mathrm{tr}_{\mathrm{e}}(e) \vee \neg \mathrm{tr}_{\mathrm{e}}(e));\\ &\mathbf{havoc}\ \mathrm{wr}(c);\\ &\mathbf{inhale}\ \mathrm{tr}_{\mathrm{a}}(I) \wedge \neg \mathrm{tr}_{\mathrm{e}}(e)\ \{Q_v\},\end{aligned} \tag{5.13}$$

(J2) proofs for each statement $c_v \in \mathrm{tr}_{\mathrm{c}}^{E}(c_f) \subseteq \mathrm{tr}_{\mathrm{c}}^{E}(c)$ with True as precondition and some semantic postcondition w.r.t. $\mathrm{tr}_{\mathrm{tc}}(T)$, and

(J3) an SL proof for $\mathrm{tr}_{\mathrm{c}}^{M}(c_f)$ and some Q' as

$$\begin{aligned}\mathrm{tr}_{\mathrm{tc}}(T) \vdash \{\mathrm{True}\}\ &\mathbf{inhale}\ \mathrm{tr}_{\mathrm{a}}(I) \wedge \mathrm{tr}_{\mathrm{e}}(e);\\ &\mathrm{tr}_{\mathrm{c}}^{M}(c_f);\\ &\mathbf{exhale}\ \mathrm{tr}_{\mathrm{a}}(I) \wedge (\mathrm{tr}_{\mathrm{e}}(e) \vee \neg \mathrm{tr}_{\mathrm{e}}(e))\ \{Q'\}.\end{aligned} \tag{5.14}$$

With the proofs for $c_v \in \mathrm{tr}_{\mathrm{c}}^{E}(c_f)$, in order to use (IH) of c_f which requires Viper annotated SL provability of c_f, we now lack the SL proof for $\mathrm{tr}_{\mathrm{c}}^{M}(c_f)$. Although Equation (5.14) looks similar to what we want, it is not easy to directly move the inhaled and exhaled assertions to the positions of pre- and postconditions. The reason is that when we try to apply the inversion rule (Lemma 2.24) on Equation (5.14) and get some assertions R and S such that $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{R\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c_f)\ \{S\}$ and $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{S\}\ \mathbf{exhale}\ \mathrm{tr}_{\mathrm{a}}(I) \wedge (\mathrm{tr}_{\mathrm{e}}(e) \vee \neg \mathrm{tr}_{\mathrm{e}}(e))\ \{Q'\}$, S may be stronger than what we want. Since there are no Viper SL rules that refine the pre- or postconditions as the rules in Figure 2.6 do, we are not able to adjust S to what we want. As a result, we instead refine the postcondition in the CSL proof. Concretely, we prove the following auxiliary lemma:

Lemma 5.41 *Let T be a front-end type context, c be a front-end statement that is well-formed w.r.t. T, and P and Q be two Viper semantic assertions. Assume the following:*

(LP1) *For any monotone Viper semantic assertions P' and Q', the Viper annotated SL provability of c with P' and Q' w.r.t. T implies*

$$T \vdash_{\mathrm{CSL}} \left\{\mathrm{btr}(P')\right\}\ c\ \left\{\mathrm{btr}(Q')\right\}.$$

(LP2) *P and Q are monotone.*

(LP3) $\mathrm{btr}(Q)$ *is self-framing.*

(LP4) $\forall c_v \in \mathrm{tr}_{\mathrm{c}}^{E}(c)$, *there exists Q_v such that* $\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ c_v\ \{Q_v\}$.

(LP5) *There exists some* Q_d *such that*

$$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ \texttt{inhale}\ P;\ \mathrm{tr}_{\mathrm{c}}^{M}(c);\ \texttt{exhale}\ Q\ \{Q_d\}\,.$$

Then $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$.

Proof From (LP5), using the inversion rule (Lemma 2.24) for sequential composition, we obtain R and S such that

$$\begin{aligned}\mathrm{tr}_{\mathrm{tc}}(T) &\vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ \texttt{inhale}\ P\ \{R\}\,,\\ \mathrm{tr}_{\mathrm{tc}}(T) &\vdash_{\mathrm{VPR}} \{R\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c)\ \{S\}\,,\\ \mathrm{tr}_{\mathrm{tc}}(T) &\vdash_{\mathrm{VPR}} \{S\}\ \texttt{exhale}\ Q\ \{Q_d\}\,.\end{aligned}$$

Monotonizing the pre- and postconditions, we get

$$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{\mathrm{True}\}\ \texttt{inhale}\ P\ \{M(R)\}\,, \tag{5.15a}$$

$$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{M(R)\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c)\ \{M(S)\}\,, \tag{5.15b}$$

$$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{M(S)\}\ \texttt{exhale}\ Q\ \{M(Q_d)\}\,. \tag{5.15c}$$

Using the inversion rule (Lemma 2.24) for **inhale**, we further get $M(R) = \mathrm{True} * P = M(P) = P$ (it is straightforward to prove that for any assertion A, $\mathrm{True} * A = M(A)$), where the last equality comes from monotonicity of P in (LP2). Therefore, Equation (5.15b) becomes

$$\mathrm{tr}_{\mathrm{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr}_{\mathrm{c}}^{M}(c)\ \{M(S)\}\,.$$

Combining with (LP4), we conclude that c is Viper annotated SL provable with monotone assertions P and $M(S)$ w.r.t. T. As a result of (LP1), we get

$$T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(M(S))\}\,. \tag{5.16}$$

Finally, we want to use the rule (ConseqPost) to refine the postcondition and conclude the proof. Since the self-framedness of $\mathrm{btr}(Q)$ is provided by (LP3), we are left to show $\mathrm{btr}(M(S)) \implies \mathrm{btr}(Q)$.

Using the inversion rule (Lemma 2.24) for **exhale**, from Equation (5.15c) we get $M(S) \implies Q * M(Q_d)$. Furthermore, as $M(Q_d) \implies \mathrm{True}$ and Q is monotone by (LP2), Lemma 5.34 gives $Q * M(Q_d) \implies Q * \mathrm{True} = M(Q) = Q$. Therefore, $M(S) \implies Q$. We eventually conclude $\mathrm{btr}(M(S)) \implies \mathrm{btr}(Q)$ by the following proposition which is easy to prove.

Proposition 5.42 *If* $A \implies B$, *then* $\mathrm{btr}(A) \implies \mathrm{btr}(B)$.

Applying the rule (ConseqPost) to Equation (5.16), we directly get $T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$ as required. □

We want to apply Lemma 5.41 with c being c_f, P being $\mathrm{tr_a}(I) \wedge \mathrm{tr_e}(e)$, and Q being $\mathrm{tr_a}(I) \wedge (\mathrm{tr_e}(e) \vee \neg \mathrm{tr_e}(e))$.

By Propositions 5.38 and 5.40, we know that $\mathrm{btr}(P) = \mathrm{btr}(\mathrm{tr_a}(I)) \wedge \mathrm{btr}(\mathrm{tr_e}(e))$ and $\mathrm{btr}(Q) = \mathrm{btr}(\mathrm{tr_a}(I)) \wedge (\mathrm{btr}(\mathrm{tr_e}(e)) \vee \mathrm{btr}(\neg \mathrm{tr_e}(e)))$. Furthermore, by Lemma 5.24 and the condition (A11), $\mathrm{btr}(\mathrm{tr_a}(I)) = I$, $\mathrm{btr}(\mathrm{tr_e}(e)) = e$, and $\mathrm{btr}(\neg \mathrm{tr_e}(e)) = \neg e$, where we use I and e also to denote the interpretation of them. As a result, $\mathrm{btr}(P) = I \wedge e$ and $\mathrm{btr}(Q) = I \wedge (e \vee \neg e)$.

Now we check the premises of Lemma 5.41:

- The well-formedness of c_f comes from the well-formedness of c. As a result, (LP1) is satisfied from (IH).
- Monotonicity of $\mathrm{tr_a}(I)$ is from Lemma 5.23, while that of $\mathrm{tr_e}(e)$ and $\neg \mathrm{tr_e}(e)$ comes from (A15). Moreover, as the proof of Lemma 5.17 shows, $\wedge$ and $\vee$ preserve monotonicity. Therefore, $\mathrm{tr_a}(I) \wedge \mathrm{tr_e}(e)$ and $\mathrm{tr_a}(I) \wedge (\mathrm{tr_e}(e) \vee \neg \mathrm{tr_e}(e))$ are monotone, satisfying (LP2).
- I is self-framing by well-formedness of c. Moreover, I frames e. It is then not hard to prove that $\mathrm{btr}(Q) = I \wedge (e \vee \neg e)$ is self-framing. As a result, (LP3) is satisfied.
- (LP4) can be derived from (J2).
- We use Equation (5.14) as (LP5).

As an application of Lemma 5.41, we then get

$$T \vdash_{\mathrm{CSL}} \{I \wedge e\}\ c_f\ \{I \wedge (e \vee \neg e)\}\,. \tag{5.17}$$

Now we want to apply the rule (While) with P being $I \wedge (e \vee \neg e)$. Note that then $(I \wedge (e \vee \neg e)) \wedge e = I \wedge e$. So Equation (5.17) can be used as one of the premises. For the others, e is naturally well defined w.r.t. $I \wedge (e \vee \neg e)$, and it is straightforward to prove that $I \wedge (e \vee \neg e)$ frames e from I frames e, which comes from well-formedness of c. As a result, the rule (While) gives us

$$T \vdash_{\mathrm{CSL}} \{I \wedge (e \vee \neg e)\}\ c\ \{I \wedge (e \vee \neg e) \wedge \neg e\}\,, \tag{5.18}$$

where the postcondition $(I \wedge (e \vee \neg e)) \wedge \neg e = I \wedge \neg e$.

Note that the exhaled and inhaled assertions in Equation (5.13) are exactly the translation of the pre- and post-conditions in Equation (5.18). Combining these two proofs with the premise that P_v and Q_v are monotone, we finally can apply the frame backward translation lemma (Lemma 5.26) to get

$$T \vdash_{\mathrm{CSL}} \{\mathrm{btr}(P_v)\}\ \texttt{while}\ e\ \texttt{inv}\ I\ \texttt{do}\ c_f\ \{\mathrm{btr}(Q_v)\}$$

as required.

Inductive Case Parallel: $c = \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\}$. The induction hypothesis in this case is:

(IH) For $i = 1, 2$, for arbitrary monotone Viper semantic assertions P and Q, if c_i is well-formed w.r.t. T, and c_i with P and Q is Viper annotated SL provable w.r.t. T, then $T \vdash_{\text{CSL}} \{\text{btr}(P)\}\ c_i\ \{\text{btr}(Q)\}$.

From the premise (P3), we get

(K1) an SL proof for $\text{tr}_{\text{c}}^{M}(c)$ as

$$\begin{aligned} \text{tr}_{\text{tc}}(T) \vdash \{P_v\}\ &\textbf{exhale}\ \text{tr}_{\text{a}}(P_1 * P_2); \\ &\textbf{havoc}\ \text{wr}(c); \\ &\textbf{inhale}\ \text{tr}_{\text{a}}(Q_1 * Q_2)\ \{Q_v\}, \end{aligned} \tag{5.19}$$

(K2) for $i = 1, 2$, proofs for each statement $c_v \in \text{tr}_{\text{c}}^{E}(c_i) \subseteq \text{tr}_{\text{c}}^{E}(c)$ with True as precondition and some semantic postcondition w.r.t. $\text{tr}_{\text{tc}}(T)$, and

(K3) for $i = 1, 2$, an SL proof for $\text{tr}_{\text{c}}^{M}(c_i)$ and some Q_i' as

$$\begin{aligned} \text{tr}_{\text{tc}}(T) \vdash \{\text{True}\}\ &\textbf{inhale}\ \text{tr}_{\text{a}}(P_i); \\ &\text{tr}_{\text{c}}^{M}(c_i); \\ &\textbf{exhale}\ \text{tr}_{\text{a}}(Q_i)\ \{Q_i'\}. \end{aligned} \tag{5.20}$$

For $i = 1, 2$, we want to apply Lemma 5.41 with c being c_i, P being $\text{tr}_{\text{a}}(P_i)$, and Q being $\text{tr}_{\text{a}}(Q_i)$ to derive $T \vdash_{\text{CSL}} \{P_i\}\ c_i\ \{Q_i\}$. The premises of the lemma are checked as follows:

- By well-formedness of c, we know c_i is well-formed, which together with (IH) provides satisfaction of (LP1).
- By Lemma 5.23, $\text{tr}_{\text{a}}(P_i)$ and $\text{tr}_{\text{a}}(Q_i)$ are monotone. Therefore, (LP2) is satisfied.
- We have $\text{btr}(\text{tr}_{\text{a}}(Q_i)) = Q_i$ by semantics-preserving property of Q_i (Lemma 5.24). Moreover, Q_i is self-framing by the well-formedness requirement of c (specified in Definition 5.8). Therefore, (LP3) is satisfied.
- (LP4) is provided by (K2).
- We use Equation (5.20) as (LP5).

Therefore, the application of Lemma 5.41 gives

$$T \vdash_{\text{CSL}} \{P_i\}\ c_i\ \{Q_i\}\,. \tag{5.21}$$

Moreover, by Lemma 5.7 and well-formedness requirement of c specified in Definition 5.8, the other premises in the rule (Par) are all satisfied. As a result, the application of the rule (Par) on Equations (5.21) for $i = 1, 2$ gives

$$T \vdash_{\text{CSL}} \{P_1 * P_2\} \ \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\} \ \{Q_1 * Q_2\}\,. \tag{5.22}$$

Finally, note that the exhaled and inhaled assertions in Equation (5.19) are exactly the translation of the pre- and postconditions in Equation (5.22). As a result, combining these two proofs with the premise that P_v and Q_v are monotone, we can apply the frame backward translation lemma (Lemma 5.26) to get

$$T \vdash_{\text{CSL}} \{\text{btr}(P_v)\} \ \{P_1\}\, c_1\, \{Q_1\} \parallel \{P_2\}\, c_2\, \{Q_2\} \ \{\text{btr}(Q_v)\}$$

as required.

This concludes the proof of Theorem 5.15. □

Finally, let's recall the desired soundness theorem (Theorem 5.11), which directly follows from Theorem 5.15:

Theorem *Let c be a front-end statement, T be a front-end type context, and P and Q be front-end syntactic assertions. Let P_f and Q_f be the front-end semantic assertions that are the interpretation of P and Q respectively. If c is well-formed w.r.t. T, and c with precondition P and postcondition Q is SL provable in Viper w.r.t. T, then $T \vdash_{\text{CSL}} \{P_f\}\ c\ \{Q_f\}$.*

Proof By Lemma 5.23, $\text{tr}_\text{a}(P)$ and $\text{tr}_\text{a}(Q)$ are monotone, and by Lemma 5.24, P_f and Q_f are exactly $\text{btr}(\text{tr}_\text{a}(P))$ and $\text{btr}(\text{tr}_\text{a}(Q))$. Therefore, applying Theorem 5.15 with $P_v = \text{tr}_\text{a}(P)$ and $Q_v = \text{tr}_\text{a}(Q)$ immediately concludes the proof. □

Chapter 6

Instantiation of the Framework

In this chapter, we instantiate the framework defined in Chapters 4 and 5 with the front-end described in Section 2.2. We first describe the instantiation of the inputs of the framework in Section 6.1. Then, in Section 6.2, we complete the instantiation of the framework by proving that our instantiation of the inputs satisfy all the conditions introduced in Chapter 4 of the framework.

6.1 Instantiation of the Inputs

We describe the instantiation of the inputs for the three building blocks in Section 4.1 respectively.

Instantiate The First Building Block

The first building block, which consists of the translation for front-end types, values, and states as shown in Figure 4.1, requires the definitions of the front-end's types, values, and heaps, the translation of these components into Viper, and a function `have_inv` as the inputs.

The types and values are T_f and V_f described in Equations (2.1) and (2.2) in Section 2.2.1. The heaps are instantiated as the permission heaps with a rational permission model (as introduced in Section 2.1.1). Concretely, a front-end permission heap is a pair of a permission mask $m : L \to [0, 1] \cap \mathbb{Q}$ and a partial heap $h : L \rightharpoonup V_f$. Here L represents the set of heap locations. As introduced in Section 2.2.2, our front-end (and also Viper) represents a heap location by a pair (r, f) (denoted as $r.f$) where r is a non-null reference and f is a field identifier.

Recall the definitions of compatibility (#), partial addition ($\oplus$), core ($|\cdot|$), stable, and stabilize operations of front-end states (which are the same as the definitions of these operations of Viper states) presented in Section 2.1.2:

- $(m_1, h_1) \# (m_2, h_2)$ if and only if: $\forall l \in L$, $m_1(l) + m_2(l) \leq 1$, and if both $h_1(l)$ and $h_2(l)$ are defined, then $h_1(l) = h_2(l)$.
- $(m_1, h_1) \oplus (m_2, h_2)$ is defined if and only if $(m_1, h_1) \# (m_2, h_2)$, and equals (m, h) where

$$m(l) = m_1(l) + m_2(l),$$
$$h(l) = \begin{cases} h_1(l), & \text{if } h_1(l) \text{ is defined,} \\ h_2(l), & \text{otherwise.} \end{cases}$$

- $|(m, h)| = (\mathbf{0}_L, h)$, where $\mathbf{0}_L$ is defined as $\mathbf{0}_L(l) = 0$ for any heap location $l \in L$.
- stable(m, h) if and only if for any l that $h(l)$ is defined, $m(l) > 0$.
- stabilize$(m, h) = (m, h_s)$ where

$$h_s(l) = \begin{cases} h(l), & \text{if } m(l) > 0, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

It is straightforward to prove that these definitions satisfy all the axioms of separation algebras for IDF, and thus front-end states (and also Viper states) are a separation algebra for IDF. Under these definitions, as we described in Section 2.1.2, we have the following equivalent characterization for the partial order relation $\succeq$ of front-end (and also Viper) permission heaps:

Proposition 6.1 (characterization for $\succeq$ of permission heaps) *For any permission heaps (m, h) and (m', h'), we have: $(m, h) \succeq (m', h') \iff \forall l \in L$, $m(l) \geq m'(l)$ and if $h'(l)$ is defined, then $h(l)$ is also defined and $h(l) = h'(l)$.*

The translation of front-end types is the identity for all types except that the type Rat in the front-end is translated to the type Real in Viper.

$$\text{tr}_\text{t}(\text{Int}) \triangleq \text{Int},$$
$$\text{tr}_\text{t}(\text{Bool}) \triangleq \text{Bool},$$
$$\text{tr}_\text{t}(\text{Ref}) \triangleq \text{Ref},$$
$$\text{tr}_\text{t}(\text{Rat}) \triangleq \text{Real}.$$

Similarly, the translation of front-end values is the identity, except that rational values are translated to real values.

$$\text{tr}_\text{v}(\text{Int } n) \triangleq \text{Int } n,$$
$$\text{tr}_\text{v}(\text{Bool } b) \triangleq \text{Bool } b,$$
$$\text{tr}_\text{v}(\text{Ref } l) \triangleq \text{Ref } l,$$

$$\mathrm{tr_v}(\text{Rat } r) \triangleq \text{Real } r.$$

The translation of front-end permission heaps operates separately on the permission mask and the partial heap. The translation of the permission mask is the identity, while the translation of the partial heap translates the value at each heap location using the function $\mathrm{tr_v}$ defined above. Concretely, for a front-end permission mask m and a partial heap h, $\mathrm{tr_h}(m, h) \triangleq (m, \mathrm{tr_p}(h))$ where we use $\mathrm{tr_p}$ to represent the translation of front-end partial heaps. For any front-end partial heap h, $\mathrm{tr_p}(h)$ is defined as follows: for each heap location $l \in L$, $\mathrm{tr_p}(h)(l)$ is defined if and only if $h(l)$ is defined, and when they are both defined, $\mathrm{tr_p}(h)(l) \triangleq \mathrm{tr_v}(h(l))$.

Since the translation of values of all the types except the rational type is bijective, the `have_inv` function for these types is undefined. For the rational type, $\texttt{have\_inv}(\text{Rat})(x) \triangleq (\exists a, b : \text{Int}.\, b \neq 0 \wedge x = a/b)$.

Instantiate the Second Building Block

The second building block, which consists of the translation for expressions and assertions as shown in Figure 4.2, requires the definitions of front-end expressions and primitive assertions, their interpretation and their translation to Viper, and a function fv that computes the set of free variables for expressions and primitive assertions as the inputs.

The expressions and primitive assertions are E_f and A_p described in Equations (2.3) and (2.5) in Section 2.2.1. The interpretation of expressions is defined in Definition 2.15 in Section 2.2.2. The interpretation of primitive assertions is defined in Figure 2.2 in Definition 2.16 in Section 2.2.2. The translation of front-end expression is the identity, with the only exception that rational literals are translated to real literals:

$$\begin{aligned}
&\mathrm{tr_e}(\text{Rat } r) \triangleq \text{Real } r,\\
&\mathrm{tr_e}(\text{Int } n) \triangleq \text{Int } n,\\
&\mathrm{tr_e}(\text{Bool } b) \triangleq \text{Bool } b,\\
&\mathrm{tr_e}(\text{Ref Null}) \triangleq \text{Ref Null},\\
&\mathrm{tr_e}(x) \triangleq x,\\
&\mathrm{tr_e}(\text{uop } e) \triangleq \text{uop } \mathrm{tr_e}(e),\\
&\mathrm{tr_e}(e_1 \text{ bop } e_2) \triangleq \mathrm{tr_e}(e_1) \text{ bop } \mathrm{tr_e}(e_2),\\
&\mathrm{tr_e}(e_1 \,?\, e_2 : e_3) \triangleq \mathrm{tr_e}(e_1) \,?\, \mathrm{tr_e}(e_2) : \mathrm{tr_e}(e_3),\\
&\mathrm{tr_e}(e.f) \triangleq \mathrm{tr_e}(e).f.
\end{aligned}$$

The translation of a front-end primitive assertion (recall that our front-end only has one kind of primitive assertion: the acc predicate $\mathrm{acc}(e.f, p)$) simply

translates each of its components:

$$\mathrm{tr_a}(\mathrm{acc}(e.f), p) \triangleq \mathrm{acc}(\mathrm{tr_e}(e).f, \mathrm{tr_e}(p)).$$

The set of free variables for front-end expressions is defined in Definition 2.17 in Section 2.2.3, and the set of free variables for front-end primitive assertions is defined as Equation (2.9) in Definition 2.18 in Section 2.2.3.

Instantiate the Last Building Block

The last building block, which consists of the translation of statements as shown in Figure 4.3, requires the definition of front-end primitive statements, the associated CSL rules, their translation to Viper, two functions fv and wr that compute the sets of free variables and written variables for primitive statements, and a well-formedness predicate wf as the inputs.

The primitive statements are S_p that consists of assignments to local variables, assignments to the heap, and allocating new heap locations, as we described in Equation (2.7) in Section 2.2.1. The CSL rules for these primitive statements consist of the rules in both Figures 2.4 and 2.6. We recall them in Figure 6.1.[1]

The translation of variable assignments and heap assignments just translates each component, while the **`alloc`** statement is translated to a sequential composition of **`havoc`** and **`inhale`** statements:

$$\begin{aligned}
&\mathrm{tr_s}(x \texttt{ := } e) \triangleq x \texttt{ := } \mathrm{tr_e}(e)\\
&\mathrm{tr_s}(e.f \texttt{ := } e') \triangleq \mathrm{tr_e}(e).f \texttt{ := } \mathrm{tr_e}(e')\\
&\mathrm{tr_s}(x \texttt{ := } \texttt{alloc}(f_1, \ldots, f_n)) \triangleq \texttt{havoc } x; \texttt{ inhale } \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)
\end{aligned}$$

Finally, the functions fv and wr for primitive statements are defined as in Equations (2.11) and (2.13) in Section 2.2.3, respectively. The well-formedness predicate for primitive statements is defined as in Definition 2.21 in Section 2.2.3.

6.2 Proof of the Instantiation

In this section, we prove that our instantiation of the inputs of the framework presented in Section 6.1 satisfies the conditions of the framework introduced in Chapter 4.

[1] Unlike the front-end CSL rules defined in Definition 2.22, which also include the rules for composite statements in Figure 2.5, the CSL rules for primitive statements and as the input to the framework only include the rules in Figure 2.4 and the structural rules in Figure 2.6. To make a distinction, we use $T \vdash_{\mathrm{FE}} \{P\}\ c\ \{Q\}$ to represent the front-end CSL rules in Definition 2.22, and $T \vdash_p \{P\}\ c\ \{Q\}$ to represent the CSL rules that are the input to the framework.

$$\frac{\begin{array}{c}A \text{ is self-framing} \\ e \text{ is well defined w.r.t. } A \qquad A \text{ frames } e\end{array}}{T \vdash_p \{A\}\ x := e\ \{A[x \mapsto e]\}}\ \text{(PrimAssignVar)}$$

$$\frac{\begin{array}{c}P = A * e.f \overset{1}{\mapsto} \_ \qquad P \text{ is self-framing} \\ e' \text{ is well defined w.r.t. } P \qquad P \text{ frames } e'\end{array}}{T \vdash_p \{P\}\ e.f := e'\ \{P[e.f \mapsto e']\}}\ \text{(PrimAssignField)}$$

$$\frac{}{T \vdash_p \{\text{True}\}\ x := \mathbf{alloc}(f_1, \ldots, f_n)\ \left\{ \circledast_{i=1}^{n} \text{acc}(x.f_i, 1) \right\}}\ \text{(PrimAlloc)}$$

$$\frac{\begin{array}{c}T \vdash_p \{P\}\ c\ \{Q\} \\ R \text{ is self-framing} \qquad \forall x \in \text{wr}(c).\, \text{indep}_T(R, x)\end{array}}{T \vdash_p \{P * R\}\ c\ \{Q * R\}}\ \text{(PrimFrame)}$$

$$\frac{T \vdash_p \{P\}\ c\ \{Q\} \qquad P' \text{ is self-framing} \qquad P' \Longrightarrow_T P}{T \vdash_p \{P'\}\ c\ \{Q\}}\ \text{(PrimConseqPre)}$$

$$\frac{T \vdash_p \{P\}\ c\ \{Q\} \qquad Q' \text{ is self-framing} \qquad Q \Longrightarrow_T Q'}{T \vdash_p \{P\}\ c\ \{Q'\}}\ \text{(PrimConseqPost)}$$

Figure 6.1: CSL rules for primitive statements.

Most conditions are trivially satisfied for our instantiation of the inputs. In the following we only show some interesting proofs for a subset of the conditions; all conditions have been formally proven in Isabelle.

For the first building block that consists of the translation of front-end types, values, and states, we prove the conditions (A8), (A9), and (A10).

(A8) For any front-end permission heaps h_1, h_2 and Viper permission heap H, if $\text{tr}_\text{h}(h_1) = H \oplus \text{tr}_\text{h}(h_2)$, then H is an image of tr_h. That is, there exists some front-end permission heap h, such that $H = \text{tr}_\text{h}(h)$.

Proof Let $h_i = (m_i, g_i)$ for $i = 1, 2$ and $H = (M, G)$, where m_1 and m_2 are front-end permission masks (i.e., the permission amount to each heap location is rational), M is a Viper permission mask (i.e., the permission amount to each heap location is real), g_1 and g_2 are front-end partial heaps, and G is a Viper partial heap. Then by the definition of tr_h in Section 6.1 and $\text{tr}_\text{h}(h_1) = H \oplus \text{tr}_\text{h}(h_2)$, we know that $\forall l \in L$, $m_1(l) = M(l) + m_2(l)$ and

$$\text{tr}_\text{p}(g_1)(l) = \begin{cases} G(l), & \text{if } G(l) \text{ is defined,} \\ \text{tr}_\text{p}(g_2)(l), & \text{otherwise.} \end{cases} \tag{6.1}$$

In order to prove that H is an image of $\mathrm{tr_h}$, we need to show that: $\forall l \in L$, $M(l)$ is a rational number in the interval $[0,1]$, and if $G(l)$ is defined, then it is in the image of $\mathrm{tr_v}$. We prove them separately in the following.

- Since $m_1(l)$ and $m_2(l)$ are both rationals, $M(l) = m_1(l) - m_2(l)$ is also rational. Moreover, as the permission amount to l in the Viper permission heap H, $M(l)$ naturally has its value in the interval $[0,1]$.
- If $G(l)$ is defined, then by Equation (6.1), we immediately know that $G(l) = \mathrm{tr_p}(g_1)(l) = \mathrm{tr_v}(g_1(l))$ is the image of front-end value $g_1(l)$ by the value translation function $\mathrm{tr_v}$. □

(A9) For any front-end permission heaps h_1, h_2 and Viper permission heap H, if $H \succeq \mathrm{tr_h}(h_1)$ and $H \succeq \mathrm{tr_h}(h_2)$, then there exists some front-end permission heap h such that $h \succeq h_i$ for $i = 1,2$ and $H \succeq \mathrm{tr_h}(h)$.

Proof Let $h_i = (m_i, g_i)$ for $i = 1,2$ and $H = (M, G)$, where m_1 and m_2 are front-end permission masks, M is a Viper permission mask, g_1 and g_2 are front-end partial heaps, and G is a Viper partial heap. From $H \succeq \mathrm{tr_h}(h_i) = (m_i, \mathrm{tr_p}(g_i))$ and Proposition 6.1, we know that for any heap location $l \in L$, $M(l) \geq m_i(l)$, and if $g_i(l)$ is defined, then $G(l)$ is also defined and $G(l) = \mathrm{tr_v}(g_i(l))$. As a result, if both $g_1(l)$ and $g_2(l)$ are defined, then $\mathrm{tr_v}(g_1)(l) = G(l) = \mathrm{tr_v}(g_2)(l)$. Further by the injectivity of $\mathrm{tr_v}$ (which is straightforward), we get $g_1(l) = g_2(l)$.

We construct the front-end permission heap h with the maximum permission between m_1 and m_2, and the partial heap of h is the union of g_1 and g_2. Concretely, let $h = (m, g)$, where for each heap location $l \in L$,

$$m(l) = \max(m_1(l), m_2(l)),$$
$$g(l) = \begin{cases} g_1(l), & \text{if } g_1(l) \text{ is defined,} \\ g_2(l), & \text{otherwise.} \end{cases}$$

Then $\forall l \in L$, $m(l) \geq m_i(l)$, and if $g_i(l)$ is defined, then $g(l) = g_i(l)$ (for $i = 1$ this is direct from the definition of g. For $i = 2$, if $g_1(l)$ is also defined, then as shown above, we have $g(l) = g_1(l) = g_2(l)$; if $g_1(l)$ is not defined, then $g(l) = g_2(l)$ follows the definition of g). As a result of Proposition 6.1, we have $h \succeq h_i$ for $i = 1,2$.

Moreover, from $M(l) \geq m_i(l)$ for $i = 1,2$ derived above, we have $M(l) \geq m(l)$. And note that by the definition of g, $g(l)$ is defined if and only if at least one of $g_1(l)$ and $g_2(l)$ is defined, and $g(l)$ equals the defined one. Therefore, $\mathrm{tr_p}(g)(l)$ equals the translation of the value of the defined one of $g_1(l)$ and $g_2(l)$. Since $G(l)$ also equals the translation of the value of the defined one of $g_1(l)$ and $g_2(l)$ as derived above, we have $G(l) = \mathrm{tr_p}(g)(l)$. Then by Proposition 6.1, we know that $H = (M, G) \succeq (m, \mathrm{tr_p}(g)) = \mathrm{tr_h}(h)$, which concludes our proof. □

(A10) For any Viper permission heap H, if it is upper bounded by the translation of some front-end permission heap h_b, i.e., $\text{tr}_\text{h}(h_b) \succeq H$, then its core is the translation of some front-end permission heap. That is, there exists some front-end permission heap heap h such that $\text{tr}_\text{h}(h) = |H|$.

Proof Let $h_b = (m_b, g_b)$ and $H = (M, G)$, where m_b is a front-end permission mask, M is a Viper permission mask, g_b is a front-end partial heap, and G is a Viper partial heap. From $\text{tr}_\text{h}(h_b) \succeq H$ and Proposition 6.1, we know that for any $l \in L$, if $G(l)$ is defined, then $G(l) = \text{tr}_\text{p}(g_b)(l) = \text{tr}_\text{v}(g_b(l))$. That is, G is the translation of some front-end partial heap g defined as: $\forall l \in L$,

$$g(l) = \begin{cases} g_b(l), & \text{if } G(l) \text{ is defined,} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Then choose $h = (\mathbf{0}_L, g)$. It is straightforward to show that $\text{tr}_\text{h}(h) = |H|$, which concludes the proof. □

With all the conditions for the first building block proved, we can now use all the conditions and results about front-end types, values, states, and their translations tr_t, tr_v, and tr_s stated in Chapters 4 and 5.

For the second building block that consists of the translation for expressions and assertions, we prove the condition (A13).

(A13) All front-end primitive assertions are backward satisfiable.

Proof The only class of primitive assertions in our front-end is of the form $\text{acc}(e.f, p)$, where e and p are front-end expressions. By the definition of backward satisfiability (Definition 5.16), to show that $\text{acc}(e.f, p)$ is backward satisfiable, we fix any front-end state ω_b and Viper state ω_v such that $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$ and $\omega_v \vDash_V \text{tr}_\text{a}(\text{acc}(e.f, p)) = \text{acc}(\text{tr}_\text{e}(e).f, \text{tr}_\text{e}(p))$, and need to find some front-end state ω_f such that $\omega_f \vDash_f \text{acc}(e.f, p)$ and $\omega_v \succeq \text{tr}_\text{s}(\omega_f)$.

Let $\omega_v = (\sigma_v, h_v)$, where the permission heap $h_v = (m_v, g_v)$. From $\omega_v \vDash_V \text{acc}(\text{tr}_\text{e}(e).f, \text{tr}_\text{e}(p))$, by the interpretation of acc in Viper (which is the same as the two rules in Figure 2.2 except that the interpretations of expressions are of Viper and all the rationals become reals), we know that there exist some reference r and real number v such that $\langle \text{tr}_\text{e}(e), \omega_v \rangle \Downarrow_V \text{Ref } r$, $\langle \text{tr}_\text{e}(p), \omega_v \rangle \Downarrow_V \text{Real } v$, $v \geq 0$, and if $v \neq 0$ then $r \neq \text{Null}$ and $m_v(r.f) \geq v$.

To construct the desired ω_f, we use (L6) to obtain the preimage of the core of ω_v whose translation evaluates the expressions $\text{tr}_\text{e}(e)$ and $\text{tr}_\text{e}(p)$ to the same results as ω_v. Then we add the precise permission amount to the heap location $r.f$.

Concretely, by (L6) and $\text{tr}_\text{s}(\omega_b) \succeq \omega_v$, there exists some front-end state ω_c such that $\text{tr}_\text{s}(\omega_c) = |\omega_v|$. In our instantiation, it is straightforward to show that the permission mask of ω_c is $\mathbf{0}_L$.

It is straightforward to prove that (A16) and (A11) hold for our instantiation. As a result, we get $\langle \mathrm{tr_e}(e), |\omega_v| \rangle \Downarrow_V \text{Ref } r$ and $\langle \mathrm{tr_e}(p), |\omega_v| \rangle \Downarrow_V \text{Real } v$ using (A16), and further $\langle e, \omega_c \rangle \Downarrow_f \text{Ref } r$ and $\langle p, \omega_c \rangle \Downarrow_f \text{Rat } v$ using (A11). Now we consider the two cases $v = 0$ and $v > 0$:

- If $v = 0$, then by the rule (AccZero), we already have $\omega_c \vDash_f \mathrm{acc}(e.f, p)$. And since $\omega_v \succeq |\omega_v| = \mathrm{tr_s}(\omega_c)$, we can conclude the existence of ω_f with ω_c and conclude the proof.
- If $v > 0$, then $r \neq \text{Null}$. The fact $\langle p, \omega_c \rangle \Downarrow_f \text{Rat } v$ further implies that v is a rational number. As a result, let ω_f be the state obtained by adding v permission amount to $r.f$ in ω_c (from $m_v(r.f) \geq v$ we know that $v \leq 1$. Therefore, adding v permission amount to some heap location in ω_c whose permission mask is $\mathbf{0}_L$ does not cause excess of the permission to any heap location). That is, denote $\omega_c = (\sigma_c, (\mathbf{0}_L, g_c))$, then $\omega_f = (\sigma_c, (m_f, g_c))$ where

  $$m_f(l) = \begin{cases} v, & \text{if } l = r.f, \\ 0, & \text{otherwise.} \end{cases}$$

  It is straightforward to show that ω_f preserves the evaluation of expressions e and p in ω_c. That is, $\langle e, \omega_f \rangle \Downarrow_f \text{Ref } r$ and $\langle p, \omega_f \rangle \Downarrow_f \text{Rat } v$. As a result, by the rule (AccPos), we have $\omega_f \vDash_f \mathrm{acc}(e.f, p)$. Moreover, $\forall l \in L$, $m_v(l) \geq m_f(l)$ (for $l \neq r.f$, this is trivial since $m_f(l) = 0$; for $l = r.f$, we have $m_v(l) \geq v = m_f(l)$), and if the partial heap of $\mathrm{tr_s}(\omega_f)$ at heap location l ($\mathrm{tr_p}(g_c)(l)$) is defined, then by $\omega_v \succeq |\omega_v| = \mathrm{tr_s}(\omega_c)$, $g_v(l)$ is also defined and $g_v(l) = \mathrm{tr_p}(g_c)(l)$. As a result of Proposition 6.1, $\omega_v \succeq \mathrm{tr_s}(\omega_f)$. We can thus conclude the proof with the front-end state ω_f we constructed as above. □

With all the conditions for the second building block proved, we can now use all the conditions and results about front-end expressions and assertions, and their translations $\mathrm{tr_e}$ and $\mathrm{tr_a}$ defined in Chapters 4 and 5.

For the last building block that consists of the translation of statements, we prove the condition (A19).

(A19) For any front-end type context T, front-end primitive statement c, and any monotone semantic Viper assertions P and Q, if $\mathrm{wf}_T(c)$ and $\mathrm{tr_{tc}}(T) \vdash_{\text{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$, then $T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$.

Proof We prove the three cases of primitive statements respectively.

Case Variable Assignment: $c = (x := e)$. Then

$$\mathrm{tr_c}(c) = (x := \mathrm{tr_e}(e))\,.$$

Using the inversion rule (Lemma 2.24) for variable assignments in Viper (the Viper SL rule for variable assignments is similar to the rule (AssignVar), except that the evaluation of expressions is in Viper), from $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$ we obtain

- P and Q are self-framing.
- $\mathrm{tr_e}(e)$ is well defined w.r.t. P.
- P frames $\mathrm{tr_e}(e)$.
- $Q = P[x \mapsto \mathrm{tr_e}(e)]$.

Now we want to use the rule (PrimAssignVar) for the front-end variable assignment statement c. Let us check its premises:

- Using Lemma 5.31, we get self-framedness of $\mathrm{btr}(P)$.
- It is straightforward to derive that e is well defined w.r.t. $\mathrm{btr}(P)$ from the fact that $\mathrm{tr_e}(e)$ is well defined w.r.t. P, using the condition (A11) that the translation of expressions preserves their semantics in the front-end.
- It is straightforward to derive that $\mathrm{btr}(P)$ frames e from the fact that P frames $\mathrm{tr_e}(e)$ using the condition (A11) that the translation of expressions preserves their semantics in the front-end, and the result (L3) of $\mathrm{tr_s}$.

As a result of the rule (PrimAssignVar), we get

$$T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(P)[x \mapsto e]\}\,.$$

Finally, we want to user the rule (PrimConseqPost) to adjust the post-condition. For this purpose (and since self-framedness of $\mathrm{btr}(Q)$ is guaranteed by self-framedness of Q derived above and Lemma 5.31 that btr preserves self-framedness), we need to show that $\mathrm{btr}(P)[x \mapsto e] \Longrightarrow_T \mathrm{btr}(Q)$.

In this paragraph, we prove the stronger result $\mathrm{btr}(P)[x \mapsto e] \implies \mathrm{btr}(Q)$. Fix any front-end state ω that satisfies $\mathrm{btr}(P)[x \mapsto e]$. Then there exist ω_0 and v such that $\mathrm{btr}(P)(\omega_0)$ holds (then $P(\mathrm{tr_s}(\omega_0))$ holds by the definition of btr in Definition 5.13), $\langle e, \omega_0\rangle \Downarrow_f v$ and $\omega = \omega_0(x \mapsto v)$. By the condition (A11) that the translation of expressions preserves their semantics in the front-end, we get $\langle \mathrm{tr_e}(e), \mathrm{tr_s}(\omega_0)\rangle \Downarrow_V \mathrm{tr_v}(v)$. As a result, $\mathrm{tr_s}(\omega) = (\mathrm{tr_s}(\omega_0))(x \mapsto \mathrm{tr_v}(v))$ satisfies $Q = P[x \mapsto \mathrm{tr_e}(e)]$. By the definition of btr in Definition 5.13, ω satisfies $\mathrm{btr}(Q)$.

Applying the rule (PrimConseqPost) then gives us

$$T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$$

as required.

Case Heap Assignment: $c = (e.f \texttt{ := } e')$. Then

$$\mathrm{tr_c}(c) = \left(\mathrm{tr_e}(e).f \texttt{ := } \mathrm{tr_e}(e')\right).$$

Using the inversion rule (Lemma 2.24) for heap assignments in Viper (the Viper SL rule for heap assignments is similar to the rule (AssignField), except that the evaluation of expressions is in Viper), from $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$ we obtain a semantic Viper assertion R such that

- $P = R * \mathrm{tr_e}(e).f \overset{1}{\mapsto} \_$.
- P is self-framing.
- $\mathrm{tr_e}(e')$ is well defined w.r.t. P.
- P frames $\mathrm{tr_e}(e')$.
- $Q = P[\mathrm{tr_e}(e).f \mapsto \mathrm{tr_e}(e')]$.

Similarly to the "**Case Variable Assignment**", we want to use the rule (PrimAssignField) for the front-end variable assignment statement c. In order to check its premises, we first need to find a front-end semantic assertion R_f such that $\mathrm{btr}(P) = R_f * e.f \overset{1}{\mapsto} \_$. To show the equality we will prove separately $R_f * e.f \overset{1}{\mapsto} \_ \implies \mathrm{btr}(P)$ and $\mathrm{btr}(P) \implies R_f * e.f \overset{1}{\mapsto} \_$ to conclude $\mathrm{btr}(P) = R_f * e.f \overset{1}{\mapsto} \_$. The former entailment is easy, while for the latter, we will again use the technique of increasing one part of the state and decreasing the other part as in the proof of the "**Inductive Case** $*$" of Lemma 5.24 in Section 5.5.2. For this purpose, we first need to monotonize R. Let $R_v = M(R)$. For the following we prove $\mathrm{btr}(P) = \mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_$ (i.e., we choose $\mathrm{btr}(R_v)$ as the R_f).

- We first prove $\mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_ \implies \mathrm{btr}(P)$: since P is monotone (promised by the premise of (A19)), it is straightforward to show that $P = R * \mathrm{tr_e}(e).f \overset{1}{\mapsto} \_ = R_v * \mathrm{tr_e}(e).f \overset{1}{\mapsto} \_$. Then by Lemma 5.35, we directly get $\mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_ \implies \mathrm{btr}(P)$ (it is straightforward to show that $\mathrm{btr}(\mathrm{tr_e}(e).f \overset{1}{\mapsto} \_) = e.f \overset{1}{\mapsto} \_$).
- We then prove $\mathrm{btr}(P) \implies \mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_$: fix any front-end state ω that satisfies $\mathrm{btr}(P)$. Then we know that $\mathrm{tr_s}(\omega)$ satisfies $P = R_v * \mathrm{tr_e}(e).f \overset{1}{\mapsto} \_$. Then there exist Viper states α_v and β_v such that $\mathrm{tr_s}(\omega) = \alpha_v \oplus \beta_v$, α_v satisfies R_v, and β_v satifies $\mathrm{tr_e}(e).f \overset{1}{\mapsto} \_$. Next, note that $e.f \overset{1}{\mapsto} \_$ is exactly the interpretation of $\mathrm{acc}(e.f, 1)$, which is backward satisfiable. As a result, we obtain some front-end state β_f that satisfies $e.f \overset{1}{\mapsto} \_$ and $\beta_v \succeq \mathrm{tr_s}(\beta_f)$. Let δ be the

difference between β_v and $\mathrm{tr_s}(\beta_f)$ (i.e., $\beta_v = \delta \oplus \mathrm{tr_s}(\beta_f)$). Then by associativity of PCMs, $\mathrm{tr_s}(\omega) = \alpha_v \oplus (\delta \oplus \mathrm{tr_s}(\beta_f)) = (\alpha_v \oplus \delta) \oplus \mathrm{tr_s}(\beta_f)$. By (L4), there exists some front-end state α_f such that $\mathrm{tr_s}(\alpha_f) = \alpha_v \oplus \delta$. Then by monotonicity of $R_v = M(R)$ and that α_v satisfies R_v, we get that $\mathrm{tr_s}(\alpha_f)$ also satisfies R_v, which means α_f satisfies $\mathrm{btr}(R_v)$. Together with $\omega = \alpha_f \oplus \beta_f$ (from $\mathrm{tr_s}(\omega) = \mathrm{tr_s}(\alpha_f) \oplus \mathrm{tr_s}(\beta_f)$ and (L1)) and that β_f satisfies $e.f \overset{1}{\mapsto} \_$, we know that ω satisfies $\mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_$. This concludes the proof of $\mathrm{btr}(P) \implies \mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_$.

Next, similarly to the "**Case Variable Assignment**", we can show that $\mathrm{btr}(P)$ is self-framing, e' is well-defined w.r.t. $\mathrm{btr}(P)$, and $\mathrm{btr}(P)$ frames e', which together with $\mathrm{btr}(P) = \mathrm{btr}(R_v) * e.f \overset{1}{\mapsto} \_$ allow us to apply the rule (PrimAssignField) to get

$$T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(P)[e.f \mapsto e']\}.$$

Finally, similarly to the "**Case Variable Assignment**", we can show that $\mathrm{btr}(Q)$ is self-framing, and $\mathrm{btr}(P)[e.f \mapsto e'] \implies \mathrm{btr}(P[\mathrm{tr_e}(e).f \mapsto \mathrm{tr_e}(e')]) = \mathrm{btr}(Q)$, which allow us to apply the rule (PrimConseqPost) to get

$$T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$$

as required.

Case Allocation: $c = (x := \mathbf{alloc}(f_1, \ldots, f_n))$. Then

$$\mathrm{tr_c}(c) = (\mathbf{havoc}\ x; \mathbf{inhale}\ \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)).$$

From the rule (PrimAlloc) we have

$$T \vdash_p \{\mathrm{True}\}\ c\ \left\{\circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)\right\}. \tag{6.2}$$

In order to get $T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$ from it, we want to use a similar approach as steps 3, 4, and 5 in the proof of Lemma 5.26: adjust the pre- and postconditions by first deriving a front-end assertion as the frame to apply the rule (PrimFrame), and then applying the rules (PrimConseqPre) and (PrimConseqPost).

We first choose the front-end assertion that we will use as the frame to apply the rule (PrimFrame). Let $R_f = \exists x : T(x).\, \mathrm{btr}(P)$. Since P, which is the precondition of the Viper SL proof $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$, is self-framing by Fact 2.27, by Lemmas 5.31 and 5.32, R_f is also self-framing. Moreover, R_f satisfies the independence requirement of the

rule (PrimFrame) because of its quantification over $\mathrm{wr}(c) = \{x\}$. As a result, applying the rule (PrimFrame) to Equation (6.2) gives us

$$T \vdash_p \left\{M(R_f)\right\}\ c\ \left\{\circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1) * R_f\right\}, \tag{6.3}$$

where we note that $\mathrm{True} * A = M(A)$ for any assertion A. Moreover, it is straightforward to prove that $A \implies M(A)$ for any assertion A. Together with the fact we showed above that R_f is self-framing, we can apply the rule (PrimConseqPre) to Equation (6.3) and get

$$T \vdash_p \left\{R_f\right\}\ c\ \left\{R_f * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)\right\}, \tag{6.4}$$

where we rewrite $\circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1) * R_f$ to $R_f * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)$ in the postcondition since the separating conjunction is commutative.

To adjust the precondition to $\mathrm{btr}(P)$, we use the rule (PrimConseqPre) again. Since P, the precondition of the Viper SL proof $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$, is self-framing by Fact 2.27, and btr preserves self-framedness as stated in Lemma 5.31, $\mathrm{btr}(P)$ is self-framing. Therefore, to apply the rule (PrimConseqPre), we are left to prove $\mathrm{btr}(P) \Longrightarrow_T R_f = \exists x : T(x).\, \mathrm{btr}(P)$: fix some state ω that is well-typed w.r.t. T and satisfies $\mathrm{btr}(P)$. Then the value of x in ω is of type $T(x)$ (well-formedness of c guarantees that $T(x)$ is defined, which implies that variable x has to have some value in the well-typed state ω). The existence of such value immediately proves that ω satisfies R_f. Therefore, $\mathrm{btr}(P) \Longrightarrow_T R_f = \exists x : T(x).\, \mathrm{btr}(P)$, and we can apply (PrimConseqPre) to Equation (6.4) and get

$$T \vdash_p \left\{\mathrm{btr}(P)\right\}\ c\ \left\{R_f * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1)\right\}. \tag{6.5}$$

To adjust the postcondition to $\mathrm{btr}(Q)$, we want to use the rule (PrimConseqPost). Since Q, the postcondition of the Viper SL proof $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$, is self-framing by Fact 2.27, and btr preserves self-framedness as stated in Lemma 5.31, $\mathrm{btr}(Q)$ is self-framing. Therefore, to apply the rule (PrimConseqPost), we are left to prove $R_f * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1) \Longrightarrow_T \mathrm{btr}(Q)$. For the following, we prove the stronger result $R_f * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1) \implies \mathrm{btr}(Q)$:

Proof Using the inversion rule (Lemma 2.24) for **`havoc`** and **`inhale`** statements and the sequential composition, from the Viper SL proof $\mathrm{tr_{tc}}(T) \vdash_{\mathrm{VPR}} \{P\}\ \mathrm{tr_c}(c)\ \{Q\}$ we get

$$Q = (\exists x : \mathrm{tr_{tc}}(T)(x).\, P) * \circledast_{i=1}^{n} \mathrm{acc}(x.f_i, 1).$$

It is straightforward to prove that

$$\mathrm{btr}\left(\overset{n}{\underset{i=1}{\circledast}} \mathrm{acc}(x.f_i, 1)\right) = \overset{n}{\underset{i=1}{\circledast}} \mathrm{acc}(x.f_i, 1).$$

Moreover, we prove in the next paragraph that

$$R_f = \exists x : T(x).\, \mathrm{btr}(P) \implies \mathrm{btr}(\exists x : \mathrm{tr_{tc}}(T)(x).\, P).$$

Fix any front-end state that satisfies R_f, then there exists some front-end value v of type $T(x)$ such that $\omega(x \mapsto v)$ satisfies $\mathrm{btr}(P)$. Then $\mathrm{tr_s}(\omega(x \mapsto v)) = (\mathrm{tr_s}(\omega))(x \mapsto \mathrm{tr_v}(v))$ satisfies P, which implies that $\mathrm{tr_s}(\omega)$ satisfies $\exists x : \mathrm{tr_{tc}}(T)(x).\, P$. By the definition of btr (Definition 5.13), ω satisfies $\mathrm{btr}(\exists x : \mathrm{tr_{tc}}(T)(x).\, P)$, which concludes the proof of $R_f \implies \mathrm{btr}(\exists x : \mathrm{tr_{tc}}(T)(x).\, P)$.

As a result, we have

$$\begin{aligned}
& R_f * \overset{n}{\underset{i=1}{\circledast}} \mathrm{acc}(x.f_i, 1) \\
\implies\ & \mathrm{btr}(\exists x : \mathrm{tr_{tc}}(T)(x).\, P) * \overset{n}{\underset{i=1}{\circledast}} \mathrm{acc}(x.f_i, 1) && \text{(by Lemma 5.34)} \\
\implies\ & \mathrm{btr}\left((\exists x : \mathrm{tr_{tc}}(T)(x).\, P) * \overset{n}{\underset{i=1}{\circledast}} \mathrm{acc}(x.f_i, 1)\right) && \text{(by Lemma 5.35)} \\
=\ & \mathrm{btr}(Q). && \square
\end{aligned}$$

Then, applying the rule (PrimConseqPost) to Equation (6.5), we finally get

$$T \vdash_p \{\mathrm{btr}(P)\}\ c\ \{\mathrm{btr}(Q)\}$$

as required. $\square$

Chapter 7

Discussion and Conclusion

In this thesis, we developed a general framework for proving the soundness of CSL-based front-ends in the Isabelle theorem prover [9]. In contrast to Arlt's approach [8], which directly connects the front-end's and Viper's operational semantics, our framework makes use of the existing sound CSL for front-ends and SL proof representation of Viper programs, and proves sound the translation by converting Viper SL proofs to front-end CSL proofs.

Our framework abstracts over many aspects of the front-end, including front-end types, values, states, expressions, primitive assertions, and primitive statements, by taking these components and their translation to Viper as inputs and imposing minimal conditions on the inputs. From these inputs, our framework allows the front-end to use many common SL assertion connectives in its assertion language, and control structures for front-end composite statements. With further requirements on the semantics of the assertion connectives and the CSL rules of the control structures, the framework is able to output an executable translation of the front-end to Viper and a proof of the soundness of the translation.

Finally, we also instantiated our framework with a concrete CSL based front-end in Isabelle, which shows that our framework is of practical use.

7.1 Comparison with the Previous Approach

Unlike Arlt's approach [8], which proves sound the translation of front-ends to Viper by directly connecting the front-end's and Viper's operational semantics and converting an execution in Viper to one in the front-end, we make use of the existing sound CSL for front-ends and SL proof representation of Viper programs and prove the soundness of the translation by converting Viper SL proofs to front-end CSL proofs. Both approaches have their own

advantages and disadvantages. We list the strengths and weaknesses of our approach compared with Arlt's approach [8] in the following.

Strengths

Being able to hide low-level details behind SL rules, our approach simplifies the soundness proof of the translation. For example, as shown in Example 3.3, the solution of Example 3.2 using our approach is much simpler and more straightforward. It does not need to consider how to choose a feasible one from the many correct Viper executions and convert it to an execution in the front-end, which the solution using Arlt's approach has to do. Instead, the solution using our approach derives a front-end CSL proof by directly converting the pre- and postconditions of the Viper SL proof.

Moreover, hiding the low-level details also allows the proof to focus on and emphasize the key challenges. For example, our proof in Section 5.5 focuses on converting the pre and postconditions of Viper SL proofs back to the front-end, and defines the properties and proves the auxiliary lemmas to ensure the existence of a feasible conversion.

Weaknesses

Our approach abstracts the front-end and Viper by the front-end CSL and Viper SL. This abstraction prevents the soundness proof of the translation from using the features that are captured in the low-level details of the front-end and Viper but are abstracted out by the front-end CSL or Viper SL. For example, in concrete Viper, verification of the statement "**inhale** $I \wedge e$" implies that I frames e, while a Viper SL proof of the statement does not. As a result, to prove sound the translation of the loop "**while** e **inv** I **do** c", we have to enforce "I frames e" in the well-formedness requirement of the statement, while it is actually impled by the verification of the translation in Viper.

7.2 Future Work

Despite the generality of our framework, it still has some limitations and can be further improved. We list some of the limitations and possible improvements here.

General type translation. Our framework takes care of many type mismatches between front-ends and Viper (such as the mismatch between the front-end's rational type and Viper's real type) by imposing certain conditions on the translation of front-ends to Viper. However, there are still some common type mismatches that are not supported by our framework since

they violate the conditions required by the framework. For example, a front-end could have natural numbers, or bounded integers; both types could be encoded using the larger type Int, but this violates the semantics-preserving condition (A11) of the framework presented in Section 4.3. As a concrete example, the expression $x - 1$ where x is a variable containing a *natural number* or a *bounded integer* is undefined when x takes the value 0 in the front-end. However, in Viper, $x - 1$ evaluates normally to -1 when x takes the value 0. This brings further challenges in the translation of assignments. For example, if we translate the assignment "$x := x - 1$" in the front-end language to "$x := x - 1$" in Viper, it would be unsound, since the assignment trivially verifies in Viper, while it runs into an error when x takes the value 0 in the front-end.

In order to support more type mismatches and provide more generality, one could try to weaken the conditions on the inputs of the framework while adjusting the translation of the front-end to Viper at the same time, such that the conditions are still strong enough to prove sound the translation.

Further extension of the assertion language. Our framework abstracts over front-end primitive assertions by taking them as input, and allows many common SL connectives. Nevertheless, there are still some constructs that are not supported by our framework, such as separating implications and recursively-defined predicates.

Getting rid of non-syntactic well-formedness requirements in the final theorem. For loops and parallel compositions, our framework imposes additional well-formedness requirements on the front-end program. Some of these requirements are semantic. As a result, we are not able to generate an executable tool that does both the well-formedness check and the translation, such that as long as the tool does not report any well-formedness issues about the front-end program and the translation of the front-end program is verified by Viper, the front-end program will be correct. It would thus be highly beneficial if we could make all the well-formedness requirements syntactic, since this would enable a static automatic check of well-formedness of front-end programs, and we would then have the executable tool described above, such that if the tool does not report any errors and produces a translation that is verified in Viper, then the front-end program is correct.

More complete translation. The type mismatch between front-ends and Viper makes our translation incomplete in some cases, as shown in Example 3.5 in Section 3.2.2. Modifying the translation to make it more complete would also be a possible improvement for our framework.

Supporting more front-end features. Although the abstraction of our framework over the many aspects of front-ends allows it to characterize a wide range of practical front-ends, there are still some restriction imposed by the framework on the front-end. For example, our framework requires the front-end to declare all the local variables at the beginning of the program. One could thus improve the framework by allowing local variable declarations. Moreover, one could also add static types for fields, which do not exist in the current framework. Finally, support for more advanced CSL features and constructs (e.g., resource invariants and conditional critical regions used by O'Hearn [2] and Brookes [14]) would also be a possible extension to our framework.

Bibliography

[1] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2002, pp. 55–74.

[2] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical computer science*, vol. 375, no. 1, pp. 271–307, 2007.

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, Springer, 2016, pp. 41–62.

[4] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of Go programs," in *International Conference on Computer Aided Verification*, Springer, 2021, pp. 367–379.

[5] M. Eilers and P. Müller, "Nagini: A static verifier for Python," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, Springer, 2018, pp. 596–603.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[7] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors tool set: Verification of parallel and concurrent software," in *Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings 13*, Springer, 2017, pp. 102–110.

[8] E. Arlt, "A formally verified automatic verifier for concurrent programs," Master's thesis, ETH Zürich, May 2023.

[9] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[10] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, R. Cousot, Ed., ser. Lecture Notes in Computer Science, vol. 2694, Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4. [Online]. Available: https://doi.org/10.1007/3-540-44898-5%5C_4.

[11] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local action and abstract separation logic," in *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, 2007, pp. 366–378. DOI: 10.1109/LICS.2007.30.

[12] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," in *European Symposium on Programming*, Springer, 2011, pp. 439–458.

[13] V. Vafeiadis, "Concurrent separation logic and operational semantics," *Electronic Notes in Theoretical Computer Science*, vol. 276, pp. 335–351, 2011.

[14] S. Brookes, "A semantics for concurrent separation logic," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 227–270, 2007.

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:

A General Approach to Formally Verify Viper Front-ends

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

Last name(s):	**First name(s):**
Ling	Hongyi

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, 23.04.2024	Hongyi Ling

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