ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Formally Verified Automatic Verifier for Concurrent Programs

Master Thesis

Ellen Arlt

May 17, 2023

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

Abstract

Writing correct concurrent programs is challenging, because one often has to consider a large number of possible orders of execution due to the unpredictability of how the concurrent sections will interleave. The same applies when verifying concurrent programs. Concurrent separation logic (CSL) provides a program verification technique that allows one to elegantly prove that a concurrent program is indeed correct. However, writing CSL proofs manually is time consuming. As a result, a variety of tools have been developed to automate CSL proofs. Among such tools some are based on the Viper infrastructure. These are Viper front-ends.

Viper front-ends are verifiers for real-world programming languages which translate input programs and their specifications into Viper. A front-end is considered sound if the correctness of the translated program implies the correctness of the input program with respect to its specifications. Proving a front-end sound is non-trivial, especially if that front-end supports concurrency. There are Viper front-ends supporting concurrency for which the translation into Viper has been formally proven correct on paper. Nevertheless, none of such correctness proofs have so far been fully mechanized in an interactive theorem prover.

In this thesis, we provide a mechanically formalized Viper front-end for a programming language that supports concurrency. Our front-end is the first such front-end that has been formally proved to be sound in the Isabelle theorem prover.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisors Thibault Dardinier and Gaurav Parthasarathy for their support, valuable feedback and for providing me with new insights during our meetings. Furthermore, I am very grateful to Prof. Dr. Peter Müller for the opportunity to work on this project.

Many thanks also to my family for always being there for me.

Contents

Chapter 1

Introduction

Ridding software of bugs has always been an issue in programming. Yet, showing that a piece of software is bug-free is an even more challenging thing to do. That holds true especially when that piece of software uses concurrency. When verifying concurrent programs one often has to consider a large number of possible orders of execution due to the unpredictability of how the concurrent sections will interleave. For that reason concurrent programs are quite prone to bugs due to issues such as race conditions. So how can one ensure that a piece of concurrent software correctly does what it is supposed to do?

One approach to resolve this is via concurrent separation logic (CSL) [1, 2], a logic used to reason about heap-manipulating concurrent programs. Concurrent separation logic provides a way to prove Hoare triples $\{P\}\, c\, \{Q\}$, where P is a precondition, c is code to be executed and Q is a postcondition. A Hoare triple $\{P\}\, c\, \{Q\}$ holds if whenever P holds before execution, then Q holds after execution of c. Concurrent separation logic uses the notion of *exclusive ownership* of parts of the heap to avoid explicitly reasoning about all possible interleavings of concurrently executed code.

One of the main rules from concurrent separation logic to reason about concurrent sections specifically is the parallel rule. It can be stated as follows[1].

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \quad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1\ *\ P_2\}\ c_1 \parallel c_2\ \{Q_1\ *\ Q_2\}}$$

Here the operator $*$ denotes the *separating conjunction*. That is, for two assertions A_1 and A_2 the separating conjunction $A_1 * A_2$ holds if the heap

[1] The parallel rule actually requires as another premise that the set of variables written to in c_1 is disjoint from the set of variables freely occurring in P_2, c_2 and Q_2 and the set of variables written to in c_2 is disjoint from the set of variables freely occurring in P_1, c_1 and Q_1. That is, variable writes should not interfere with the execution of the other thread.

```
1  x := alloc(f);
2  y := alloc(f);
3    x.f := 2   ||   y.f := 3
```

Listing 1: Example of a concurrent program.

```
1  {True}
2  x := alloc(f);
3  {x.f ↦ _}
4  y := alloc(f);
5  {(x.f ↦ _) ∗ (y.f ↦ _)}
6   {(x.f ↦ _)}        {(y.f ↦ _)}
7     x.f := 2   ||   y.f := 3
8   {(x.f ↦ 2)}        {(y.f ↦ 3)}
9  {(x.f ↦ 2) ∗ (y.f ↦ 3)}
```

Listing 2: Proof outline of a concurrent program in Listing 1.

can be divided into two disjoint sections such that A_1 holds having access to only the first section and A_2 holds when having access to only the second section of the heap. In other words, each assertion can be proved to hold when only retaining the information from their section of the heap and disregarding the rest of the heap. The parallel rule therefore states that if Hoare triples $\{P_1\}\, c_1\, \{Q_1\}$ and $\{P_2\}\, c_2\, \{Q_2\}$ hold, then we can conclude that if preconditions P_1 and P_2 hold on disjoint parts of the heap then after concurrently executing c_1 and c_2 postconditions Q_1 and Q_2 hold on disjoint parts of the heap. On a high level that means that if c_1 and c_2 operate on disjoint parts of the heap, then their executions do not interfere with each other.

Consider as an example the code snippet in Listing 1. The code snippet shows an example where we allocate the field f for each of two reference variables (i.e. they are Null or point to a memory address), and then concurrently set the field f of one variable to 2 and the other to 3. Listing 2 gives a proof outline for the Hoare triple

$$\{True\}\, c\, \{(x.f \mapsto 2) * (y.f \mapsto 3)\},$$

where c is the code from Listing 1, via the annotations shown in orange. The *poinst-to assertion* $(x.f \mapsto 2)$ expresses that the current process or thread *owns* the heap location associated with $x.f$, i.e it may access and change it, and that heap location contains value 2. The notation $(x.f \mapsto \_)$ expresses only ownership.
We start with the precondition in line 1. After the first allocation we know that $x.f$ is allocated. That also means the execution process gains ownership of the heap location associated with $x.f$. Thus, after the first allocation

$(x.f \mapsto \_)$ holds. With the second allocation the process also gains ownership of the heap location associated with $y.f$. By the semantics of allocation we are using it is ensured that both heap locations are different. Thus, the separating conjunction in line 5 holds after the two allocations.
It is easy to see that the Hoare triples $\{x.f \mapsto \_\}\, x.f := 2\, \{x.f \mapsto 2\}$ and $\{y.f \mapsto \_\}\, y.f := 3\, \{y.f \mapsto 3\}$ hold. Hence we can use the parallel rule to obtain the Hoare triple

$$\{(y.f \mapsto \_) * (x.f \mapsto \_)\}\, x.f := 2 \;\parallel\; y.f := 3\, \{(x.f \mapsto 2) * (y.f \mapsto 3)\}.$$

As we have established that before the concurrent sections $(y.f \mapsto \_) * (x.f \mapsto \_)$ holds, we obtain that after execution of the concurrent sections the desired postcondition $(x.f \mapsto 2) * (y.f \mapsto 3)$ holds.

The proof for Listing 1 is simple, but for more complex programs such proofs become very hard to do manually. Therefore there is a lot of merit in automating this process. One tool that performs automatic verification is Viper [3]. Viper [3] is a verification infrastructure for permission-based reasoning and includes the Viper intermediate verification language (IVL), which is based on separation logic, and an automatic verifier to check the correctness of Viper IVL programs with respect to user-provided specifications (e.g., pre- and postconditions).

The Viper infrastructure has been used as a foundation to develop quite a few verifiers for real-world programming languages. This has been achieved by developing Viper front-ends which translate input programs and their specifications into Viper IVL programs so that correctness of the Viper program implies correctness of the input program. There already exist multiple Viper front-ends, examples of which include Gobra [4] for Go, Nagini [5] for Python, Prusti [6] for Rust, and VerCors [7] to verify Java Programs.

During translation, there will usually be non-trivial gaps between the front-end program and the encoding into the Viper IVL. For example, Viper itself does not directly support concurrency. One can encode the proof obligations arising from the rules in CSL in the Viper IVL, but even so there will still be a gap between the Viper encoding and CSL. Due to such gaps in the encoding it is a non-trivial task to prove such an encoding correct. In the case of the aforementioned Viper front-ends it has never been formally proven that the translations into Viper performed by these front-ends are correct. It could therefore very well be that there are bugs in said translations leading to incorrect verification results. That means we have no formal guarantees on the correctness of the verification results of these front-end verifiers. There are some other Viper front-ends also supporting concurrency for which the encoding into Viper has been formally proven correct on paper [8, 9]. However, none of these correctness proofs have ever been mechanized in an interactive theorem prover.

In this thesis, we provide the first mechanically formalized Viper front-end in the Isabelle theorem prover [10]. Our front-end takes as input programs in a language ImpCon which we defined ourselves and supports concurrency via parallel branching. We formally prove in Isabelle that if the Viper program resulting from the encoding is correct, based on an existing formalization for a subset of the Viper language, then the input program satisfies its specification (which includes the absence of data races). Finally, we create an executable of our front-end.

1.1 Outline

In Chapter 2 we introduce the syntax and semantics of the programming language ImpCon, which is the front-end language that we designed. In Chapter 3 we then define and discuss the translation function from ImpCon into the Viper IVL. We formally prove the soundness of that translation function in Chapter 4. In Chapter 5 we show some initial alternative approaches we took in defining ImpCon and in the soundness proof and discuss why they were sub-optimal and what we learned from them. In Chapter 6 we explain how we created a front-end executable with formal guarantees for ImpCon using our translation function. Finally, in Chapter 7 we conclude this thesis.

Chapter 2

The ImpCon Language

This chapter introduces the programming language ImpCon (where ”Imp” stands for imperative and ”Con” stands for concurrent) that we have defined and formalised in Isabelle.

In Section 2.1 we describe ImpCon’s syntax. After defining the ImpCon state in Section 2.2, we show the semantics of ImpCon in Section 2.3. Finally we define what it means for an ImpCon program to be correct in Section 2.4.

2.1 The Syntax

When defining the syntax of the ImpCon language we had to decide what kinds of commands and functionalities should be supported and which ones could be considered out of scope for our project. One big factor in this decision was that we tried to keep the ImpCon language as simple as possible in order to avoid unnecessary complications when proving soundness of the translation in chapter 4. On the other hand, of course, it was also important to achieve a good level of expressivity. The notions we wanted to support at the very least were heap operations and concurrency. The definition of the syntax proceeds accordingly.

In order to define ImpCon commands we need to start with their most elementary building block, ImpCon expressions. Only after we have properly defined ImpCon expressions we can formally define ImpCon assertions which partly consist of ImpCon expressions. And after we have defined both ImpCon expressions and ImpCon assertions it becomes possible to define the syntax of ImpCon commands. Therefore, let us start with ImpCon expressions.

2.1.1 Expressions

ImpCon expressions can consist of integer or boolean literals, the Null-literal and local variables. In terms of operations on these objects we support the unary negation operator and the binary operations for equality (==), comparison via <, logical conjunction and disjunction, addition, subtraction and multiplication.

$$\begin{aligned} E ::= & \mathit{Int}\ \mathit{integer} \mid \mathit{Bool}\ \mathit{boolean} \mid \mathit{Null} \\ & \mid x \\ & \mid \neg\, E \\ & \mid E == E \mid E \wedge E \mid E \vee E \mid E < E \mid E + E \mid E - E \mid E * E \end{aligned}$$

This set of operations already offers enough functionality to build most common expressions not involving division or modulo operations, albeit it might be slightly more tedious to build them without a direct operator. E.g. $e_1 > e_2$ can be expressed as $\neg(e_1 < e_2 \vee e_1 == e_2)$ using the expression syntax above. Hence adding more expressions would have resulted in more work when proving our translation correct without adding much more functionality.

One decision to keep it simple caused us to forgo heap-dependent expressions. Hence, while ImpCon does support heap operations as we will see in Section 2.1.3, it is with the above expression syntax impossible to make any expressions involving the heap. Our expressions are *heap-independent*.

Remark 2.1 *While we defined the syntax of expressions to to use the prefixes Int and Bool for integer and boolean literals, we will throughout the thesis leave that prefix out if it can be clearly derived from context.*

2.1.2 Assertions

As we want to not only be able to write programs using ImpCon but also reason about their correctness we need to also be able to specify properties on the program state, which we do via assertions.

$$\begin{aligned} A ::= & E \\ & \mid x.f \mapsto E \\ & \mid x.f \mapsto \_ \\ & \mid E \Rightarrow A \\ & \mid A \ * \ A \end{aligned}$$

ImpCon expressions that evaluate to True or False can be ImpCon assertions. ImpCon assertions also include points-to assertions where $x.f \mapsto e$ means that the field f of local variable x contains the value obtained from evaluating

expression e while $x.f \mapsto \_$ means that the field f of local variable x exists. points-to assertions also represent ownership in the sense that if they hold in the current thread of a concurrent execution or the process currently performing the execution of the program, then that thread or process owns the associated heap locations. Hence we also do not support fractional permissions to heap locations. Heap locations are always either fully owned or not at all.

Note that due to heap-independence of expressions points-to assertions of the form $x.f \mapsto e$ cannot relate the value of a field to another field. This will not hurt us in terms of expressiveness though, as we could simply create a local dummy variable to which we assign the value of the field we wish to mention in e.

Furthermore, ImpCon assertions include implications, and separating conjunctions via the binary operator $*$. We only support implications whose premises are expressions. The reason we do not support implications whose premises are other types of assertions and also the reason why we do not support usual logical conjunction of assertions alongside separating conjunctions is that these are not supported in Viper either. That is because they present fundamental challenges for automation.

2.1.3 Commands

The commands ImpCon supports are the following.

$$
\begin{aligned}
C ::= \; & SKIP \\
\mid \; & Assert\ A \\
\mid \; & x := E \\
\mid \; & x.f := E \\
\mid \; & x := y.f \\
\mid \; & x := alloc(f_1, f_2, \ldots, f_n) \\
\mid \; & C \ ; \ C \\
\mid \; & If\ E\ then\ C\ else\ C \\
\mid \; & \{A\}\ C\ \{A\} \parallel \{A\}\ C\ \{A\} \\
\mid \; & C \parallel C
\end{aligned}
$$

Let us go through the meanings of these commands. The command $SKIP$ denotes the empty command, i.e. a code block without any instructions. Assert-commands simply query whether a given assertion holds in the current program state.
The commands $x := e$, $x.f := e$, $x := y.f$ and $x := alloc(f_1, f_2, \ldots, f_n)$ are all different kinds of assignments. $x := e$ is a local assignment of the value of an expression e to a local variable x.

The next command, $x.f := e$, performs an assignment of the value of an expression e to the field f of a local reference variable x.
The third kind of assignment we have is the command $x := y.f$, where x and y are local variables with the value of y being a reference and f is a field name. This command assigns the value of the field access $y.f$ to x. The reason we included a command to specifically assign heap values to local variables lies in the fact that ImpCon expressions are heap-independent. Hence local variable assignment from expressions is insufficient to express what this command does. Moreover, without this command we cannot assign values from fields to other fields, again due to heap-independence of expressions. Hence, to express the assignment $x.f := y.f$, which ImpCon does not support, we can use another local variable $temp$ and write $temp := y.f; x.f := temp$.
The last type of assignment, $x := alloc(f_1, f_2, \ldots, f_n)$, exists to allocate fresh memory on the heap for a local reference variable x and its fields f_1 to f_n. The reason we also call it an assignment is that it assigns a new reference value pointing to an unused memory address when executed.
The next two commands are ones one is usually familiar with from other programming languages; Sequential composition of commands and If-statements. For the sake of simplicity we decided not to include While-loops, although adding support for them should not pose any fundamental challenges.
The most important command we included is the parallel command. The command $\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\}$ executes the ImpCon commands c_1 and c_2 in parallel. Through the parallel command we achieve our objective of supporting concurrency.
Note that, next to the two commands to be executed in parallel, the parallel command also takes four ImpCon assertions as arguments. These assertions are expected to be the pre- and postconditions of the respective commands. While these assertions do not have any impact on the semantics of ImpCon they will be needed for verification.
The final command listed in grey is the annotation-free parallel command and is only formally defined as a command. This command should never be part of an ImpCon program and exists solely for the purpose of defining the semantics of the parallel command as we will see in Section 2.3. The annotation-free parallel command is only supposed to signify the current state of execution while executing the parallel command step by step. We call an ImpCon program *user-writable* if it does not contain any annotation-free parallel commands.

2.1.4 The Type Context

What we have not mentioned yet is how local variables are declared. Note that there is no ImpCon command to that effect. Instead of declaring variables within the command we decided to have the user declare all local variables in the beginning before the actual top-level command starts. The information

on the declared local variables together with their types is then collected into what we call an *ImpCon type context*.

Remark 2.2 *To denote a* partial map *f from its domain X to another set Y we use the notation*

$$f : X \rightharpoonup Y.$$

Definition 2.3 *An* ImpCon type context *T is a partial map*

$$T : \mathit{variable\ name} \rightharpoonup \{\mathit{Int}, \mathit{Bool}, \mathit{Ref}\},$$

which maps each declared local variable to its declared type.

2.2 The ImpCon State Model

Our ImpCon state consists of a store, which maps local variables to values, and a heap which maps pairs of addresses and field identifiers to values.

Definition 2.4 *An* ImpCon state *is a pair (s, h) consisting of a* store *s and a* heap *h which are defined as partial functions*

$$s : \mathit{variable\ name} \rightharpoonup \mathit{values}_I,$$
$$h : (\mathit{address}, \mathit{field}) \rightharpoonup \mathit{values}_I.$$

We also refer to pairs $(a, f) \in \mathit{address} \times \mathit{field}$ as heap locations.

ImpCon values can be integers, booleans or references, where references can either be Null or an address a which stands for a memory location. Thus values_I is defined as follows.

$$\mathit{values}_I ::= \mathit{Int\ integer} \mid \mathit{Bool\ boolean} \mid \mathit{Null} \mid \mathit{Ref\ address}$$

Having defined ImpCon states, we would also like to define addition of states at this point, as it will play a role in the semantics of our assertion language.

Definition 2.5 (Addition of ImpCon states) *For two ImpCon states φ_1 and φ_2, we say that φ_1 and φ_2 are* joinable *if they have the same store and the domains of their heaps are disjoint.*

If states (s_1, h_1) and (s_2, h_2) are joinable, then

$$(s_1, h_1) \bigoplus\nolimits_I (s_2, h_2) \triangleq (s_1, h_1 \uplus h_2)$$

where we define $h_1 \uplus h_2$ to be the heap obtained by joining the heaps h_1 and h_2. Otherwise we say that $(s_1, h_1) \bigoplus_I (s_2, h_2)$ is undefined.

Furthermore, we say that ImpCon state φ is greater than or equal to ImpCon state φ' if there exists an ImpCon state φ'' such that

$$\varphi' \bigoplus\nolimits_I \varphi'' = \varphi.$$

2.3 The Semantics

Now that we have defined the syntax of ImpCon and ImpCon states we can now move on to the semantics, which formally tell us the effect on the state of each command. Again we start on the level of expressions and then go on to assertions and commands.

2.3.1 Expressions

On the level of expressions we define what it means for an expression to evaluate to some value.

Definition 2.6 *Given an ImpCon expression e, an ImpCon state φ and $v \in values_I$, we say that e evaluates to value v in state φ if and only if*

$$\langle e,\ \varphi \rangle \Downarrow v.$$

The semantics of expression evaluation are given in Figure 2.1.

As we can see in Figure 2.1 expression evaluation is quite intuitive. Local variables evaluate to the value they are mapped to in the store of the current state and all logical and arithmetic operations operate on the expected types and lead to the expected result regarding the usual interpretation of these operators. Furthermore, in the case of logical conjunction and disjunction we have also added the rules (OrLazy) and (AndLazy) which directly evaluate a logical conjunction and disjunction if the value is already clear from the left-hand-side of the operator. Also, in the case of equality, it is worth noting that we decided to allow comparison of any two types of values. If the types of the values of the operands differ the result will be *False*.

2.3.2 Assertions

Using expression evaluation we can proceed towards the semantics of our assertion language. That is, we define formally what is needed for an ImpCon assertion to hold.

Definition 2.7 *Given an ImpCon assertion A and an ImpCon state φ a, we say that A holds in state φ if and only if*

$$\varphi \vDash A.$$

The inference rules of $\vDash$ are given in Figure 2.2.

The inference rules in the cases in which the assertion is an expression are again quite intuitive. For points-to assertions of the form $x.f \mapsto e$ we require two things. Firstly, the heap location indicated by $x.f$ has to to exist, that is x has to be a local variable denoting a reference which contains an address

$$\frac{}{\langle Int\ i,\ \varphi\rangle \Downarrow (Int\ i)}\ (Int) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Bool\ True)}{\langle e_1 \vee e_2,\ \varphi\rangle \Downarrow (Bool\ True)}\ (OrLazy)$$

$$\frac{}{\langle Bool\ b,\ \varphi\rangle \Downarrow (Bool\ b)}\ (Bool) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Bool\ b_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Bool\ b_2)}{\langle e_1 \wedge e_2,\ \varphi\rangle \Downarrow (Bool\ (b_1 \wedge b_2))}\ (And)$$

$$\frac{}{\langle Null,\ \varphi\rangle \Downarrow Null}\ (Null) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Bool\ False)}{\langle e_1 \wedge e_2,\ \varphi\rangle \Downarrow (Bool\ False)}\ (AndLazy)$$

$$\frac{}{\langle x,\ (s,h)\rangle \Downarrow s(x)}\ (Var) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Int\ i_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Int\ i_2)}{\langle e_1 < e_2,\ \varphi\rangle \Downarrow (Bool\ (i_1 < i_2))}\ (Lt)$$

$$\frac{\langle e,\ \varphi\rangle \Downarrow (Bool\ b)}{\langle \neg\ e,\ \varphi\rangle \Downarrow (Bool\ \neg b)}\ (Not) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Int\ i_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Int\ i_2)}{\langle e_1 + e_2,\ \varphi\rangle \Downarrow (Int\ (i_1 + i_2))}\ (Add)$$

$$\frac{\langle e_1,\ \varphi\rangle \Downarrow v_1 \quad \langle e_2,\ \varphi\rangle \Downarrow v_2}{\langle e_1 == e_2,\ \varphi\rangle \Downarrow (Bool\ (v_1 == v_2))}\ (Eq) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Int\ i_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Int\ i_2)}{\langle e_1 - e_2,\ \varphi\rangle \Downarrow (Int\ (i_1 - i_2))}\ (Sub)$$

$$\frac{\langle e_1,\ \varphi\rangle \Downarrow (Bool\ b_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Bool\ b_2)}{\langle e_1 \vee e_2,\ \varphi\rangle \Downarrow (Bool\ (b_1 \vee b_2))}\ (Or) \qquad \frac{\langle e_1,\ \varphi\rangle \Downarrow (Int\ i_1) \quad \langle e_2,\ \varphi\rangle \Downarrow (Int\ i_2)}{\langle e_1 * e_2,\ \varphi\rangle \Downarrow (Int\ (i_1 * i_2))}\ (Mul)$$

Figure 2.1: Expression evaluation for ImpCon expressions.

$$\frac{\langle e,\ \varphi\rangle \Downarrow (Bool\ True)}{\varphi \vDash e}\ (Exp) \qquad \frac{\langle e,\ \varphi\rangle \Downarrow (Bool\ True) \implies \varphi \vDash A}{\varphi \vDash e \implies A}\ (Imp)$$

$$\frac{\langle e,\ (s,h)\rangle \Downarrow v \quad s(x) = Ref\ a \quad h(a,f) = v}{(s,h) \vDash x.f \mapsto e}\ (PointsTo) \qquad \frac{\varphi_1 \vDash A_1 \quad \varphi_2 \vDash A_2 \quad \varphi_1 \bigoplus_I \varphi_2 = \varphi}{\varphi \vDash A_1\ *\ A_2}\ (SepCon)$$

$$\frac{s(x) = Ref\ a \quad (a,f) \in dom(h)}{(s,h) \vDash x.f \mapsto \_}\ (PointsToSth)$$

Figure 2.2: Satisfaction of ImpCon assertions.

and the heap location corresponding to that address and the chosen field f has to be in the domain of the heap. Secondly, we require the heap to map that heap location to the value of e. For points-to assertions of the form $x.f \mapsto \_$ we only require the heap location indicated by $x.f$ to exist. Lastly, if the assertion is a separating conjunction of two ImpCon assertions A_1 and A_2, we need both assertions to hold using disjoint parts of the heap.

2.3.3 Commands

Having defined what it means for an ImpCon assertion to hold and how to evaluate expressions we now possess all tools to define the semantics of the ImpCon programming language. We use a small-step semantics rather than a big-step semantics, as there is no easy way of defining a big-step semantics for concurrent programs. That is because for concurrently executed code we need to consider each possible interleaving of steps executed in concurrent sections and that leads to a large amount of possible final states upon finishing execution.

Definition 2.8 *Given ImpCon commands c and c', ImpCon states φ and φ', and an ImpCon type context T we say that configuration (c, φ) results in configuration (c', φ') when executing a single step of c in state φ with respect to T if and only if*

$$(c,\ \varphi) \rightarrow_T (c',\ \varphi').$$

The inference rules of these small-step semantics are given in Figure 2.3.

There are some things worth noting about these small-step semantics.
Firstly, assignments to and from fields can only be executed if the heap location indicated by the field access exists, i.e. it lies within the domain of the heap.
Secondly, we only perform local variable assignments if the state resulting from this assignment will not cause a conflict with the type context. This can be seen in in the rules (Assign), (AssignFromField) and (Alloc).

The (Alloc)-rule we discuss in more detail. The alloc command non-deterministically chooses and assigns a new, fresh address to the reference variable we wish to allocate. By fresh address we mean here that this address has not been used yet, meaning that no memory on the heap associated with that address has been allocated so far. Freshness of an address a we ensure by checking that there exists no field f such that the heap location (a, f) exists on the heap. While assigning that fresh address to the reference variable we wish to allocate we also set the values of the heap locations corresponding to that address and the fields we wish to allocate to $Null$. Hence these heap locations are then part of the domain of the heap. In ImpCon it does not matter that the type of the value we initially assign to those fields is Ref, as ImpCon does not fix the types of fields. Note that the command for

$$\frac{\varphi \vDash A}{(Assert\ A,\ \varphi) \rightarrow_T (SKIP,\ \varphi)}\ (Assert) \qquad \frac{}{(SKIP\ ;\ c_2,\ \varphi) \rightarrow_T (c_2,\ \varphi)}\ (SeqS)$$

$$\frac{\langle e, (s,h)\rangle \Downarrow v \quad T(x) = type(v)}{(x := e,\ (s,h)) \rightarrow_T (SKIP,\ (s[x := v],h))}\ (Assign) \qquad \frac{(c_1,\ \varphi) \rightarrow_T (c_1',\ \varphi')}{(c_1\ ;\ c_2,\ \varphi) \rightarrow_T (c_1'\ ;\ c_2,\ \varphi')}\ (Seq1)$$

$$\frac{\langle e, (s,h)\rangle \Downarrow v \quad s(x) = Ref\ a \quad (a,f) \in dom(h)}{(x.f := e,\ (s,h)) \rightarrow_T (SKIP,\ (s,h[(a,f) := v]))}\ (AssignToField) \qquad \frac{\langle e,\ \varphi\rangle \Downarrow (Bool\ True)}{(If\ e\ then\ c_1\ else\ c_2,\ \varphi) \rightarrow_T (c_1,\ \varphi)}\ (If1)$$

$$\frac{s(y) = Ref\ a \quad h(a,f) = v \quad T(x) = type(v)}{(x := y.f,\ (s,h)) \rightarrow_T (SKIP,\ (s[x := v],h))}\ (AssignFromField) \qquad \frac{\langle e,\ \varphi\rangle \Downarrow (Bool\ False)}{(If\ e\ then\ c_1\ else\ c_2,\ \varphi) \rightarrow_T (c_2,\ \varphi)}\ (If2)$$

$$\frac{\begin{array}{c}\forall f, (a,f) \notin dom(h) \\ T(x) = Ref \\ f_1, f_2, \ldots, f_n\ pairwise\ distinct\end{array}}{(x := alloc(f_1, f_2, \ldots, f_n),\ (s,h)) \rightarrow_T (SKIP,\ (s[x := Ref\ a], h[(a,f_1) := Null, \ldots, (a,f_n) := Null]))}\ (Alloc)$$

$$\frac{}{(\{P_1\}\ c_1\ \{Q_1\} \parallel \{P_2\}\ c_2\ \{Q_2\},\ \varphi) \rightarrow_T (c_1 \parallel c_2,\ \varphi)}\ (Par)$$

$$\frac{(c_1,\ \varphi) \rightarrow_T (c_1',\ \varphi')}{(c_1 \parallel c_2,\ \varphi) \rightarrow_T (c_1' \parallel c_2,\ \varphi')}\ (EPar1) \qquad \frac{(c_2,\ \varphi) \rightarrow_T (c_2',\ \varphi')}{(c_1 \parallel c_2,\ \varphi) \rightarrow_T (c_1 \parallel c_2',\ \varphi')}\ (EPar2)$$

$$\frac{}{(SKIP \parallel SKIP,\ \varphi) \rightarrow_T (SKIP,\ \varphi)}\ (EParS)$$

Figure 2.3: Small-step semantics for ImpCon commands.

assignment to fields does not check the type of the former value of the field we assign to.

Other rules we wish to discuss in more detail are the semantics for parallel and annotation-free parallel commands. Starting from a configuration consisting of a parallel command and some ImpCon state the only applicable inference rule is the (Par)-rule. The (Par)-rule ignores all assertions that are part of the parallel command and lets us transition directly from the parallel command to the corresponding annotation-free parallel command. Hence the assertions that are part of the parallel command do not play any role in the ImpCon semantics. They only become important when we translate the parallel command to Viper in Section 3.1.
After executing the step that results in the configuration containing the annotation-free parallel command we actually start executing the parallel sections using the rules (EParS), (EPar1) and (EPar2). In order to simulate concurrent execution of both parallel sections, at each step it will be decided

non-deterministically whether to execute the next step on the left parallel section by applying rule (EPar1) or the right parallel section by applying rule (EPar2). To be able to make that non-deterministic choice *at each step* was the reason to use small-step semantics rather than big-step semantics. Only if both parallel sections have finished executing, i.e. we reached a configuration involving an annotation-free parallel command in which both parallel sections are empty ($SKIP$), then we finish the execution of the annotation-free parallel command trough rule (EParS).

The rules in figure 2.3 only formalize a single execution step. To reason about multiple steps of execution, we introduce the following definition.

Definition 2.9 *Given ImpCon commands c and c′, ImpCon states φ and φ′, and an ImpCon type context T we say that configuration* (c', φ') *can be* reached *from configuration* (c, φ) *with respect to T if and only if*

$$(c,\ \varphi) \rightarrow_T^* (c',\ \varphi'),$$

which expresses that we can end up in configuration (c', φ') *starting from configuration* (c, φ) *with respect to T via zero or a finite number of transitions via* $\rightarrow$.

2.4 Correctness of ImpCon Programs

Now that we have defined the syntax and semantics of ImpCon, let us return to the bigger picture of ImpCon programs.

Definition 2.10 *An* ImpCon program *is a pair* (T, c) *of an ImpCon type context T and an ImpCon command c.*

Considering our goal is to prove soundness of the translation of ImpCon into Viper and this involves proving correctness of ImpCon programs it is not sufficient to only know how to write and execute ImpCon programs. We also need to formally define what it means for an ImpCon program to be correct. For that purpose we also have to define what it means for the execution of an ImpCon command to abort.

Definition 2.11 *We say that an ImpCon command c aborts in the next step from ImpCon state φ with respect to a type context T if and only if*

$$(c,\ \varphi) \rightarrow_T ABORT.$$

The semantics involving ABORT are given in Figure 2.4.

We consider the execution of a command to abort when either an assertion does not hold or we are trying to read or write to unallocated heap memory. That is, either there is no heap location associated with the field access due to

$$\frac{\varphi \not\models A}{(Assert\ A,\ \varphi) \to_T ABORT}\ (AssertA) \qquad \frac{\begin{array}{c} s(y) = Ref\ a \\ (a,f) \notin dom(h) \end{array}}{(x := y.f,\ (s,h)) \to_T ABORT}\ (AFFA2)$$

$$\frac{\nexists\, a, s(x) = Ref\ a}{(x.f := e,\ (s,h)) \to_T ABORT}\ (ATFA1) \qquad \frac{(c_1,\ \varphi) \to_T ABORT}{(c_1\ ;\ c_2,\ \varphi) \to_T ABORT}\ (SeqA)$$

$$\frac{\begin{array}{c} s(x) = Ref\ a \\ (a,f) \notin dom(h) \end{array}}{(x.f := e,\ (s,h)) \to_T ABORT}\ (ATFA2) \qquad \frac{(c_1,\ \varphi) \to_T ABORT}{(c_1 \parallel c_2,\ \varphi) \to_T ABORT}\ (EParA1)$$

$$\frac{\nexists\, a, s(y) = Ref\ a}{(x := y.f,\ (s,h)) \to_T ABORT}\ (AFFA1) \qquad \frac{(c_2,\ \varphi) \to_T ABORT}{(c_1 \parallel c_2,\ \varphi) \to_T ABORT}\ (EParA2)$$

Figure 2.4: Abortion criteria for ImpCon commands.

the absence of an address or the heap location does not exist. Note that we do not abort when trying to assign a value of the wrong type to a local variable. In these situations the execution only gets "stuck" in the sense that there is no inference rule that can be applied in such a configuration. The reason is that in ImpCon such typing errors are independent from the concept of aborting. The absence of typing errors can be checked independently so that we can restrict ourselves to ImpCon programs in which we never try to assign values of the wrong type to local variables, that is, we *uphold* the type context.

Definition 2.12 *We say that an ImpCon state $\varphi = (s,h)$* upholds *a type context T if for all variable names x,*

- *if $x \notin dom(T)$ then $x \notin dom(s)$, and*
- *if $x \in dom(T)$ then $type(s(x)) = T(x)$.*

Using the concept of aborting we can now define what it means for an ImpCon program to be correct given some pre- and postcondition.

Definition 2.13 *Let P, Q be ImpCon assertions. We say that an ImpCon program (T,c) satisfies a Hoare triple with precondition P and postcondition Q, or short*

$$T \models \{P\}\, c\, \{Q\},$$

if and only if for all ImpCon states φ upholding type context T such that $\varphi \models P$ we have that for all c', φ' such that $(c,\varphi) \to_T^* (c',\varphi')$, it is true that

- *$(c',\ \varphi') \to_T ABORT$ does not hold, and*
- *$c' = SKIP \implies \varphi' \models Q$*

To state this definition in other words, we have that $T \vDash \{P\}\, c\, \{Q\}$ if for all ImpCon states φ that uphold T and satisfy precondition P, we never abort when starting from configuration (c, φ) and if we finish the execution of c we end up in a state that satisfies the postcondition Q.

Chapter 3

Translating ImpCon into Viper

The translation function from ImpCon into Viper is the main building block connecting ImpCon and Viper. The main goal of this project was to define a translation that is completely sound and prove this fact. Hence, in this chapter we will present, and provide an intuition for, our translation function, which we will then prove sound in chapter 4.

Just like ImpCon we have implemented the translation function in Isabelle. We are doing so by connecting to an existing Isabelle implementation of Viper.

3.1 The Translation Function

Similarly to ImpCon programs we let Viper programs consist of a Viper statement and a Viper type context. Therefore in this section we define translation functions from ImpCon commands to Viper statements and ImpCon type contexts into Viper type contexts.

When translating ImpCon programs into the Isabelle Implementation of Viper, we have to consider that ImpCon commands are incrementally built from assertions and expressions. Consequently, we also have to define translation functions for expressions and assertions before translating ImpCon commands.

3.1.1 Expressions

The translation function for expressions is actually quite trivial, due to the fact that ImpCon expressions form a subset of the set of Viper expressions. For each ImpCon expression, the same expression also exists in the Viper formalisation and is also evaluated in the exact same way. For that reason the translation function for expressions is essentially an identity function other than having to translate an expression from the ImpCon Abstract Syntax Tree

$$
\begin{aligned}
&[\![e]\!] && := [\![e]\!] \\
&[\![x.f \mapsto e]\!] && := acc(x.f)\ \&\&\ x.f == [\![e]\!] \\
&[\![x.f \mapsto \_]\!] && := acc(x.f) \\
&[\![e \Rightarrow A]\!] && := [\![e]\!] \Rightarrow [\![A]\!] \\
&[\![A_1 * A_2]\!] && := [\![A_1]\!]\ \&\&\ [\![A_2]\!]
\end{aligned}
$$

Figure 3.1: Translation of assertions.

(AST) to the Viper AST. Thus, for the purposes of this thesis the translation function for expressions is simply

$$[\![e]\!] := e,$$

although in this thesis we will not simplify the use of the translation function away in order to distinguish between when we are viewing an expression as an ImpCon expression or as a Viper expression.

3.1.2 Assertions

For ImpCon there does not always exist a close to syntactically identical equivalent assertion in the Viper formalisation, unlike for ImpCon expressions. That is why we list the recursive definition of our translation function in full in Figure 3.1.

In the case where our assertion is simply a boolean expression or our assertion is an implication, there is semantically equivalent Viper assertion syntax which uses the same notation.

In Viper's assertion language points-to assertions as we have defined for ImpCon do not exist. Instead Viper contains the *accessibility predicate* $acc(x.f)$, which expresses exclusive ownership of heap location $x.f$ for the current thread or process and therefore has the same interpretation as $x.f \mapsto \_$ does in ImpCon. Thereby, as, unlike ImpCon, Viper does support heap-dependent expressions, we can also express points-to assertions of the form $x.f \mapsto e$ as $acc(x.f)$ && $x.f == [\![e]\!]$, where && denotes the separating conjunction operator in Viper.

Separating conjunctions in ImpCon can simply be translated into separating conjunctions in Viper. However, separating conjunctions in Viper do not quite have the same semantics as separating conjunctions in ImpCon. That is because Viper supports fractional permissions [11]. That will, however, not cause any problems for our translation. That is because the subset of Viper assertions that ImpCon assertions translate to does not include any assertions with non-integral amounts of ownership. Thus, the separating conjunctions

$$
\begin{aligned}
&[\![SKIP]\!]^{Main} && := SKIP \\
&[\![Assert\ A]\!]^{Main} && := Assert\ [\![A]\!] \\
&[\![x := e]\!]^{Main} && := x := [\![e]\!] \\
&[\![x.f := e]\!]^{Main} && := x.f := [\![e]\!] \\
&[\![x := y.f]\!]^{Main} && := x := y.f \\
&[\![x := alloc(f_1, f_2, \ldots, f_n)]\!]^{Main} && := Havoc\ x\ ;\ Inhale\ acc(x.f_1)\ \&\& \\
& && \quad acc(x.f_2)\ \&\&\ \ldots\ \&\&\ acc(x.f_n) \\
&[\![c_1\ ;\ c_2]\!]^{Main} && := [\![c_1]\!]^{Main}\ ;\ [\![c_2]\!]^{Main} \\
&[\![If\ e\ then\ c_1\ else\ c_2]\!]^{Main} && := If\ [\![e]\!]\ then\ [\![c_1]\!]^{Main}\ else\ [\![c_2]\!]^{Main} \\
&[\![\{Pre_1\}\ c_1\ \{Post_1\}\ ||\ \{Pre_2\}\ c_2\ \{Post_2\}]\!]^{Main} && := Exhale\ [\![Pre_1]\!]\ ;\ Exhale\ [\![Pre_2]\!]\ ; \\
& && \quad Inhale\ [\![Post_1]\!]\ ;\ Inhale\ [\![Post_2]\!]
\end{aligned}
$$

Figure 3.2: Main proof obligation resulting from translation of commands.

in ImpCon and Viper behave the same way for ImpCon assertions and their translated counterparts. We will show this in more detail in Section 4.2.

3.1.3 Commands

Using the translation functions for expressions and assertions we can define our translation function from ImpCon commands to Viper statements.

The translation function for commands consists of two parts. The first part is a function $[\![\cdot]\!]^{Main}$ which maps an ImpCon command to its main proof obligation. By proof obligation we mean here a Viper statement that has to be verified with respect to the same type context as the given ImpCon command. The second part is a function $[\![\cdot]\!]_T^{EPO}$ which maps an ImpCon command to the set of extra proof obligations (also expressed as Viper statements) that have to verify alongside the main proof obligation. The definitions of these functions are given in Figures 3.2 and 3.3. For most ImpCon commands it suffices to take a Viper statement with very similar semantics and define that statement to be the main proof obligation. In the case of the parallel command, however, we have to define some extra proof obligations. The reason we will motivate later.

Apart from parallel commands the only ImpCon command for which there is no direct equivalent is allocation of reference variables[1]. This command

[1]The full Viper subset does support allocation of references via the ”new” statement but this statement does not exist in the Isabelle formalisation.

$$
\begin{aligned}
&[\![SKIP]\!]_T^{EPO} && := \{\} \\
&[\![Assert\ A]\!]_T^{EPO} && := \{\} \\
&[\![x := e]\!]_T^{EPO} && := \{\} \\
&[\![x.f := e]\!]_T^{EPO} && := \{\} \\
&[\![x := y.f]\!]_T^{EPO} && := \{\} \\
&[\![x := alloc(f_1, f_2, \ldots, f_n)]\!]_T^{EPO} && := \{\} \\
&[\![c_1\ ;\ c_2]\!]_T^{EPO} && := [\![c_1]\!]_T^{EPO} \cup [\![c_2]\!]_T^{EPO} \\
&[\![If\ e\ then\ c_1\ else\ c_2]\!]_T^{EPO} && := [\![c_1]\!]_T^{EPO} \cup [\![c_2]\!]_T^{EPO} \\
&[\![\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\}]\!]_T^{EPO} && := [\![c_1]\!]_T^{EPO} \cup [\![c_2]\!]_T^{EPO} \cup \\
& && \quad \{Inhale\ [\![Pre_1]\!]\ ;\ [\![c_1]\!]^{Main}\ ;\ Exhale\ [\![Post_1]\!], \\
& && \quad\ Inhale\ [\![Pre_2]\!]\ ;\ [\![c_2]\!]^{Main}\ ;\ Exhale\ [\![Post_2]\!]\}.
\end{aligned}
$$

Figure 3.3: Extra proof obligations resulting from translation of commands.

and the parallel command do not exist at all in the Viper version we are connecting to.

Recall that in ImpCon the command $x := alloc(f_1, f_2, \ldots, f_n)$ links reference variable x to a nondeterministically chosen fresh address a for which no fields have been allocated on the heap, and allocates the fields $f_1, f_2, \ldots, f_n$ for a. In Viper we try to simulate that behaviour through a *havoc* statement followed by an *inhale* statement.
The havoc statement in Viper takes a local variable as argument and nondeterministically assigns a new value to the specified local variable which is of the same type as the local variable. As in the translation havoc statements are only introduced by alloc commands the havoced local variable will always be of type reference.
The inhale statement in Viper takes an assertion as argument and adds all permissions in that assertion to the current permissions and then assumes the constraints in the assertion to hold. Hence we simulate the allocation of the fields $f_1, f_2, \ldots, f_n$ for reference variable x by inhaling $(acc(x.f_1)$ && $acc(x.f_2)$ && $\ldots$ && $acc(x.f_n))$. That is, we are adding full permission to each of the heap locations indicated by $x.f_1, x.f_2, \ldots, x.f_n$.
Note that this translation is actually less restrictive than the alloc command, because the havoc statement nondeterministically chooses one of all possible addresses, while ImpCon only considers fresh address a for which no fields have been allocated on the heap. Thus, while this is a sound translation as we will see in Chapter 4 (intuitively, because the translation models at least all possible choices taken by the alloc), the aforementioned fact can lead to incompletenesses, where the Viper program cannot be verified while the

$$\frac{\begin{array}{c}\{P_1\}\, c_1\, \{Q_1\} \\ \{P_2\}\, c_2\, \{Q_2\} \\ fv(P_1, c_1, Q_1) \cap wr(c_2) = \varnothing \\ fv(P_2, c_2, Q_2) \cap wr(c_1) = \varnothing\end{array}}{\{P_1 \ * \ P_2\}\, c_1 \parallel c_2\, \{Q_1 \ * \ Q_2\}}\ (parallel\ rule) \qquad \frac{\begin{array}{c}\{P\}\, c\, \{Q\} \\ fv(F) \cap wr(c) = \varnothing\end{array}}{\{P \ * \ F\}\, c\, \{Q \ * \ F\}}\ (frame\ rule)$$

Figure 3.4: The parallel rule and frame rule.

ImpCon program is correct.

Moving on to parallel commands, the principal motivation in defining our translation the way we did are the parallel and frame rule from concurrent separation logic. These rules can be written as in Figure 3.4, where $fv(\cdot)$ denotes the set of free local variables in an assertion or command and $wr(\cdot)$ denotes the set of local variables written to in a command. The intuition behind wanting to use the parallel rule for parallel commands is that it allows one to check properties on each branch independently rather than having to consider both branches at once. The frame rule, on the other hand, intuitively says that we can frame information on parts of the state which are not mentioned or altered in the Hoare triple around it such that we can retain that information for the remainder of the commands to be executed. We want to use to frame to retain all information on heap locations not accessed in parallel sections that will be necessary for the remainder of the execution around the parallel command. As therefore, when checking the correctness of parallel commands, we want both to apply the parallel rule and frame around it, we use the following rule which we obtain from merging both rules.

$$\frac{\begin{array}{c}\{P_1\}\, c_1\, \{Q_1\} \\ \{P_2\}\, c_2\, \{Q_2\} \\ fv(P_1, c_1, Q_1) \cap wr(c_2) = \varnothing \\ fv(P_2, c_2, Q_2) \cap wr(c_1) = \varnothing \\ fv(F) \cap wr(c_1, c_2) = \varnothing\end{array}}{\{P_1 \ * \ P_2 \ * \ F\}\, c_1 \parallel c_2\, \{Q_1 \ * \ Q_2 \ * \ F\}}$$

To be able to apply this rule we hence need to specify pre- and postconditions for each of the two commands executed in parallel. Recall that we require the user to input pre- and postconditions for the two parallel sections as part of the parallel command even though they were unnecessary in terms of the ImpCon semantics. The reason that we ask the user to input them rather than inferring them is that inferring suitable instantiations of pre- and postconditions is not trivial to do in general.

Given an ImpCon command $\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\}$, the first two premises of our merged rule we require to hold by defining our extra proof obligations as the encoding of the translated versions of the Hoare

triples $\{Pre_1\}\, c_1\, \{Post_1\}$ and $\{Pre_2\}\, c_2\, \{Post_2\}$. That is, we encode the translated version of a Hoare triple $\{P\}\, c\, \{Q\}$ in Viper as

$$Inhale\ [\![P]\!]\ ;\ [\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!].$$

Recall that the inhale statement in Viper takes an assertion as argument and adds all permissions in that assertion to the current permissions and then assumes the constraints in the assertion to hold. The *exhale* statement in Viper takes an assertion as input and first asserts whether the given assertion holds and then, while that is irrelevant for this particular encoding, removes the permissions that are necessary to satisfy the assertion. Hence this encoding verifies from an empty state if and only if given that the precondition holds then the postcondition follows after execution of the statement $[\![c]\!]^{Main}$, as desired.

Disregarding the constraints on the free variables and the variables written to, the extra proof obligations essentially ensure the premises of the merged frame and parallel rule hold. Hence we can derive the rule's conclusion that

$$\{Pre_1 * Pre_2 * F\}\, c_1 \,||\, c_2\, \{Post_1 * Post_2 * F\}.$$

The main proof obligation we defined such that we can apply the above conclusion. If we can exhale both preconditions in sequence, then, as we subtract permissions with each exhale and translated assertions only assert integral permissions, the preconditions must operate on disjoint parts of the heap and so $(Pre_1 * Pre_2 * F)$ holds for a well-chosen frame. Then the motivation is that through the derived Hoare triple we can after execution of the parallel command assume both postconditions and the frame to hold. Thus, we inhale the postconditions so that all information derivable from them can be used to verify the commands that follow after the parallel command.

Let us get back to the premises about local variables in the merged rule. When proving soundness in Chapter 4 we will make the additional assumption that there will be no assignments to local variables within parallel sections. Our executable presented in Chapter 6 will also abort execution if that condition is not met. The assumption means that both of the parallel blocks can read from the store but neither can write to it. Hence the set of variables written to in both parallel blocks will be empty, and so the premises about local variables in the merged rule will be satisfied.

Moreover, without the assumption that there will be no assignments to local variables within parallel sections the main proof obligation obtained from the parallel command would actually be incorrect. Consider the ImpCon program in Listing 3. This program would clearly not be correct in ImpCon but the extra proof obligations and main proof obligation resulting from its

```
x : Int
x := 4;
Parallel {
  {True}   {True}
  x := 2 || x := 3
  {True}   {True}
};
Assert (x == 4);
```

Listing 3: Incorrect ImpCon program with verifying translation.

```
x : Ref
y : Ref
x := alloc(f);
y := alloc(f);
Parallel {
{x.f ↦ _} {y.f ↦ _}
  x.f := 2  y.f := 3
{x.f ↦ 2} {y.f ↦ 3}
}
Assert (x.f ↦ 2) * (y.f ↦ 3)
```

Listing 4: Example ImpCon command.

translation would verify in Viper. Without our additional assumption this would have to be resolved for example by havocing all local variables that are modified in parallel sections before inhaling the postconditions in the main proof obligation.

To give an example of our translation function in action consider again the example from Chapter 1 which we have converted into a proper ImpCon program in Listing 4. The translation of this ImpCon program can be seen in Listing 5. Here we encoded the extra proof obligations in the form of Viper methods *left* and *right* with pre- and postconditions and the translated Viper statement as body, for better readability. The main proof obligation is given in the main method.

3.1.4 The Type Context

Viper type contexts are defined just like ImpCon type contexts except that ImpCon's types are only a subset of IViper's types. Hence, to obtain the correct Viper type context when translating an ImpCon program it suffices to use the ImpCon type context as is. Since we will only ever talk about ImpCon type contexts and their translations, in the remainder of this thesis, for any ImpCon type context T we write T to denote the ImpCon type context itself as well as its translation into Viper.

```
1  method left(x : Ref)
2    requires acc(x.f)
3    ensures acc(x.f) && x.f == 2
4    {
5    x.f := 2
6    }
7
8  method right(y : Ref)
9    requires acc(y.f)
10   ensures acc(y.f) && y.f == 3
11   {
12   y.f := 3
13   }
14
15 method main(){
16   var x : Ref
17   var y : Ref
18
19   havoc x;
20   inhale acc(x.f);
21
22   havoc y;
23   inhale acc(y.f);
24
25   exhale acc(x.f);
26   exhale acc(y.f);
27   inhale acc(x.f) && x.f == 2;
28   inhale acc(y.f) && y.f == 3;
29
30   assert acc(x.f) && x.f == 2
31       && acc(y.f) && y.f == 3
32   }
```

Listing 5: Viper Translation of ImpCon command in Listing 5.

Chapter 4

Soundness of the Translation

Now that we have defined our translation function, which is the heart of our front-end, we are getting to the core of this thesis; the proof that this translation is **sound**. In this chapter we describe how we proved the soundness of the ImpCon front-end. By formalising the proof in Isabelle we have proven the ImpCon front-end sound once-and-forall.

In Section 4.1 we state the soundness theorem and our assumptions for the soundness proof. In Section 4.2 we introduce the notions of forward and backward translations between ImpCon and Viper states, which will prove very useful in our soundness proof. Finally, in Section 4.3, we prove the soundness theorem.

4.1 Expressing Soundness

In practice, we would like that if our front-end outputs a positive verification result then the input program is actually correct. Considering the structure

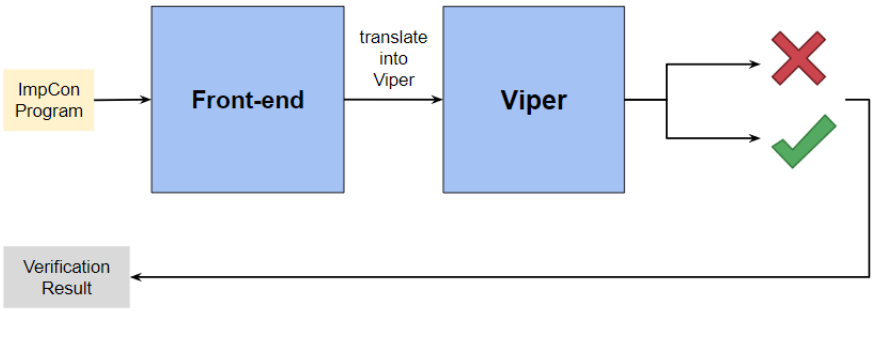


Figure 4.1: Overview of the Interaction between our Front-End and Viper.

of our front-end as seen in Figure 4.1, this means the following. If we take an ImpCon program c, let our Viper front-end translate it into a Viper statement s_V and then verify s_V using the Viper verifier, then Viper outputting a positive verification result implies that c must have been a correct ImpCon program. This can informally be expressed as

> For any ImpCon program Pr_I, if translate(Pr_I) is correct with respect to Viper's semantics, then Pr_I is correct with respect to ImpCon's semantics.

Recall that in Definition 2.13 we have defined what it means for an ImpCon program to be correct given some pre- and postconditions. Now we have to do the same for Viper.

Definition 4.1 *We write that a statement s_V verifies in Viper with respect to a Viper type context T and a Viper state ω if and only if s_V reduces to some set of states S when starting in state ω and with respect to type context T, according to IViper's semantics. We use the notation $Red_T(s_V,\ \omega) \Downarrow S$ to express that s_V reduces to the set of states S when starting in state ω with respect to type context T. Moreover, use the notation $Ver_T(s_V,\ \omega)$ to denote that it verifies, that is, there exists some set of states that s_V reduces to when starting in state ω with respect to type context T, or in short*

$$Ver_T(s_V,\ \omega) \triangleq (\exists S.\, Red_T(s_V,\ \omega) \Downarrow S)\,.$$

Just like we did when defining the encoding of our extra proof obligations in Section 3.1.3 we encode the translation of a Hoare triple $\{P\}\,c\,\{Q\}$ for ImpCon assertions P, Q and an ImpCon command c as the Viper statement

$$Inhale\ [\![P]\!]\ ;\ [\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!].$$

But in which Viper state should this Viper statement verify? To answer that question let us first properly define the Viper state.

Definition 4.2 *Let $values_V$ be the set of values in Viper. A* Viper permission heap *is a pair (m,h) consisting of a* permission mask *m and a* Viper heap *h which are defined as total and partial functions*

$$m : (address, field) \rightarrow [0,1],\ and$$
$$h : (address, field) \rightharpoonup values_V,$$

such that for all heap locations hl we have that

$$m(hl) > 0 \implies hl \in dom(h).$$

A Viper state *is then defined as pair (s, vs)*[1] *consisting of a* Viper store *s, where*

$$s : variable\ name \rightharpoonup values_V,$$

[1] Actually a Viper state is defined as a triple (s, t, vs) where t is a trace. We, however, never need to consider the trace throughout this thesis. Hence we leave it out when talking about Viper states.

and a permission heap vs.

Ideally we would like to verify an encoded Hoare triple starting from an empty state, i.e. a state in which no local variables have been declared and no memory has been allocated on the permission heap. That is, however, not possible in our case. Note that Viper statements resulting from translation of ImpCon commands do not contain local variable declarations, while they may contain operations on local variables. Furthermore, inhale statements in Viper do not declare local variables as a side effect, but can only allocate and add permission to the permission heap. Therefore, when verifying the encoding of the Hoare triple we wish to prove correct, we need to start from a Viper state in which all necessary local variables have already been declared. The necessary local variables and their types can be derived from the type context of the ImpCon program we want to prove correct. Hence the state we verify our encoding in will be any minimal state such that all local variables given in the ImpCon type context (and therefore also the corresponding Viper type context) are declared and of the correct type. The state will be minimal in the sense that no unnecessary store or heap locations lie in the domains of the store or permission heap. Such states we formally define as follows.

Definition 4.3 *A Viper state* $\omega = (s, (m, h))$ models *a type context* T *if*

- (m, h) *is the empty permission heap, i.e.* $dom(h) = \varnothing$ *and for all heap locations* hl *we have* $m(hl) = 0$*, and*
- *for all variable names* x*,*
  - *if* $x \notin dom(T)$ *then* $x \notin dom(s)$*, and*
  - *if* $x \in dom(T)$ *then* $type(s(x)) = T(x)$*.*

Thus, for an ImpCon program (T, c), precondition P and postcondition Q, we define the main proof obligation resulting from the translation of (T, c) to be correct with respect to P and Q if $Ver_T(Inhale\ [\![P]\!]\ ;\ [\![T]\!]^{Main}\ ;\ Exhale\ [\![Q]\!],\ \omega)$ for all states ω modelling type context T.

As presented in Section 3.1.3 the translation of an ImpCon program (T, c) does not only give a main proof obligation but also extra proof obligations in the case that c contains a parallel command. Therefore, to show correctness of the translation of (T, c), we also need to verify that the extra proof obligations emitted by the translation verify. This should also be done from a state modelling T for the same reasons.

We can now formally express the informal property stated at the start of this subsection.

Theorem 4.4 (Soundness) *Let* c *be an ImpCon command and* T *be an ImpCon type context, such that the pair* (T, c) *is a user-writable ImpCon program and* c *contains no assignments to local variables within parallel sections. Furthermore, let*

P and Q be ImpCon assertions. Assume that for all Viper states ω modelling T we have that

- $Ver_T(Inhale[\![P]\!] \,;\, [\![c]\!]^{Main} \,;\, Exhale[\![Q]\!],\, \omega)$, *and*
- *for all Viper statements* $stmt \in [\![c]\!]_T^{EPO}$ *we have that* $Ver_T(stmt,\, \omega)$.

Then we have that $T \vDash \{P\}\, c\, \{Q\}$.

Here we added one further restriction on the ImpCon command. The restriction is that the initial ImpCon command does not contain any assignments to local variables within parallel sections. That includes normal assignments, assignments to local variables from fields, and allocation. This assumption saved us quite a bit of work when using the parallel rule and frame rule from separation logic, as we will see later. Also, it is the reason we do not have to havoc any local variables in the main translation of a parallel command. That is because if there are no assignments to local variables of any kind in the parallel blocks the values of all local variables are guaranteed to stay the same throughout the parallel section.

It is also important to note that, albeit it greatly simplifies the proof, this assumption does not really affect the number of programs we can express. It may forbid the user to assign to a local variable in a parallel block, but it does not disallow the user from assigning to the heap. Therefore if the user wants to modify a local variable in a parallel block they could instead write the result to a dedicated heap location and after conclusion of the parallel block write the value of the dedicated heap location back to the local variable.

4.2 Forward and Backward State Translation

Before we look at the proof strategy there is one more concept we need to introduce. We have already described how to translate expressions, assertions, commands and type contexts, but not how to translate between ImpCon and Viper *states*. The idea of the proof we will describe in the next section is to relate between the ImpCon program and its Viper encoding by establishing a relationship between the states reached during their reduction. To do so, in this section we introduce translation functions from ImpCon states to Viper states and from Viper states to ImpCon states.

Recall from Definition 2.4 that an ImpCon state is defined as a pair (s_I, h_I) of an ImpCon store s_I and an ImpCon heap h_I and from Definition 4.2 that a Viper state is defined as a pair $(s_V, (m, h_V))$ of a Viper store s_V and a well-defined permission heap (m, h_V) consisting of a permission mask m and a Viper heap h_V.

In terms of structure the only difference between ImpCon states and Viper states is the existence of the permission mask. Hence, when translating

ImpCon states to Viper states we need to define the correct permission mask such that we can obtain a well-formed permission heap. One can view the ImpCon state as a binary permission model. If a heap location exists in an ImpCon state we have full permission to it, otherwise we have no permission. Hence, when translating an ImpCon state into Viper we simply set the permission mask to 1 for all existing heap locations and to 0 otherwise.

The only other difference between both state definitions lies in the sets of values the languages support. The set of values supported by Viper is the following.

$$\begin{aligned} values_V ::= & \mathit{Int}\ integer \\ & \mid \mathit{Bool}\ boolean \\ & \mid \mathit{Null} \\ & \mid \mathit{Ref}\ address \\ & \mid \mathit{Perm}\ permission \\ & \mid \mathit{Domain}\ domain \end{aligned}$$

Viper supports all values ImpCon does and additinally supports permissions, which are non-negative rationals, and domains, which are used to define one's own first-order theories in Viper [3]. Thus, the values ImpCon supports form a subset of the values Viper supports. Therefore, the ImpCon heap can also be interpreted as a Viper heap.

Hence we define the translation function from ImpCon states into Viper states, which we henceforth refer to as forward translation, as follows.

Definition 4.5 *The* forward translation *of an ImpCon state, $ftr(\cdot)$ is defined as*

$$\begin{aligned} ftr: State_I &\to State_V \\ (s,h) &\mapsto (s,(m,h)),\ \textit{where} \\ m(hl) &= \begin{cases} 1 & \textit{if } hl \in dom(h) \\ 0 & \textit{if } hl \notin dom(h) \end{cases} \end{aligned}$$

The permission heap of the forward translated state can be easily seen to be well-formed, as for all heap locations the permission mask maps to a value greater than 0 if and only if that heap location lies in the domain of the heap. Thereby forward translation gives a well-defined Viper state.

We, however, not only need to switch from ImpCon to Viper states in the proof, but also the other way around. That is because in our soundness proof we want to be able to derive the postcondition in the final ImpCon state after execution of an ImpCon command, from the fact that the exhale in the Viper encoding of the corresponding Hoare triple reduces in some particular Viper state. Hence we also need to define backward translation of states.

This is not as straightforward, as we now have to consider the situation that we want to translate a Viper state which contains permission- or domain values on the heap or store into an ImpCon state, which does not support these types of values. What should we map these store and heap locations to? Our solution is to simply delete these store and heap locations during backward translation. That is a sufficient solution for our purposes, because we actually never need to consider Viper states containing permission or domain values. In any Viper encoding we wish to verify we start from a state upholding an ImpCon type context, implying it only contains integers, booleans or references as values. From these states we only reduce Viper statements emitted by translations from ImpCon commands, and exhale and inhale statements operating on translated ImpCon assertions. None of these statements introduce domain or permission values in the state explicitly. Only inhale statements might implicitly introduce domain or permission values on the heap when gaining permission to an empty heap location, as on an intuitive level it non-deterministically chooses a value for that heap location if there is no constraint on its value. But in those cases we can safely ignore the executions where it non-deterministically chooses domain or permission values.
We also need to consider the Viper permission mask when backward translating. Of course, in the backward translated ImpCon state we have to map heap locations with full permission to existing values and heap locations with no permission we delete, as our ImpCon's binary permission model suggests. For all permissions between 0 and 1 exclusive we are left with a choice whether to round up to full permission, mapping to an existing heap value in the translation, or round down to 0 permission and delete the heap location. We chose to do the latter for all permissions between 0 and 1 exclusive. The reason is that in all translated assertions we only ever query whether we have full permission to a heap location by asserting access predicate $acc(x.f)$ for some field access $x.f$ corresponding to that heap location. That is due to the absence of fractional permissions in ImpCon. Hence deleting heap locations with less than full permission never changes the truth value of a translated assertion to False. Rounding up permissions on the other hand could change the truth value of a translated assertion from False to True.

Eventually this leads to the following definition of backward translation.

Definition 4.6 *The* backward translation *of a Viper state, $btr(\cdot)$ is defined as*

$$
\begin{aligned}
&btr\colon State_V \to State_I \\
&(s,(m,h)) \mapsto (s|_S, h|_H), \textit{ where} \\
&\qquad S = \{v \in varName \mid s(v) \in values_I\}, \textit{ and} \\
&\qquad H = \{hl \in Adresses \times Fields \mid h(hl) \in values_I \textit{ and } m(hl) = 1\}.
\end{aligned}
$$

One can show some interesting properties of forward- and backward translation and their relationship, some of which are the following.

Lemma 4.7 *Let $\varphi_1 \triangleq (s_1, h_1)$, $\varphi_2 \triangleq (s_2, h_2)$ be ImpCon states such that $dom(h_1) \cap dom(h_2) = \varnothing$. Then the domains of the heaps of $ftr(\varphi_1)$ and $ftr(\varphi_2)$ are disjoint as well.*

Lemma 4.8 *Let φ be an ImpCon state. Then $btr\,(ftr(\varphi)) = \varphi$.*

Some properties of forward- and backward translation are about addition of states. Due to the different structure of the states addition of Viper states is also defined slightly different. Recall that in order for $\varphi_1 \bigoplus_I \varphi_2$ to be defined for two ImpCon states φ_1 and φ_2, the states must have the same store and the domains of their heaps must be disjoint. For Viper states ω_1 and ω_2 on the other hand in order for $\omega_1 \bigoplus_V \omega_2$ to be defined, the domains of ω_1 and ω_2's permission heap need not be disjoint. Instead their heaps need to *agree*, that is, if a heap location lies in the domain of both heaps, it must map to the same value in both heaps. Moreover the sum of the permissions to that heap locations in both states must not add up to more than full permission. If these criteria are met we call their permission heaps *compatible*. Also just as for ImpCon states in order to be joinable both states need to have the same store.

We summarize this as part of the following definition.

Definition 4.9 *For two Viper states $\omega_1 \triangleq (s_1, (m_1, h_1))$ and $\omega_2 \triangleq (s_2, (m_2, h_2))$, we say that ω_1 and ω_2 are* joinable *if*

- *$s_1 = s_2$, and*
- *the permission heaps (m_1, h_1) and (m_2, h_2) are* compatible*, i.e.*
  - *for all heap locations hl, $m_1(hl) + m_2(hl) \leq 1$, and*
  - *for all heap locations hl, if $hl \in dom(h_1)$ and $hl \in dom(h_2)$, then $h_1(hl) = h_2(hl)$.*

If Viper states $(s_1, (m_1, h_1))$ and $(s_2, (m_2, h_2))$ are joinable, then

$$(s_1, (m_1, h_1)) \bigoplus_V (s_2, (m_2, h_2)) \triangleq (s_1, (m_1 + m_2, h_1 \uplus h_2))$$

where $m_1 + m_2$ is the permission mask obtained by adding the permissions in m_1 and m_2, and $h_1 \uplus h_2$ is the heap obtained by joining the heaps h_1 and h_2.
Otherwise we say that $(s_1, (m_1, h_1)) \bigoplus_V (s_2, (m_2, h_2))$ is undefined.

Furthermore, we say that Viper state ω is grater than or equal to ImpCon state ω' if there exists an ImpCon state ω'' such that

$$\omega' \bigoplus_V \omega'' = \omega.$$

We can then proceed to state some more results about forward- and backward translation.

Lemma 4.10 *Let ω be a Viper state that only contains values in $values_I$. Then ω is greater than or equal $ftr\,(btr(\omega)) = \varphi$.*

Lemma 4.11 *Let φ_1, φ_2 and φ be ImpCon states such that*

$$\varphi_1 \bigoplus_I \varphi_2 = \varphi.$$

Then

$$ftr(\varphi_1) \bigoplus_V ftr(\varphi_2) = ftr(\varphi).$$

Lemma 4.12 *Let ω_1, ω_2 and ω be Viper states such that*

$$\omega_1 \bigoplus_V \omega_2 = \omega.$$

Then $btr(\omega_1) +_I btr(\omega_2)$ is defined and $btr(\omega)$ is greater than or equal to $btr(\omega_1) +_I btr(\omega_2)$.

Forward- and backward translation also has some interesting properties in connection with evaluation of expressions and assertions which were very helpful during the soundness proof of the translation. We present and prove the most important ones in the remainder of this section.

We first relate the evaluation of expressions.

Proposition 4.13 *Let e be an ImpCon expression, φ an ImpCon state and $v \in values_I$. Then*

$$\langle e,\ \varphi\rangle \Downarrow v \iff \langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow v.$$

Proof Let e be an ImpCon expression, φ an ImpCon state and $v \in values_I$.

Recall that by our translation $[\![e]\!]$ is the same as e and we only write $[\![e]\!]$ to distinguish that we interpret it as a Viper expression. Also note that by definition of forward translation φ and $ftr(\varphi)$ have the same store. Therefore, as ImpCon expressions (and so also their translations) are heap-independent, we must have that $\langle e,\ \varphi\rangle \Downarrow v \iff \langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow v$. □

Moving on to assertions, however, we need to account for the non-existence of a direct equivalent of most ImpCon assertions in Viper due to the forward translation being injective but not surjective. Consequently, proving a similar result for satisfaction of assertions is more intricate. Hence we first prove only one direction.

Proposition 4.14 *Let A be an ImpCon assertion and φ an ImpCon state. Then*

$$\varphi \vDash A \implies ftr(\varphi) \vDash [\![A]\!].$$

Proof Let A be an ImpCon assertion and φ an ImpCon state. Assume that $\varphi \vDash A$. We prove the desired result by structural induction on A, starting with the base cases.

Base Case 1: $A = e$**.**
In this case

$$\begin{aligned} \varphi \vDash A &\iff \langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ True} && \text{(by definition)} \\ &\iff \langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow \mathit{Bool\ True} && \text{(by Proposition 4.13)} \\ &\iff ftr(\varphi) \vDash [\![A]\!] && \text{(by definition).} \end{aligned}$$

Base Case 2: $A = (x.f \mapsto e)$**.**
By the ImpCon assertion language we have that $\varphi \vDash A$ if and only if there exists an address a and a value $v \in \mathit{values}_I$ such that $\langle e,\ \varphi\rangle \Downarrow v$, the store of φ maps x to $\mathit{Ref}\ a$ and the permission heap of φ maps (a, f) to v. Then by Proposition 4.13 we have $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow v$ and by the definition of forward translation the store of $ftr(\varphi)$ maps x to $\mathit{Ref}\ a$ as well and the heap of $ftr(\varphi)$ maps (a, f) to v with full permission. Therefore $ftr(\varphi) \vDash x.f = [\![e]\!]$. Considering that $[\![A]\!] = Acc(x.f) \ * \ x.f = [\![e]\!]$, to see that $ftr(\varphi) \vDash [\![A]\!]$ we can split $ftr(\varphi)$ into two states ω_1 and ω_2 where ω_1 consists of the store of $ftr(\varphi)$ and the permission heap which maps (a, f) to v with full permission and contains no other heap locations in its domain. ω_2 consists of the store and permission heap of $ftr(\varphi)$ except that it maps (a, f) to v with zero permission. We can then see that $\omega_1 \bigoplus_V \omega_2 = ftr(\varphi)$, that $\omega_1 \vDash Acc(x.f)$ and that $\omega_2 \vDash x.f = [\![e]\!]$ as this expression does not require permissions and albeit having no permission to it ω_2's permission heap maps (a, f) to v. Thus $ftr(\varphi) \vDash [\![A]\!]$.

Base Case 3: $A = (x.f \mapsto \_)$**.**
By the ImpCon assertion language have that $\varphi \vDash A$ if and only if there exists an address a and a value $v \in \mathit{values}_I$ such that the store of φ maps x to $\mathit{Ref}\ a$ and the heap of φ maps (a, f) to v. Then by the definition of forward translation the store of $ftr(\varphi)$ maps x to $\mathit{Ref}\ a$ as well and the heap of $ftr(\varphi)$ maps (a, f) to v with full permission. Hence $ftr(\varphi) \vDash Acc(x.f)$, which by our translation function is equivalent to $ftr(\varphi) \vDash [\![A]\!]$.

Next we show the inductive cases.

Inductive Case 1: $A = \big(e \Rightarrow A_{Imp}\big)$**.**
In this case our inductive assumption is that

(IH) for all ImpCon states γ, $\gamma \vDash A_{Imp} \implies ftr(\gamma) \vDash [\![A_{Imp}]\!]$.

We have that

$$\varphi \vDash A \iff \big(\langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ True} \implies \varphi \vDash A_{Imp}\big) \quad \text{(by definition).}$$

Hence, if $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow \mathit{Bool\ True}$, then by Proposition 4.13 $\langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ True}$ and so $\varphi \vDash A_{Imp}$. By our inductive assumption then $ftr(\varphi) \vDash [\![A_{Imp}]\!]$. Thus $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow \mathit{Bool\ True} \implies ftr(\varphi) \vDash A_{Imp}$, which by definition is equivalent to $ftr(\varphi) \vDash [\![A]\!]$.

Inductive Case 2: $A = (A_1\ *\ A_2)$**.**

This time our inductive assumption is that for $i = 1, 2$,

(IH) for all ImpCon states γ, $\gamma \vDash A_i \implies ftr(\gamma) \vDash [\![A_i]\!]$.

We have that

$$\varphi \vDash A \iff \text{there exist ImpCon states } \varphi_1,\ \varphi_2 \text{ such that}$$
$$\varphi_1 \vDash A_1,\ \varphi_2 \vDash A_2 \text{ and } \varphi_1 \bigoplus_I \varphi_2 = \varphi.$$

Hence by our assumption that $\varphi \vDash A$ and (IH) we have that $ftr(\varphi_1) \vDash [\![A_1]\!]$ and $ftr(\varphi_2) \vDash [\![A_2]\!]$. Moreover, since $\varphi_1 \bigoplus_I \varphi_2 = \varphi$, by Lemma 4.11 we have that $ftr(\varphi_1) \bigoplus_V ftr(\varphi_2) = ftr(\varphi)$. Ultimately $ftr(\varphi) \vDash [\![A_1]\!]$ && $[\![A_2]\!]$ which is equivalent to $ftr(\varphi) \vDash [\![A]\!]$. □

That takes care of the case in which we are moving from ImpCon to Viper. For the other direction we prove a slightly more general statement from which we then derive the desired result.

Proposition 4.15 *Let A be an ImpCon assertion and ω a Viper state such that all values in the range of its store and heap lie in $values_I$. Then*

$$\omega \vDash [\![A]\!] \implies btr(\omega) \vDash A.$$

When proving this result and also some others in later sections we will need a very useful result called monotonicity of assertions, which holds in ImpCon as well as for translated assertions in Viper.

Lemma 4.16 (Monotonicity of Assertions - ImpCon) *Let φ_+ and φ be ImpCon states such that φ_+ is greater than or equal to φ and let A be an ImpCon assertion. Then*

$$\varphi \vDash A \implies \varphi_+ \vDash A$$

Lemma 4.17 (Monotonicity of Translated Assertions - Viper) *Let ω_+ and ω be Viper*
states such that ω_+ is greater than or equal to ω and let A be an ImpCon assertion. Then

$$\omega \vDash [\![A]\!] \implies \omega_+ \vDash [\![A]\!]$$

The intuition behind this result is that if we have enough information to show an assertion holds using only part of the heap resp. permission heap then that assertion also holds when we gain even more information by considering the full state.

We now prove Proposition 4.15.

Proof Let A be an ImpCon assertion and ω a Viper state such that all values in the range of its store and heap lie in $values_I$. Assume that $\omega \vDash [\![A]\!]$. Again we prove the desired result by structural induction on A, starting with the base cases.

Base Case 1: $A = e$**.**
In this case

$$\omega \vDash [\![A]\!] \iff \langle [\![e]\!], \omega \rangle \Downarrow \mathit{Bool\ True} \qquad \text{(by definition).}$$

As ω is not a forward translated state in general, we cannot use Proposition 4.13 here. But since $[\![e]\!] = e$ and all ImpCon expressions are heap-independent it is sufficient to observe that since all values in the range of ω's store lie in $values_I$, ω and $btr(\omega)$ have the same store and so we can deduce that $\langle e, btr(\omega) \rangle \Downarrow \mathit{Bool\ True}$ as well. Therefore $btr(\omega) \vDash A$.

Base Case 2: $A = (x.f \mapsto e)$**.**
We have that

$$\begin{aligned} \omega \vDash [\![A]\!] \iff &\text{there exist Viper states } \omega_1,\ \omega_2 \text{ such that} \\ &\omega_1 \vDash Acc(x.f),\ \omega_2 \vDash x.f = [\![e]\!] \text{ and} \\ &\omega_1 \bigoplus\nolimits_V \omega_2 = \omega \end{aligned}$$

by definition. As ω is greater than or equal to both ω_1 and ω_2 we then have that $\omega \vDash Acc(x.f)$ and $\omega \vDash x.f = [\![e]\!]$. Hence ω's store maps x to $Ref\ a$ for some address a and its store maps (a, f) to some value $v \in values_V$ with full permission by the former and also $\langle [\![e]\!], \omega \rangle \Downarrow v$ by the latter. Consequently, as all values in the range of ω's store and heap lie in $values_I$, v must be in $values_I$ and by the same reasoning as in Base Case 1 $\langle e, btr(\omega) \rangle \Downarrow v$. Now noting that since ω's store maps x to $Ref\ a$ and its permission heap maps (a, f) to $v \in values_I$ with full permission we obtain that $btr(\omega)$'s store maps x to $Ref\ a$ and its heap maps (a, f) to v as well. Thus, eventually, $btr(\omega) \vDash A$.

Base Case 3: $A = (x.f \mapsto \_)$**.**
Here by definition

$$\omega \vDash [\![A]\!] \iff \omega \vDash Acc(x.f).$$

Therefore ω's store maps x to $Ref\ a$ for some address a and its store maps (a, f) to some value $v \in values_V$ with full permission. Consequently, as all values in the range of ω's store and heap lie in $values_I$, v must be in $values_I$ and so $btr(\omega)$'s store maps x to $Ref\ a$ and its heap maps (a, f) to that value v as well. Hence $btr(\omega) \vDash A$.

Next we show the inductive cases.

Inductive Case 1: $A = \big(e \Rightarrow A_{Imp}\big)$**.**

In this case our inductive assumption is that

(IH) for all Viper states ρ such that all values in the range of its store and heap lie in $values_I$, $\rho \vDash [\![A_{Imp}]\!] \implies btr(\rho) \vDash A_{Imp}$.

We have that

$$\omega \vDash [\![A]\!] \iff \big(\langle [\![e]\!], \omega\rangle \Downarrow Bool\ True \implies \omega \vDash [\![A_{Imp}]\!]\big) \quad \text{(by definition).}$$

Again by the same reasoning as in Base case 1 if we assume that $\langle [\![e]\!], \omega\rangle \Downarrow Bool\ True$, then $\langle e, btr(\omega)\rangle \Downarrow Bool\ True$ and so $\omega \vDash [\![A_{Imp}]\!]$. By our inductive assumption then $btr(\omega) \vDash A_{Imp}$. Thus $\langle [\![e]\!], \omega\rangle \Downarrow Bool\ True \implies btr(\omega) \vDash A_{Imp}$, which by definition is equivalent to $btr(\omega) \vDash A$.

Inductive Case 2: $A = (A_1 * A_2)$**.**

This time our inductive assumption is that for $i = 1, 2$,

(IH) for all Viper states ρ such that all values in the range of its store and heap lie in $values_I$, $\rho \vDash [\![A_i]\!] \implies btr(\rho) \vDash A_i$.

$$\omega \vDash [\![A]\!] \iff \text{there exist Viper states } \omega_1,\ \omega_2 \text{ such that}$$
$$\omega_1 \vDash [\![A_1]\!],\ \omega_2 \vDash [\![A_2]\!] \text{ and } \omega_1 \bigoplus_V \omega_2 = \omega.$$

Observing that since $\omega_1 \bigoplus_V \omega_2 = \omega$ all values in the range of ω_1's and ω_2's stores and heaps must lie in $values_I$ as well, we can use (IH) to deduce that $btr(\omega_1) \vDash A_1$ and $btr(\omega_2) \vDash A_2$. Also because $\omega_1 \bigoplus_V \omega_2 = \omega$ there cannot be any heap locations which both ω_1 and ω_2 have full permission to, so by the definition of $btr(\cdot)$ the heaps of $btr(\omega_1)$ and $btr(\omega_2)$ must be disjoint, making them joinable in ImpCon. Thus $btr(\omega_1) \bigoplus_I btr(\omega_2) \vDash A$. Moreover, as $\omega_1 \bigoplus_V \omega_2 = \omega$, by Lemma 4.12 we have that $btr(\omega)$ is greater than or equal to $btr(\omega_1) \bigoplus_I btr(\omega_2)$. Thus, by monotonicity of assertions we get that $btr(\omega) \vDash A$ as desired. □

By considering Lemma 4.10, monotonicity of assertions, and the fact that forward translated states are Viper states such that all values in the range of its store and heap lie in $values_I$, we can put Proposition 4.14 and Proposition 4.15 together to obtain the following corollary.

Corollary 4.18 *Let A be an ImpCon assertion and φ an ImpCon state. Then*

$$\varphi \vDash A \iff ftr(\varphi) \vDash [\![A]\!].$$

4.3 Proving Soundness

In this section we finally prove the soundness theorem. For that purpose we first discuss the overall proof strategy.

4.3.1 The Proof Strategy

To prove the soundness theorem we want to use structural induction on the ImpCon command. Yet, Theorem 4.4 does not give rise to a very helpful inductive hypothesis. That is why our proof strategy is to split the theorem into three lemmas. One lemma should give rise to the needed inductive hypothesis and the other two should tie up the ends.

Recall the soundness theorem given in Theorem 4.4. As is done in the theorem, we assume (T, c) to be a user-writable ImpCon program, where c does not contain any local variable assignments within parallel sections, and we let P and Q be ImpCon assertions. We also assume the following.

(AV1) For all Viper states ω modelling T_{Viper},
$Ver_T(Inhale\ [\![P]\!]\ ;\ [\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!],\ \omega)$.

(AV2) For all Viper states ω modelling T_{Viper}, for all Viper statements $stmt \in [\![c]\!]_T^{EPO}$ we have that $Ver_T(stmt,\ \omega)$.

We are trying to prove that $T \vDash \{P\}\, c\, \{Q\}$. In Definition 2.13 we defined $T \vDash \{P\}\, c\, \{Q\}$ to mean that for all ImpCon states φ upholding type context T such that $\varphi \vDash P$, for all ImpCon commands c' and ImpCon states φ' such that $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ we have that c' does not abort in the next step from φ', φ' upholds type context T, and $c' = SKIP \implies \varphi' \vDash Q$. Thus, we fix ImpCon states φ, φ', ImpCon command c' and **assume**

(A1) φ upholds type context T,

(A2) $\varphi \vDash P$, and

(A3) $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$.

What we then have to **show** is that

(R1) $(c',\ \varphi') \rightarrow_T ABORT$ does not hold,

(R2) $c' = SKIP \implies \varphi' \vDash Q$.

To show (R2) want to somehow tie (AV1) to our reduction of configuration (c, φ) in ImpCon. That we do by finding a relationship between φ, φ' and the Viper states reached during verification of the statement in (AV1).

The formal semantics of sequential composition, inhale and exhale statements are given in Figure 4.2. The semantics for sequential composition say that we need the first part to reduce and we need the second part to reduce from each state in the set of states the first part reduces (f here is the function mapping each starting state to a[2] set of states the second part reduces to from

[2]By following different paths in the reduction process we can in some cases reduce to different sets. This sequential rule actually gives us the option to choose one of these sets for the set the first statement reduces to and for each reduction of the second statement.

$$\frac{\begin{array}{c} Red_T(s_1,\ \omega) \Downarrow S_1 \\ \forall \omega_1.\ \ \omega_1 \in S_1 \implies Red_T(s_2,\ f(\omega_1)) \Downarrow \omega_1 \end{array}}{Red_T(s_1\ ;\ s_2,\ \omega) \Downarrow \bigcup_{\omega_1 \in S_1} f(\omega_1)} \quad (SeqV)$$

$$\frac{A \text{ well-defined in } \omega}{Red_T(Inhale\ A,\ \omega) \Downarrow \{\omega' \mid \left(\exists \omega_{Inh}.\ \omega' = \omega \bigoplus_V \omega_{Inh} \wedge \omega_{Inh} \vDash A\right)\}} \quad (Inhale)$$

$$\frac{\begin{array}{c} \omega' \vDash A \\ \omega_{Exh} \bigoplus_V \omega' = \omega \\ \omega_{Exh} \text{ stable} \end{array}}{Red_T(Exhale\ A,\ \omega) \Downarrow \{\omega_{Exh}\}} \quad (Exhale)$$

Figure 4.2: Semantics for Viper statements: Inhale, Exhale and sequential composition.

that starting state). Sequential composition reduces to the union of sets of states that we reduce to by reducing both parts in sequence, first reducing the first part to some set of states and then reducing the second part from each of the states in that set of states.
The semantics for inhale statements require that the assertion to inhale is well-defined with respect to the store. That is, e.g. we cannot inhale the assertion $x == 5$ from a state where store variable x does not exist. An inhale statement reduces to the set of states that can be obtained by adding a state which satisfies the assertion to the current one. As the state to be added must satisfy the assertion, it must also satisfy all access predicates contained in it, and so by adding that state we gain all permissions required in the assertion (and possibly more). By monotonicity of assertions in Viper the fact that the state to be added satisfies the assertion also implies that each state in the set of states we reduce to satisfies the assertion, so the inhale statement also assumes the assertion.
The semantics for exhale statements require that the assertion holds in the current state, i.e. that the assertion can be asserted in the current state. It then splits the current state into two states, one which satisfies the assertion and which so contains a least all permissions required in it, and one state containing all other permissions. The second state is required to be *stable*, which means that it does not contain heap locations which are mapped to values but without any permission. The exhale statement then discards all permission required in the assertion (and possibly more) by reducing to only the second state which is what is left when all permissions from the first state satisfying the assertion are removed from the current state.

Let us get back to (AV1) and its implications. For visualisation purposes consider Figure 4.3, which gives a rough sketch of how the set of states

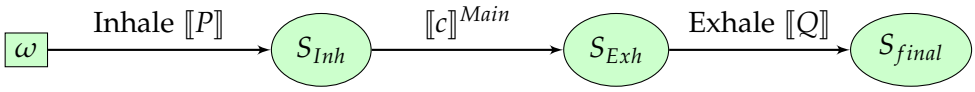


Figure 4.3: Reduction from (AV1).

evolves as we verify the statement

$$Inhale\ [\![P]\!]\ ;\ [\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!]$$

for some starting state ω. Due to (AV1) we know that this statement verifies, therefore we know it reduces to some set of states S_{final}. Let us look at the intermediate states as we process the three main sub-statements of this statement starting in ω. After the Inhale, due to (Inhale) we end up in a set of states S_{Inh} consisting of all Viper states ω' such that there exists a state ω_+ such that $\omega \bigoplus_V \omega_+ = \omega'$ and $\omega_+ \vDash [\![P]\!]$. Then, by (SeqV), we must have that for each $\omega' \in S_{Inh}$ the Viper statement $[\![c]\!]^{Main}$ must reduce to some set of states $S_{\omega'}$ and that $S_{Exh} = \bigcup_{\omega' \in S_{Inh}} S_{\omega'}$. Then, also by the semantics of sequential composition, the Exhale must also verify in all states $\omega'' \in S_{Exh}$. The exact composition of S_{final} is irrelevant for our proof, but the observation here that will be important is that if the Exhale verifies from a state ω'', then by the Viper semantics and monotonicity of assertions we must have that $\omega'' \vDash [\![Q]\!]$.

Now, how will this relate to the result we would like to prove? We show that for some well-chosen ω the forward translation of φ is in S_{Inh} and, if $c' = \text{SKIP}$, i.e. (c, φ) reduces to $(SKIP, \varphi')$ with respect to T, then the forward translation of φ' is in S_{Exh}. That is, we want to obtain the following diagram.

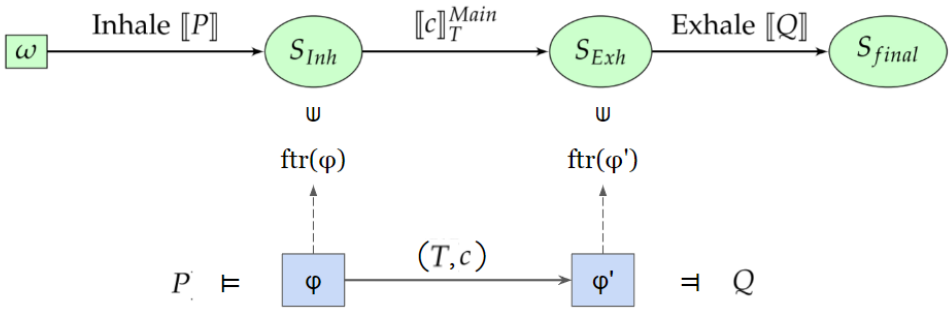


Figure 4.4: Relating the reduction from (AV1) to the reduction in ImpCon.

To obtain this diagram, by (AV1) there are three parts we need to prove.

(P1) There is some Viper state ω modelling T such that $ftr(\varphi) \in S_{Inh}$.

(P2) $[\![c]\!]^{Main}$ reduces in Viper with respect to $ftr(\varphi)$ and T to a set of states $S_{ftr(\varphi)}$ such that $ftr(\varphi') \in S_{ftr(\varphi)}$. (Note that this implies $ftr(\varphi') \in S_{Exh}$)

(P3) If the Exhale reduces from state $ftr(\varphi')$, then we must have that $\varphi' \vDash Q$.

Once we have proven these three results the diagram essentially proves (R2). The following three lemmas state the three parts formally together with all assumptions we need to prove them. In the case of (P1) and (P3) we give the proof right away. The lemma corresponding to (P2) we will prove in the next subsection.

Lemma 4.19 (Inhaling the Precondition) *Let P be an ImpCon assertion, T be an ImpCon type context and φ an ImpCon state. Assume that $\varphi \vDash P$ and the store of φ upholds T. Furthermore, assume that for all Viper states ω_{Inh} modelling T, we have that $Ver_T(Inhale\ [\![P]\!],\ \omega_{Inh})$[3]. Then there exists a Viper state ω and a set of Viper states S such that*

- *ω models T,*
- *$Red_T(Inhale\ [\![P]\!],\ \omega) \Downarrow S$, and*
- *$ftr(\varphi) \in S$.*

Proof Let P be an ImpCon assertion, T be an ImpCon type context and φ an ImpCon state. Assume that $\varphi \vDash P$ and the store of φ upholds T. Furthermore, assume that for all Viper states ω_{Inh} modelling T $Ver_T(Inhale\ [\![P]\!],\ \omega_{Inh})$.

Let ω be the Viper state consisting of the store of φ and a permission heap constructed from an empty heap (i.e. the domain of the heap is the empty set) and an empty permission mask (i.e. all heap locations map to 0). As the store of φ upholds T, we have that ω models T.

Furthermore, let S be the set of all Viper states ω' that have the same store as ω and are such that $\omega' \vDash [\![P]\!]$. As the heap and permission mask of ω are empty, we have that $\omega \bigoplus_V \omega' = \omega'$ and so S is exactly the set of states such that if the Inhale reduces then $Red_T(Inhale\ [\![P]\!],\ \omega) \Downarrow S$. Thus, as $Ver_T(Inhale\ [\![P]\!],\ \omega_{Inh})$ since ω models T, we get that $Red_T(Inhale\ [\![P]\!],\ \omega) \Downarrow S$.

Now we only need to show that $ftr(\varphi) \in S$. By assumption $\varphi \vDash P$ and so by Proposition 4.14 $ftr(\varphi) \vDash [\![P]\!]$. Therefore as ω and φ and hence also $ftr(\varphi)$ have the same store, by construction of S we have that $ftr(\varphi) \in S$.

This concludes the proof. □

[3]This assumption is actually not needed to reach the conclusions of this lemma and is also not an assumption in our Isabelle formalisation. The reason we added it here is that it allows us to skip some low-level details and it can also easily be deduced from an assumption of the soundness theorem.

Lemma 4.20 (Reducing the Command) *Let c be an ImpCon command and T be a type context, such that the pair (T, c) is a well-formed user-writable ImpCon program and c contains no assignments to local variables within parallel sections. Additionally, let φ be an ImpCon state upholding T and let S be a set of Viper states. Assume that*

- *for all Viper states ω modelling the type context T and Viper statements $s_V \in [\![c]\!]_T^{EPO}$ we have that $Ver_T(s_V,\ \omega)$, and*
- *$Red_T([\![c]\!]^{Main},\ ftr(\varphi)) \Downarrow S$.*

Then for all ImpCon commands c' and ImpCon states φ' such that $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ we have that

- *$(c',\ \varphi') \rightarrow_T ABORT$ does not hold, and*
- *$c' = SKIP \implies ftr(\varphi') \in S$.*

Lemma 4.21 (Exhaling the Postcondition) *Let Q be an ImpCon assertion, T be a Viper type context and φ an ImpCon state. Assume that $Ver_T(Exhale\ [\![Q]\!],\ ftr(\varphi))$. Then $\varphi \vDash Q$.*

Proof Let Q be an ImpCon assertion, T be a Viper type context and φ an ImpCon state. Assume that $Ver_T(Exhale\ [\![Q]\!],\ ftr(\varphi))$. Then there exists a set of Viper states S such that $Red_T(Exhale\ [\![Q]\!],\ ftr(\varphi)) \Downarrow S$. Therefore by the Viper semantics and monotonicity of assertions we must have $ftr(\varphi) \vDash [\![Q]\!]$. Thus, the result follows from Proposition 4.18. □

Note that in Lemma 4.20 (R1) was added as a conclusion. Hence after we have proven Lemma 4.20 the proof of (R1) will follow. Using all three lemmas together we obtain (R2). Thus we will obtain soundness once we have proven Lemma 4.20, which we do in the rest of this chapter.

At first glance Lemma 4.20 might not seem simpler to prove than the soundness theorem itself, but the advantage of this formulation is that it gives rise to an inductive assumption that albeit long offers all information needed when proving the Lemma by structural induction on c. That is exactly what we wanted to achieve. The proof of Lemma 4.20 we will discuss from Section 4.3.2 onward.

We wrap up this subsection by formally proving Theorem 4.4 using the lemmas we just stated.

Proof Assume (T, c) to be a user-writable ImpCon program, where c does not contain any local variable assignments within parallel sections, and we let P and Q be ImpCon assertions. Furthermore assume the following.

(AV1) For all Viper states ω modelling T,
$Ver_T(Inhale\ [\![P]\!]\ ;\ [\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!],\ \omega)$.

(AV2) For all Viper states ω modelling T, for all Viper statements $s_V \in [\![c]\!]_T^{EPO}$ we have that $Ver_T(s_V,\ \omega)$.

Fix ImpCon states φ, φ' and ImpCon command c' such that

(A1) φ upholds type context T,

(A2) $\varphi \vDash P$, and

(A3) $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$.

From (AV1) and the semantics of sequential composition we can derive that for all Viper states ω_{Inh} modelling T, we have that $Ver_T(Inhale\ [\![P]\!],\ \omega_{Inh})$. From that, (A1) and (A2) we obtain all premises in order to use Lemma 4.19. Thus, we obtain a state ω and a set of states S such that

- ω models T,
- $Red_T(Inhale\ [\![P]\!],\ \omega) \Downarrow S$, and
- $ftr(\varphi) \in S$.

We then obtain by (AV1) and the semantics of sequential composition that for some set of states S_{final},

$$Red_T([\![c]\!]^{Main}\ ;\ Exhale\ [\![Q]\!],\ ftr(\varphi)) \Downarrow S_{final}.$$

Then again by applying the rule for sequential composition backwards we get a set of states S_{Exh} such that

- $Red_T([\![c]\!]^{Main},\ ftr(\varphi)) \Downarrow S_{Exh}$, and
- for all states $\omega_{Exh} \in S_{Exh}$ we have that $Ver_T(Exhale\ [\![Q]\!],\ \omega_{Exh})$.

Using the first of these properties, (AV2), (A3) and our initial assumptions on (T, c) and φ we can use Lemma 4.20 to derive that

- $(c',\ \varphi') \rightarrow_T ABORT$ does not hold, and
- $c' = SKIP \implies ftr(\varphi') \in S_{Exh}$.

The first of these properties gives the first part we had to show in order for $T \vDash \{P\}\, c\, \{Q\}$ to hold. For the second part assume $c' = SKIP$. Then we know that $ftr(\varphi') \in S_{Exh}$ and so by the fact that for all states $\omega_{Exh} \in S_{Exh}$ we have that $Ver_T(Exhale\ [\![Q]\!],\ \omega_{Exh})$, then

$$Ver_T(Exhale\ [\![Q]\!],\ ftr(\varphi')).$$

Thus, by Lemma 4.21 we obtain that $\varphi' \vDash Q$ and we are done. □

4.3.2 Proving Sound the Translation of Commands (Except Parallel Commands)

Recall Lemma 4.20:

Lemma (Reducing the Command) *Let c be an ImpCon command and T be a type context, such that the pair (T, c) is a well-formed user-writable ImpCon program and c contains no assignments to local variables within parallel sections. Additionally, let φ be an ImpCon state upholding T and let S be a set of Viper states. Assume that*

- *for all Viper states ω modelling the type context T and Viper statements $s_V \in [\![c]\!]_T^{EPO}$ we have that $Ver_T(s_V, \omega)$, and*
- *$Red_T([\![c]\!]^{Main}, ftr(\varphi)) \Downarrow S$.*

Then for all ImpCon commands c' and ImpCon states φ' such that $(c, \varphi) \rightarrow_T^* (c', \varphi')$ we have that

- *$(c', \varphi') \rightarrow_T ABORT$ does not hold, and*
- *$c' = SKIP \implies ftr(\varphi') \in S$.*

As foreshadowed at the end of the previous subsection, we prove this lemma by structural induction on the ImpCon command. In this subsection we cover all the structural induction cases *but* the case for parallel commands. The case for parallel commands will be proven separately in the next subsection, as it is quite involved.

Proof Let c, c' be ImpCon commands, T an ImpCon type context, φ, φ' ImpCon states and S a set of Viper states. We assume the following.

(A1) (T, c) is a user-writable ImpCon program.

(A2) c contains no assignments to local variables within parallel sections.

(A3) φ upholds type context T.

(A4) $Red_T([\![c]\!]^{Main}, ftr(\varphi)) \Downarrow S$.

(A5) For all Viper states ω modelling T and for all Viper statements $s_V \in [\![c]\!]_T^{EPO}$, we have that $Ver_T(s_V, \omega)$.

(A6) $(c, \varphi) \rightarrow_T^* (c', \varphi')$.

We need to prove the following.

(R1) $(c', \varphi') \rightarrow_T ABORT$ does not hold, and that

(R2) $c' = SKIP \implies ftr(\varphi') \in S$.

We start with the base cases; SKIP, assert commands and assignments.

Base Case SKIP : $c = SKIP$**.**
In this case we have that $[\![c]\!]^{Main} = Skip$ and so by the Viper semantics and (A4) $S = \{ftr(\varphi)\}$. Furthermore by the ImpCon semantics and (A6), as there is no step we can take from SKIP, we are able to reach (c', φ') from (c, φ) in zero steps. Therefore $(c', \varphi') = (c, \varphi)$ and so $c' = SKIP$ and $\varphi' = \varphi$. Hence as we never abort after $SKIP$ (R1) holds. Also, by the above $ftr(\varphi') = ftr(\varphi) \in \{ftr(\varphi)\} = S$, giving (R2).

Base Case Assert : $c = Assert\ A$**.**
In this case we have that $[\![c]\!]^{Main} = Assert\ [\![A]\!]$ and so again by the Viper semantics and (A5), we get that $S = \{ftr(\varphi)\}$ and that $ftr(\varphi) \vDash [\![A]\!]$, as an assertion does not change the state. Considering the ImpCon semantics and (A6) there are only two cases for (c', φ'). Either we can reach (c', φ') from (c, φ) in zero steps, or we execute at least one step which must lead to $c' = SKIP$.

In the first case by Corollary 4.18, since $ftr(\varphi) \vDash [\![A]\!]$, we have that $\varphi \vDash A$, implying (R1). As $c' \neq SKIP$ we get (R2) for free.

In the second case, where we reach (c', φ') by executing a step, by the ImpCon semantics of Assert we must have that $c' = SKIP$ and $\varphi' = \varphi$. Hence, again, as we never abort after $SKIP$ (R1) holds. And by the above $ftr(\varphi') = ftr(\varphi) \in \{ftr(\varphi)\} = S$, giving (R2).

Base Case Assign Local : $c = (x := e)$**.**
By the same line of argumentation as in the Assert case, either we can reach (c', φ') from (c, φ) in zero steps, or we execute at least one step which must lead to $c' = SKIP$. As local assignments never abort the first case is trivial after obtaining that $c' = (x := e) \neq SKIP$. In the second case we again need to show $ftr(\varphi') \in S$ to finish the proof. For that we leverage (A4) again. As $[\![c]\!]^{Main} = (x := [\![e]\!])$ and so by the Viper semantics and (A4) there exists a value v such that $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow v$ and $S = \{\omega\}$ where ω is the same as $ftr(\varphi)$ except that the store of ω maps x to v. We can then use that $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow v$ together with Proposition 4.13 to obtain that $\langle e,\ \varphi\rangle \Downarrow v$ so that by the ImpCon semantics φ' is the same as φ except that the store of φ' maps x to v. Then by observing that $\omega = ftr(\varphi')$ we obtain the desired result that $ftr(\varphi') \in S$.

Base Case Assign to field: $c = (x.f := e)$**.**
By the same line of argumentation as in the Assert case, either we can reach (c', φ') from (c, φ) in zero steps, or we execute at least one step which must lead to $c' = SKIP$.

This time to show (R1) in the first case we need to prove that the store of φ maps x to a reference value containing some address a and that (a, f) lies in the domain of the heap of φ, by definition of abort. All

this can be obtained through the fact that $[\![c]\!]^{Main} = (x.f := [\![e]\!])$ and $Red_T([\![c]\!]^{Main}, ftr(\varphi)) \Downarrow S$.

The second case follows the same line of arguments as the Assign Local case, only that this time we update the heap instead of the store.

Base Case Assign from field : $c = (x := y.f)$**.**
We perform the usual case split.

To show (R1) in the first case we need to prove that the store of φ maps y to a reference value containing some address a and that (a, f) lies in the domain of the heap of φ, by definition of abort. Once more, all this can be obtained through the fact that $[\![c]\!]^{Main} = (x := y.f)$ and $Red_T([\![c]\!]^{Main}, ftr(\varphi)) \Downarrow S$.

In the second case we again need to show $ftr(\varphi') \in S$ to finish the proof. For that we leverage (A4) again. As $[\![c]\!]^{Main} = (x := y.f)$ and so by the Viper semantics and (A4) there exists a value v such that the heap of $ftr(\varphi)$ maps (a, f) to v and its permission mask maps (a, f) to 1. As $ftr(\varphi)$ is a forward-translated state that implies the heap of φ must map (a, f) to v as well. Hence by the ImpCon semantics φ' is the same as φ except that the store of φ' maps x to v. Furthermore by the reduction we have that $S = \{\omega\}$, where ω is the same as $ftr(\varphi)$ except that the store of ω maps x to v. Thus we again observe that $\omega = ftr(\varphi')$ and obtain the desired result that $ftr(\varphi') \in S$.

Base Case Alloc : $c = (x := alloc(f_1, f_2, \ldots, f_n))$**.**
We perform the usual case split.

As alloc commands never abort the first case is trivial after obtaining that $c' = (x := alloc(f_1, f_2, \ldots, f_n)) \neq SKIP$.

Hence we only need to show (R2) in the second case. We know in this case that $(c, \varphi) \rightarrow_T (SKIP, \varphi')$. From this we can derive the following side-conditions through the ImpCon semantics of allocation.

(D1) $f_1, f_2, \ldots, f_n$ are distinct,

(D2) T maps x to a reference type,

and for some address a

(D3) (a, f_i) does not lie in the domain of the heap of φ for $i = 1, \ldots, n$.

(D4) φ' is the same as φ except that the store of φ' maps x to $Ref\ a$ and the heap φ' maps (a, f_i) to $Null$ for $i = 1, \ldots, n$.

As $[\![c]\!]^{Main} = (Havoc\ x\ ;\ Inhale\ Acc(x.f_1)\ \&\&\ Acc(x.f_2)\ \&\&\ \ldots\ \&\&\ Acc(x.f_n))$, we know by (A4) that $Havoc\ x$ reduces in Viper w.r.t T and $ftr(\varphi)$ to some set S_{Havoc}. By the semantics of Havoc, by (D2) as $Ref\ a$ is of

type reference, S_{Havoc} contains the state ω which is the same as $ftr(\varphi)$ except that the store of ω maps x to $Ref\ a$.

Then by the semantics of sequential composition the Inhale reduces w.r.t. T and ω to some set S_{Inh} and $S_{Inh} \subseteq S$. Let ω_{Inh} be the state which is the same as ω except that the heap of ω_{Inh} maps (a, f_i) to $Null$ for $i = 1, \ldots, n$. Now if we can show that S_{Inh} contains the state ω_{Inh}, then we are done, because then by (D4) and the definition of ω, $ftr(\varphi') = \omega_{Inh} \in S_{Inh} \subseteq S$.

To see that S_{Inh} contains ω_{Inh} we need to show that there exists some Viper state ω_+ such that $\omega_{Inh} = \omega \bigoplus_V \omega_+$ and

$$\omega_+ \vDash Acc(x.f_1)\ \&\&\ Acc(x.f_2)\ \&\& \ldots \&\&\ Acc(x.f_n).$$

That is due to the semantics of Inhale in Viper, which adds states joinable with the current state and satisfying the assertion to the current state. We can take ω_+ to be the state consisting of the store of ω and the permission heap that maps (a, f_i) to $Null$ with full permission for $i = 1, \ldots, n$. Since by (D3) (a, f_i) does not lie in the domain of the heap of φ for $i = 1, \ldots, n$, the same holds true for $ftr(\varphi)$ and so also for ω. Hence we can add ω and ω_+. It is then easy to see that $\omega_{Inh} = \omega \bigoplus_V \omega_+$. Moreover, clearly $\omega_+ \vDash Acc(x.f_1)\ \&\&\ Acc(x.f_2)\ \&\& \ldots \&\&\ Acc(x.f_n)$ as by (D1) the right-hand-side cannot evaluate to false. Thus $\omega_{Inh} \in S_{Inh}$ and we are done.

Thus we have shown all the base cases. Now we move on to the structural commands; Sequential Composition, If-statements, and parallel commands. Note that by (A1) we do not need to consider the case $c = (c_1 \parallel c_2)$. For all structural commands we additionally have our inductive assumption at our disposal. That is for each command c_i that c consists of (e.g. $c = c_1\ ;\ c_2$ consists of c_1 and c_2) we get the following assumption.

(IH) For any ImpCon states φ_i, φ_i', and ImpCon command c_i', set of Viper states S_i such that

- $(c_i,\ \varphi_i) \rightarrow_T^* (c_i',\ \varphi_i')$,
- φ_i upholds type context T,
- $Red_T([\![c_i]\!]^{Main},\ ftr(\varphi_i)) \Downarrow S_i$,
- For all Viper states ω modelling T, for all Viper statements $s_V \in [\![c_i]\!]_T^{EPO}$ we have that $Ver_T(s_V,\ \omega)$,
- c_i contains no assignments to local variables within parallel sections, and
- (T, c_i) is a user-writable program

we have that c_i' does not abort in the next step from φ_i', and that $c_i' = SKIP \implies ftr(\varphi_i') \in S_i$.

Note that the last two bullet points only depend on c_i and are always true. That is because by (A1) and (A2) both statements hold for c and as the properties in both statements are defined inductively they can easily be shown to be true for each command c_i that c consists of. Hence whenever we use (IH) we will omit the last two criteria. The third to last criterion can also be shown to hold in general for all structural commands c by noting that $[\![c_i]\!]_T^{EPO} \subseteq [\![c]\!]_T^{EPO}$ by the definition of $[\![\cdot]\!]_T^{EPO}$ and so the criterion holds by (A5). Thus whenever we wish to use the inductive hypothesis we only need to show the first three criteria.

Inductive Case Seq : $c = (c_1 \,;\, c_2)$**.**

Note that by the ImpCon semantics of sequential composition and (A6) there can only be two cases; Either there exists c_1' such that $(c_1,\ \varphi) \rightarrow_T^* (c_1',\ \varphi')$ and $c' = \left(c_1' \,;\, c_2\right)$ or there exists φ_1' such that $(c_1,\ \varphi) \rightarrow_T^* (SKIP,\ \varphi_1')$ and $(c_2,\ \varphi_1) \rightarrow_T^* (c',\ \varphi')$. That is, when reaching (c', φ') we are either still executing c_1 or we finished executing c_1 and are executing c_2.

Subcase: Still executing c_1**.** Fix c_1' such that $(c_1,\ \varphi) \rightarrow_T^* (c_1',\ \varphi_1')$ and $c' = \left(c_1' \,;\, c_2\right)$. As $c' = \left(c_1' \,;\, c_2\right) \neq SKIP$ we get (R2) for free. For (R1) we use (IH) to obtain that c_1' does not abort in the next step from φ' which implies (R1) by the definition of abort for Seq commands. Choosing $\varphi_1 \triangleq \varphi$ and $\varphi_1' \triangleq \varphi'$ we obtain the first two criteria through the above and (A3). Moreover we get the fourth criterion by (A4) as $[\![c]\!]^{Main} = [\![c_1]\!]^{Main} \,;\, [\![c_2]\!]^{Main}$ and so $Ver_T([\![c_1]\!]^{Main},\ ftr(\varphi_1))$. Thus we can use (IH) and conclude this case.

Subcase: Executing c_2**.**Now fix φ_1' such that $(c_1,\ \varphi) \rightarrow_T^* (SKIP,\ \varphi_1')$ and $(c_2,\ \varphi_1) \rightarrow_T^* (c',\ \varphi')$. Then by choosing $\varphi_1 \triangleq \varphi$ and $c_1' \triangleq SKIP$ and S_1 such that $Red_T([\![c_1]\!]^{Main},\ ftr(\varphi_1)) \Downarrow S_1$, by a similar argument as in the first case we can use (IH) to this time obtain that $ftr(\varphi_1') \in S_1$ since $c_1' = SKIP$. Because $ftr(\varphi_1') \in S_1$ and $Red_T([\![c_1]\!]^{Main},\ ftr(\varphi)) \Downarrow S_1$ by the Viper semantics for sequential composition and (A5) there must exist a set S_2 that $[\![c_2]\!]^{Main}$ reduces to w.r.t. T and $ftr(\varphi_1')$, and that $S_2 \subseteq S$. By noting that executing a step in the ImpCon semantics from a state which upholds type context T and for which only finitely many addresses are allocated on the heap can only result in a state for which both of these criteria hold as well, we can derive that both of these criteria hold for φ_1 since they held for φ. Hence we have all we need to use (IH) for c_2 with $\varphi_2 \triangleq \varphi_1'$, $\varphi_2' \triangleq \varphi'$, $c_2' \triangleq c_2$. Using (IH) we conclude that c' does not abort in the next step from φ', and that $c' = SKIP \implies ftr(\varphi') \in S_2 \subseteq S$.

Inductive Case If : $c = If\ e\ then\ c_1\ else\ c_2$**.**

By the ImpCon semantics of If-statements, (A6) and well-formedness of

c there can only be two cases, other than the trivial case in which $c = c'$; Either $\langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ True}$ and $(c_1,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ or $\langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ False}$ and $(c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$. That is, either the If-condition is true or false. As the proofs of both cases are very similar we only prove the first case here.

Assume that $\langle e,\ \varphi\rangle \Downarrow \mathit{Bool\ True}$ and $(c_1,\ \varphi) \rightarrow_T^* (c',\ \varphi')$. Then by Proposition 4.13 $\langle [\![e]\!],\ ftr(\varphi)\rangle \Downarrow \mathit{Bool\ True}$ and so by the Viper semantics of If-statements as $[\![c]\!]^{Main} = \mathit{If}\ [\![e]\!]\ \mathit{then}\ [\![c_1]\!]^{Main}\ \mathit{else}\ [\![c_2]\!]^{Main}$, we have that $Red_T([\![c_1]\!]^{Main},\ ftr(\varphi)) \Downarrow S$. By this fact, (A3) and our assumption that $(c_1,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ using (IH) for c_1 concludes the proof. □

The last case we need the prove is that of c being the parallel command. As this is by far the most challenging case we decided to devote the entire next subsection to it.

4.3.3 Proving Sound the Translation of Parallel Commands

Now we prove the last inductive case.

Throughout this subsection let $c \triangleq (\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\})$. We show that in this case as well (R1) and (R2) hold.

For this proof we leverage the parallel rule and frame rule from separation logic. For that purpose we have formally proven the equivalence between our definition of Hoare triples in ImpCon and that of Viktor Vafeiadis in [12] to ensure that these rules are valid for ImpCon. We have, however, not reproven the rules themselves in Isabelle. For context we repeat the adapted parallel rule and frame rule from [12] here.

$$\frac{\begin{array}{c} T \vDash \{P_1\}\, c_1\, \{Q_1\} \\ T \vDash \{P_2\}\, c_2\, \{Q_2\} \\ fv(P_1, c_1, Q_1) \cap wr(c_2) = \varnothing \\ fv(P_2, c_2, Q_2) \cap wr(c_1) = \varnothing \end{array}}{T \vDash \{P_1 \ast P_2\}\, c_1 \parallel c_2\, \{Q_1 \ast Q_2\}}\ (\mathit{parallel\ rule}) \qquad \frac{T \vDash \{P\}\, c\, \{Q\} \quad fv(F) \cap wr(c) = \varnothing}{T \vDash \{P \ast F\}\, c\, \{Q \ast F\}}\ (\mathit{frame\ rule})$$

To give a rough sketch of the proof, we will show that $T \vDash \{Pre_1\}\, c_1\, \{Post_1\}$ and $T \vDash \{Pre_2\}\, c_2\, \{Post_2\}$ using the inductive hypothesis and then proceed to use the parallel rule to obtain $T \vDash \{Pre_1 \ast Pre_2\}\, c_1 \parallel c_2\, \{Post_1 \ast Post_2\}$. From that we will show (R1) via the Hoare triple definition. To show (R2), we will use the frame rule as well to get that

$$T \vDash \{Pre_1 \ast Pre_2 \ast F\}\, c_1 \parallel c_2\, \{Post_1 \ast Post_2 \ast F\}$$

for a well-chosen frame F, as described in Section 3.1.3, and use this together with (A4) to conclude the desired result.

Now, let us get started with the proof.

Obtaining the Hoare triple from the parallel rule. Since the proofs of $T \vDash \{Pre_1\}\, c_1\, \{Post_1\}$ and $T \vDash \{Pre_2\}\, c_2\, \{Post_2\}$ are analogous we only prove $T \vDash \{Pre_1\}\, c_1\, \{Post_1\}$ here. Fix ImpCon states φ_1, φ_1' and ImpCon command c_1'. Assume that $(c_1,\ \varphi_1) \rightarrow_T^* (c',\ \varphi_1')$, φ_1 upholds type context T, $\varphi_1 \vDash Pre_1$ and only finitely many addresses are allocated on the heap of φ_1.
Then in order to use (IH) for c_1 all we have left to show is that $[\![c_1]\!]^{Main}$ reduces to some set S_1 w.r.t. T and $ftr(\varphi_1)$. This we do using (A5), that is, using that the extra proof obligation emitted when translating the parallel command verifies. By definition of $[\![\cdot]\!]_T^{EPO}$ we have that

$$\big(Inhale[\![Pre_1]\!]\ ;\ [\![c_1]\!]^{Main}\ ;\ Exhale[\![Post_1]\!]\big) \in [\![c]\!]_T^{EPO}.$$

Therefore by (A4) we have that $Inhale[\![Pre_1]\!]\ ;\ [\![c_1]\!]^{Main}\ ;\ Exhale[\![Post_1]\!]$ reduces in Viper w.r.t. T and any state that models T. As we assumed that φ_1 upholds type context T and $\varphi_1 \vDash Pre_1$, we can use Lemma 4.19 to obtain such a state ω and a set of states S_{Inh1} for which $Inhale\ [\![P]\!]$ reduces to S_{Inh1} in Viper with respect to T_{Viper} and ω, and $ftr(\varphi_1) \in S_{Inh1}$. Then, using that $Inhale[\![Pre_1]\!]\ ;\ [\![c_1]\!]^{Main}\ ;\ Exhale[\![Post_1]\!]$ reduces in Viper w.r.t. T and ω, by the Viper semantics of sequential composition there exists a set S_1 such that $[\![c_1]\!]^{Main}$ reduces to S_1 w.r.t. T and $ftr(\varphi_1)$ and $Exhale[\![Post_1]\!]$ reduces w.r.t. T and all states $\omega_S \in S_1$. From this we have what we wanted in order to use (IH).
Using (IH) for φ_1, φ_1', c_1' and S_1 we obtain that c_1' does not abort in the next step from φ_1', and that $c_1' = SKIP \implies ftr(\varphi_1') \in S_1$. What we need to show in order to conclude the proof is that c_1' does not abort in the next step from φ_1', φ_1' upholds type context T, and $c_1' = SKIP \implies \varphi_1' \vDash Post_1$, the first of which we have already shown. To show the third part recall that $Exhale[\![Post_1]\!]$ reduces w.r.t. T and all states $\omega_S \in S_1$ and $c_1' = SKIP \implies ftr(\varphi_1') \in S_1$. Thus we are able to apply Lemma 4.21 to obtain that $c_1' = SKIP \implies ftr(\varphi_1') \vDash Post_1$. For the second part, just as we did in Section 4.3.1, we note that since $(c_1,\ \varphi_1) \rightarrow_T^* (c_1',\ \varphi_1')$ and φ_1 upholds T by the ImpCon semantics φ_1' must uphold T as well. This concludes the proof that $T \vDash \{Pre_1\}\, c_1\, \{Post_1\}$.

Noting that by (A2) the set of local variables written to in c_1 resp. c_2 is empty, we are now all set to use the parallel rule from separation logic. From the proven fact that $T \vDash \{Pre_1\}\, c_1\, \{Post_1\}$ and $T \vDash \{Pre_2\}\, c_2\, \{Post_2\}$ we derive that $T \vDash \{Pre_1 \ast Pre_2\}\, c_1 \parallel c_2\, \{Post_1 \ast Post_2\}$.

Proving (R1). This Hoare triple is already sufficient to prove (R1). We will simply show that we can without loss of generality assume $(c_1 \parallel c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ and $\varphi \vDash Pre_1 \ast Pre_2$, so that by (A3) and the established fact that $T \vDash \{Pre_1 \ast Pre_2\}\, c_1 \parallel c_2\, \{Post_1 \ast Post_2\}$ we can use our Hoare triple definition to obtain (R1).
The reason that we can w.l.o.g. assume that $(c_1 \parallel c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ is because of the ImpCon semantics of the parallel command. By (A6) either

we do not execute a step and $(c, \varphi) = (c', \varphi')$ or we do execute at least one step, the first of which can only lead to $(c_1 \parallel c_2, \varphi)$, so that we must have $(c_1 \parallel c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ as desired. As we never abort on a parallel command (R1) is trivially true in the first case and so is (R2) since $c \neq SKIP$. Thus we can w.l.o.g. assume that $(c_1 \parallel c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$.
To show that $\varphi \vDash Pre_1 \ * \ Pre_2$ we will use (A4). For

$$c = (\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\})$$

we have that

$$[\![c]\!]^{Main} = Exhale\ [\![Pre_1]\!]\ ;\ Exhale\ [\![Pre_2]\!]\ ;\ Inhale\ [\![Post_1]\!]\ ;\ Inhale\ [\![Post_2]\!].$$

So by (A4) we have that

$$Red_T(Exhale\,[\![Pre_1]\!]\ ;\ Exhale\,[\![Pre_2]\!]\ ;\ Inhale\,[\![Post_1]\!]\ ;\ Inhale\,[\![Post_2]\!], ftr(\varphi)) \Downarrow S.$$

We can rewrite this to say that

$$Red_T(Exhale\ [\![Pre_1]\!]\ \&\&\ [\![Pre_2]\!]\ ;\ Inhale\ [\![Post_1]\!]\ \&\&\ [\![Post_2]\!],\ ftr(\varphi)) \Downarrow S.$$

Then through the Viper semantics of Exhale we can derive that $ftr(\varphi) \vDash [\![Pre_1]\!]\ \&\&\ [\![Pre_2]\!]$ which is by definition of $[\![\cdot]\!]$ equivalent to saying $ftr(\varphi) \vDash [\![Pre_1 \ * \ Pre_2]\!]$. Thus Proposition 4.18 gives us that $\varphi \vDash Pre_1 \ * \ Pre_2$ as desired. Hence we have all we need in order to use our Hoare triple definition to show (R1).

Proving (R2). Proving (R2) for the parallel command will be a bit trickier. Assume $c' = SKIP$. We need to show that

$$Exhale\ [\![Pre_1]\!]\ \&\&\ [\![Pre_2]\!]\ ;\ Inhale\ [\![Post_1]\!]\ \&\&\ [\![Post_2]\!]$$

reduces to a set containing $ftr(\varphi')$ in Viper w.r.t. T and $ftr(\varphi)$. From the Viper semantics of Exhale, introduced in Figure 4.2, and monotonicity of assertions we obtain Viper states ω, ω_{Exh} and the following properties.

- **(P1)** $ftr(\varphi) \vDash [\![Pre_1 \ * \ Pre_2]\!]$,
- **(P2)** the Exhale reduces to the set $\{\omega_{Exh}\}$,
- **(P3)** $\omega \vDash [\![Pre_1 \ * \ Pre_2]\!]$,
- **(P4)** $ftr(\varphi) = \omega_{Exh} \bigoplus_V \omega$, and
- **(P5)** ω_{Exh} is a stable state (i.e. there are no heap locations that are mapped to some value with zero permission).

Ultimately by (P2), to show that

$$Exhale\ [\![Pre_1]\!]\ \&\&\ [\![Pre_2]\!]\ ;\ Inhale\ [\![Post_1]\!]\ \&\&\ [\![Post_2]\!]$$

reduces to a set containing $ftr(\varphi')$ in Viper w.r.t. T and $ftr(\varphi)$, we need to show that $ftr(\varphi')$ lies among the states we can reach by inhaling $[\![Post_1]\!] * [\![Post_2]\!]$ from ω_{Exh}. By the Viper semantics of Inhale that translates to having to show that there exists some Viper state ω_{Inh} such that

(RI1) $ftr(\varphi') = \omega_{Exh} \bigoplus_V \omega_{Inh}$, and

(RI2) $\omega_{Inh} \vDash [\![Post_1]\!]$ && $[\![Post_2]\!]$.

Therefore once we have shown (RI1) and (RI2) we are done.

Before we dive into the proof, let us get a clearer picture of what we are trying to do here. We want to get from ω_{Exh} to $ftr(\varphi')$ by adding another state. Due to (A2) we need to only consider the changes to the permission heap here. As there are no assignments to local variables in c_1 and c_2, the store will not change when we execute $(c_1 \parallel c_2)$ and so as $(c_1 \parallel c_2,\ \varphi) \rightarrow_T^* (c',\ \varphi')$, φ and φ' must have the same store. By definition of forward translation this implies $ftr(\varphi)$ and $ftr(\varphi')$ must have the same store as well. Then due to (P4) $ftr(\varphi')$ and ω_{Exh} must also have the same store.

As mentioned before we want to leverage the frame rule. The intuition is that we want our frame to ensure that $ftr(\varphi')$ is greater than or equal to ω_{Exh}, which implies by definition that we can add some Viper state ω_{Inh}. Therefore it is not too far-fetched to incorporate ω_{Exh} directly into our frame. That is, by showing that the frame holds in $ftr(\varphi')$ we can directly deduce that ω_{Exh} is a part of $ftr(\varphi')$.

The frame has to be defined on ImpCon states. The first naive idea that comes to mind here might be to use pur notion of forward translation and set our ImpCon frame to be $F'(\gamma) \triangleq (ftr(\gamma) = \omega_{Exh})$[4]. But that would give rise to the following problem.

Consider for example the case where $[\![Pre_1]\!] \triangleq acc(x.f)$ and $[\![Pre_2]\!] \triangleq acc(y.f)$ and $ftr(\varphi)$ has full permission to exactly the three heap locations associated with $x.f$, $y.f$ and $z.f$. So (P1) holds which implies there is a state ω satisfying (P3) and (P4). It could then be the case that ω is the state that has full permission to access the fields $x.f$, $y.f$ and $\frac{2}{3}$ permission to access $z.f$, which would make it a sub state of $ftr(\varphi)$ and ensure that (P3) holds. Therefore by (P4), ω_{Exh} must be the state having exactly permission $\frac{1}{3}$ to the heap location associated with $z.f$ and no other heap locations. But forward translation always results in integral permissions! Hence there is no ImpCon state which could satisfy the frame F'.

Recall that we want our frame to ensure that $ftr(\varphi')$ is greater than or equal to ω_{Exh}. Hence, sticking to our previous example we would need the ImpCon

[4]Note that this is not an ImpCon assertion, but rather a function from ImpCon states to booleans. We interpret this function as an assertion by saying that it is satisfied by a state if it maps that state to *True*.

frame to ensure that $ftr(\varphi')$ has at least permission $\frac{1}{3}$ to the heap location associated with $z.f$. Hence we also cannot define the frame via our back translate function from Section 4.2. That is, we cannot simply define our frame as $F''(\gamma) \triangleq (\gamma = btr(\omega_{Exh}))$. In our example, as ω_{Exh} has less than full permission to access $z.f$, and no other permissions, the back translation of ω_{Exh} is a state with an empty heap. Thus, the frame does not ensure that $z.f$ is allocated in φ' and so it does not help us in determining if $ftr(\varphi')$ is greater than or equal to ω_{Exh}.

Therefore, naturally, we want our ImpCon frame to ensure instead that $z.f$ is allocated in φ', so that $ftr(\varphi')$ has *more* permission to $z.f$ than ω_{Exh}. Our definition of back translation effectively rounded down permissions by deallocating all heap locations with less than full permissions. Hence for our frame we define a similar notion which effectively *rounds up* permissions by making sure all heap locations with non-zero permission stay allocated.

Definition 4.22 *The* backward translation over-approximating permissions *of a Viper state, $btr_{over}(\cdot)$ is defined as*

$$
\begin{aligned}
&btr_{over}\colon State_V \to State_I \\
&\quad (s,(m,h)) \mapsto (s|_S, h|_H), \text{ where} \\
&\qquad S = \{v \in varName \mid s(v) \in values_I\}, \text{ and} \\
&\qquad H = \{hl \in Adresses \times Fields \mid h(hl) \in values_I \text{ and } m(hl) > 0\}.
\end{aligned}
$$

The function $btr_{over}(\cdot)$ also has some interesting properties in connection with forward translation and usual backward translation.

Lemma 4.23 *Let φ be an ImpCon state. Then $btr_{over}(ftr(\varphi)) = \varphi$.*

Lemma 4.24 *Let ω be a Viper state that only contains values in $values_I$. Then $ftr(btr_{over}(\omega))$ is greater than or equal to ω.*

Lemma 4.25 *Let ω_1,ω_2 be a Viper states, and φ an ImpCon state. Assume that*

$$\omega_1 \bigoplus_V \omega_2 = ftr(\varphi).$$

Then

$$btr(\omega_1) \bigoplus_V btr_{over}(\omega_2) = \varphi.$$

The first two lemmas can easily be seen to hold by inspecting heap locations cases by case. The third lemma can be shown as follows.

Proof Let ω_1,ω_2 be a Viper states, and φ an ImpCon state. Assume that $\omega_1 \bigoplus_V \omega_2 = ftr(\varphi)$. Then the heaps of $btr(\omega_1)$ and $btr_{over}(\omega_2)$ must be disjoint for otherwise there must be a heap location for which we have full permission in ω_2 and non-zero permission in ω_1 resulting in more than

full permission in $ftr(\varphi)$, which is impossible. Moreover, as $\omega_1 \bigoplus_V \omega_2 = ftr(\varphi)$, all three Viper states must have the same store. Therefore $btr(\omega_1)$ and $btr_{over}(\omega_2)$ also have the same store. Thus, $btr(\omega_1)$ and $btr_{over}(\omega_2)$ are joinable.

Now we show $btr(\omega_1)$ and $btr_{over}(\omega_2)$ must add up to φ. As $ftr(\varphi)$ has full permission to all heap locations allocated in φ, either both ω_1 and ω_2 have fractional permissions to such a heap location with the same value so that $btr_{over}(\omega_2)$ carries the correct value, or one of ω_1 and ω_2 has full permission with the correct value while the other has zero permission. In that case the corresponding state of φ_2 and φ_1 carries the same value. For all heap locations not allocated in φ, $ftr(\varphi)$ has zero permission and hence so do ω_{Exh} and ω ensuring the heap location exists in neither of $btr(\omega_1)$ and $btr_{over}(\omega_2)$. Thus, $btr(\omega_1) \bigoplus_V btr_{over}(\omega_2) = \varphi$. □

Using this version of backward translation we then define our frame choice as

$$F(\gamma) \triangleq (\gamma = btr_{over}(\omega_{Exh}))\,.$$

Just like for the previous frame choice, as the set of local variables written to in $c_1 \parallel c_2$ is empty, we may use the frame rule on the Hoare triple $T \vDash \{Pre_1 * Pre_2\}\, c_1 \parallel c_2\, \{Post_1 * Post_2\}$ that we have shown before and thus obtain the Hoare triple

$$T \vDash \{Pre_1 * Pre_2 * F\}\, c_1 \parallel c_2\, \{Post_1 * Post_2 * F\}.$$

To be able to derive any information from this Hoare triple we need to first formally show that $\varphi \vDash Pre_1 * Pre_2 * F$. Recall the definition the separating conjunction for ImpCon states.

$$\begin{aligned}\gamma \vDash A_1 * A_2 \iff &\text{there exist ImpCon states } \gamma_1,\ \gamma_2 \text{ such that}\\ &\gamma_1 \vDash A_1,\ \gamma_2 \vDash A_2 \text{ and } \gamma_1 \bigoplus_I \gamma_2 = \gamma.\end{aligned}$$

Hence we need to show there exist φ_1, φ_2 such that $\varphi_1 \vDash Pre_1 * Pre_2$, $F(\varphi_2)$ holds, and $\varphi_1 \bigoplus_I \varphi_2 = \varphi$. We choose $\varphi_1 \triangleq btr(\omega)$ and $\varphi_2 \triangleq btr_{over}(\omega_{Exh})$. It is clear that $F(\varphi_2)$ holds. To see that $\varphi_1 \vDash Pre_1 * Pre_2$ recall that by (P3) $\omega \vDash [\![Pre_1 * Pre_2]\!]$. Hence, as ω by (P4) is part of a translated state and therefore contains only values in $values_I$, we can use Proposition 4.15. That $\varphi_1 \bigoplus_I \varphi_2 = \varphi$ we obtain from Lemma 4.25 and (P4). Thus $\varphi \vDash Pre_1 * Pre_2 * F$.

Then finally by (A3) and (A6) we can use our Hoare triple to obtain that $\varphi' \vDash Post_1 * Post_2 * F$ and so by definition there exist ImpCon states φ'_1 and φ'_2 such that $\varphi'_1 \vDash Post_1 * Post_2$, $F(\varphi'_2)$ holds, and $\varphi'_1 \bigoplus_I \varphi'_2 = \varphi'$.

We use this to find a Viper state ω_{Inh} such that

(RI1) $ftr(\varphi') = \omega_{Exh} \bigoplus_V \omega_{Inh}$, and

(RI2) $\omega_{Inh} \vDash [\![Post_1]\!] \;\&\&\; [\![Post_2]\!]$.

Recall that this is what we want to prove.

As $\varphi'_1 \bigoplus_I \varphi'_2 = \varphi'$, by Lemma 4.11 we can derive that

(D) $ftr(\varphi'_1) \bigoplus_V ftr(\varphi'_2) = ftr(\varphi')$.

Furthermore, by Proposition 4.14, since $\varphi'_1 \vDash Post_1 \;*\; Post_2$, we have that $ftr(\varphi'_2) \vDash [\![Post_1 \;*\; Post_2]\!]$. Moreover, because $F(\varphi'_2)$ holds, we have that $\varphi'_2 = btr_{over}(\omega_{Exh})$. As ω_{Exh} is by (P4) a part of $ftr(\varphi)$, a forward translated state, ω_{Exh} can only have values in $values_I$. Hence by Lemma 4.24 $ftr(\varphi'_2) = ftr(btr_{over}(\omega_{Exh}))$ is greater than or equal to ω_{Exh}.

We now know that $ftr(\varphi'_2)$ is greater than or equal to ω_{Exh} and that we can add $ftr(\varphi'_1)$ such that $ftr(\varphi'_1) \vDash [\![Post_1 \;*\; Post_2]\!]$ to $ftr(\varphi'_2)$ to obtain $ftr(\varphi')$ by (D). Hence choosing ω_{Inh} to be $ftr(\varphi'_1)$ plus the difference between $ftr(\varphi'_2)$ and ω_{Exh} works as then ω_{Inh} satisfies (RI1) and by monotonicity of assertions, because ω_{Inh} is then greater than or equal to $ftr(\varphi'_1)$, it also satisfies (RI2), as required.

Chapter 5

Lessons Learnt

When proving soundness not everything fell into place right away. In this chapter we go through some choices we initially made that ended up being sub-optimal and were therefore not used in the final version of the proof.

In Section 5.1 we will talk about how we initially defined the ImpCon states differently and in Section 5.2 we will present an alternative proof of the final part of the Parallel-command case of Theorem 4.4 by constructing a frame explicitly, instead of reusing the one derived from the Viper semantics.

5.1 Alternative ImpCon Heap Definition

Recall that in Definition 2.4 we defined an ImpCon heap as a partial map from heap locations to values. That is,

$$h : (address, field) \rightharpoonup values_I.$$

This was in fact not our first attempt at defining the heap. Initially we defined an ImpCon heap as a partial map from addresses to partial maps from field identifiers to values, that is

$$h_{alt} : address \rightharpoonup (field \rightharpoonup values_I).$$

Here a heap location (a, f) exists if both $a \in dom(h_{alt})$ and $f \in dom(h_{alt}(a))$. The value of that heap location is then $(h_{alt}(a))(f)$.

This heap definition was due to what we defined allocation to mean in ImpCon. Recall that alloc commands assign a fresh address that *has not already been used* to a reference variable. Hence, using this heap definition we wanted to be able to express the situation that an address had already been used to allocate memory for some reference variable but all its fields had been deallocated. That is, the address exists on the heap but there are currently no

allocated fields using that address. In that case the address is not considered fresh and we would not like to choose that address during allocation. Using our alternative heap definition, we can check if some address a already exists simply by observing if $a \in dom(h_{alt})$.

However, this alternative heap definition caused some extra complications during the process of proving soundness of our translation. The root of some of these complications lies in the mismatch it caused between the ImpCon state and the Viper state. When forward translating an ImpCon state into a Viper state multiple ImpCon heaps would translate to the same Viper heap. From Definition 4.2 recall that a Viper heap was defined as a partial map from heap locations to values. Thus, using our alternative heap definition, forward translation would be defined as follows.

$$
\begin{aligned}
ftr_{alt}&: State_I \to State_V \\
&(s, h_{alt}) \mapsto (s, (m, h)), \text{ where} \\
(a, f) \in dom(h) &\iff a \in dom(h_{alt}) \text{ and } f \in dom(h_{alt}(a)), \text{ and} \\
(a, f) \in dom(h) &\implies h((a, f)) = h_{alt}(a)(f), \text{ and} \\
m(hl) &= \begin{cases} 1 & \text{if } hl \in dom(h) \\ 0 & \text{if } hl \notin dom(h) \end{cases}
\end{aligned}
$$

Due to the mismatch between state definitions, both when an address does not lie in the domain of the ImpCon heap, and when an address is mapped to the empty map (i.e. the domain of the map the address is mapped to is empty), the heap of the resulting state would not contain any heap locations involving that address in its domain. That is, both cases have the same effect on the resulting forward translated state. This is visualised in Figure 5.1.

Hence we defined equivalence classes of ImpCon heaps, where two ImpCon heaps are in the same equivalence class if they give rise to the same Viper heap during forward translation. That also meant that when considering backward translation, on the other hand, on top of dealing with fractional permissions, we had to decide which particular ImpCon heap from an equivalence class of ImpCon heaps should be taken as the heap of the back translated state. Moreover, no matter which ImpCon heap we choose from each equivalence class when defining the back translation $btr_{alt}(\cdot)$, it will, among other things, never be the case that $btr_{alt}(ftr_{alt}(h_{alt})) = h_{alt}$ in general. If in Figure 5.1 we choose the first ImpCon heap to back-translate to, then this result does not hold for the other ImpCon heap and vice versa. In the context of the heap definition as in Definition 2.4 we have stated that result in Lemma 4.8 and used it to show Corollary 4.18.

Given that we could not prove that property, proving the backward direction of Corollary 4.18 turned out to be very fiddly. The backward direction of

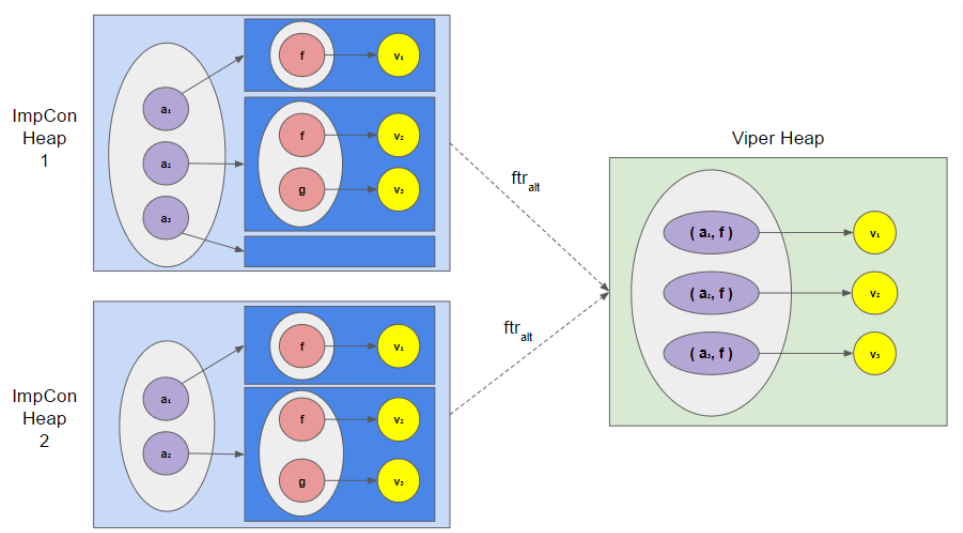


Figure 5.1: Multiple ImpCon heaps mapping to the same Viper heap during forward translation.

Corollary 4.18 states that for an Impcon assertion A and an Impcon state φ

$$\varphi \vDash A \iff ftr_{alt}(\varphi) \vDash [\![A]\!]$$

In the absence of a lemma analogous to Lemma 4.8 we tried to prove the result directly by structural induction. The fiddliness was mostly due to the case where A is a separating conjunction $A_1 * A_2$.

By the assumption in that case were are given two Viper states ω_1, ω_2 such that

$$\begin{aligned}
&\omega_1 \vDash [\![A_1]\!], \\
&\omega_2 \vDash [\![A_2]\!] \text{ and,} \\
&\omega_1 \bigoplus_V \omega_2 = ftr_{alt}(\varphi).
\end{aligned}$$

In order for the inductive assumption for this inductive case to be helpful we would have then needed two states φ_1, φ_2 such that

$$\begin{aligned}
&ftr_{alt}(\varphi_1) = \omega_1, \\
&ftr_{alt}(\varphi_2) = \omega_2 \text{ and,} \\
&\varphi = \varphi_1 \bigoplus_I \varphi_2.
\end{aligned}$$

But such states φ_1, φ_2 might not even exist, as ω_1, ω_2 might contain fractional permissions and hence not be forward-translated states.

Hence, by shifting permissions between states, we attempted to *rebalance* the permission heaps of states ω_1 and ω_2 into forward translated states ω_1' and

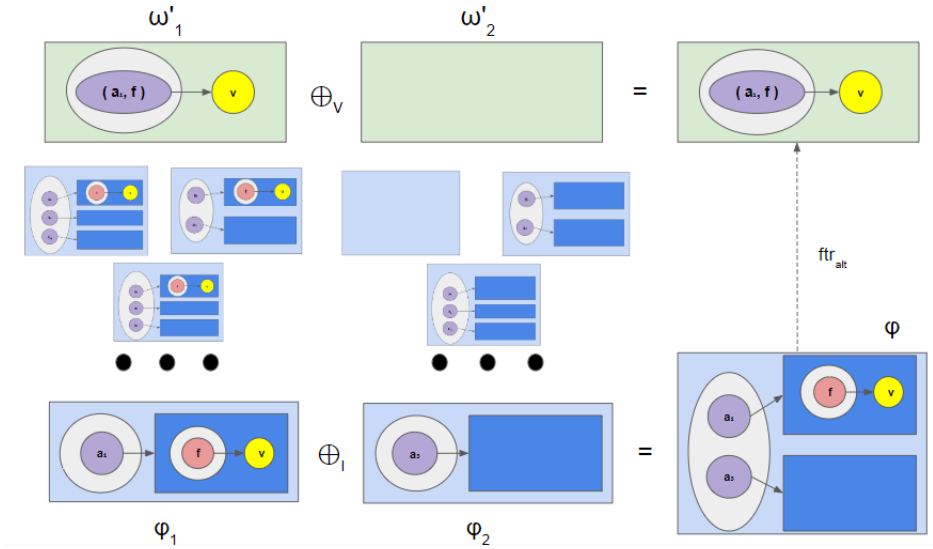


Figure 5.2: Choosing states φ_1, φ_2 given states ω_1' and ω_2'.

ω_2' such that

$$\omega_1' \vDash [\![A_1]\!],$$
$$\omega_2' \vDash [\![A_2]\!] \text{ and,}$$
$$\omega_1' \bigoplus_V \omega_2' = ftr_{alt}(\varphi).$$

Then we would attempt to find states φ_1, φ_2 such that $ftr_{alt}(\varphi_1) = \omega_1'$, $ftr_{alt}(\varphi_2) = \omega_2'$ and $\varphi = \varphi_1 \bigoplus_I \varphi_2$ to use the inductive hypothesis.

Showing the existence of such states φ_1, φ_2 given ω_1' and ω_2' is also non-trivial. Due to the mismatch between Viper states there can by numerous states φ_1, φ_2 such that $ftr_{alt}(\varphi_1) = \omega_1'$ and $ftr_{alt}(\varphi_2) = \omega_2'$, but only a fraction of choices will be such that $\varphi = \varphi_1 \bigoplus_I \varphi_2$. That is because in forward translation both addresses without allocated fields and non-existent addresses lead to non-existent heap locations. Figure 5.2 shows an example where we choose such states φ_1, φ_2 given states ω_1' and ω_2'.

As can be seen from this proof sketch the proof we have given in Section 4.2 using the original heap definition involving back translation was quite a lot cleaner. That is because, since it did not involve rebalancing states, which in Isabelle must be carefully defined, or choosing ImpCon states that add up to a given sum explicitly.

Switching the ImpCon heap definition made some proofs quite a bit easier and cleaner. That is mostly due to the state models being much better aligned for the state model we chose in the end. Because of that forward translation $ftr(\cdot)$ for the final state model is injective, so that backward translation can

be properly defined as we did in Definition 4.6. The drawback, on the other hand, is that we cannot express the case where an address has already been used but does not have any allocated fields. Hence such addresses might be reused by our alloc command. In the end, though, as ImpCon does not support deallocation, this fact did not matter that much to us in the scope of this project.

5.2 Defining an Explicit Frame to Show Soundness of Parallel Commands

Recall that in Section 4.3.3 we have proven the inductive case of parallel commands by applying the frame rule from separation logic. There we chose an ImpCon frame by leveraging the frame computed by the exhale command in Viper. In this section we sketch another proof that we also implemented in Isabelle, which uses an explicitly defined ImpCon frame instead. That is, will explicitly define the set of heap locations which should be part of our frame. We do so by computing the set of heap locations that are needed in the pre- and postcondition of the Hoare triple and defining our frame to contain all information on heap locations that do not lie in these computed sets.

Before we start the proof sketch, let us re-state some of the assumptions and derived facts from Section 4.3.3 up to the point where we used the frame rule, and what we were trying to prove.

We assumed for an ImpCon type context T, an ImpCon command $c \triangleq (\{Pre_1\}\ c_1\ \{Post_1\} \parallel \{Pre_2\}\ c_2\ \{Post_2\})$, ImpCon command c' and ImpCon states φ, φ' that

- c contains no assignments to local variables within parallel sections,
- $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$.

We obtained Viper states ω, ω_{Exh} such that

(E1) $ftr(\varphi) \vDash [\![Pre_1\ *\ Pre_2]\!]$,

(E2) $\omega \vDash [\![Pre_1\ *\ Pre_2]\!]$,

(E3) $ftr(\varphi) = \omega_{Exh} \bigoplus_V \omega$, and

We had also established that the Hoare triple

$$T\ \vDash \{Pre_1\ *\ Pre_2\}\, c_1 \parallel c_2\, \{Post_1\ *\ Post_2\}$$

holds.

What we want to show is that there exists some Viper state ω_{Inh} such that

(RI1) $ftr(\varphi') = \omega_{Exh} \bigoplus_V \omega_{Inh}$, and

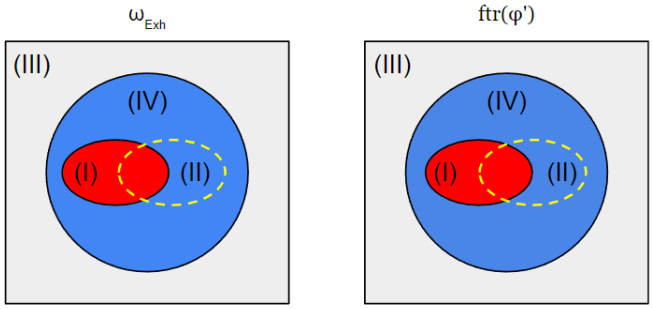


Figure 5.3: Dividing the heaps of ω_{Exh} and $ftr(\varphi')$ into regions.

(RI2) $\omega_{Inh} \vDash [\![Post_1]\!] * [\![Post_2]\!]$.

In this proof we will explicitly compare states $ftr(\varphi')$ and ω_{Exh} to find such a state ω_{Inh}. First we only consider (RI1). To find a Viper state ω_{Inh} such that (RI1) holds it is sufficient to show that $ftr(\varphi')$ is greater than or equal to ω_{Exh}. As c does not contain any local variable assignments we know from the fact that $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$, that φ and φ' have the same store. Through (E1) this implies that ω_{Exh} and $ftr(\varphi')$ have the same store. Hence, to see whether $ftr(\varphi')$ is greater than or equal to ω_{Exh} it suffices to compare their permission heaps.

We want to show that $ftr(\varphi')$ is greater than or equal to ω_{Exh}. That means we want that for all heap locations $ftr(\varphi')$ has at least as much permission as ω_{Exh} and for all heap locations that ω_{Exh} has non-zero permission to we need the heap values of $ftr(\varphi')$ and ω_{Exh} to have the same value. Our assumptions do not give us any direct relationship between ω_{Exh} and $ftr(\varphi')$. But both states have a relationship with state φ. Hence we will derive the relationship between ω_{Exh} and $ftr(\varphi')$ by considering state φ.

In Figure 5.3 we can see a visualisation of the permission heaps of $ftr(\varphi')$ and ω_{Exh}. Here the regions represent different sets of heap locations. These are listed in the following

- Region (I) represents the set of heap locations that are needed to satisfy $Pre_1 * Pre_2$.
- Region (II) represents the set of heap locations that are needed to satisfy $Post_1 * Post_2$.
- Region (III) represents the set of heap locations that do not exist in φ.
- Region (IV) represents the set of heap locations that do not lie in region (I) or region (III).

Note that these regions cover the entire set of heap locations.

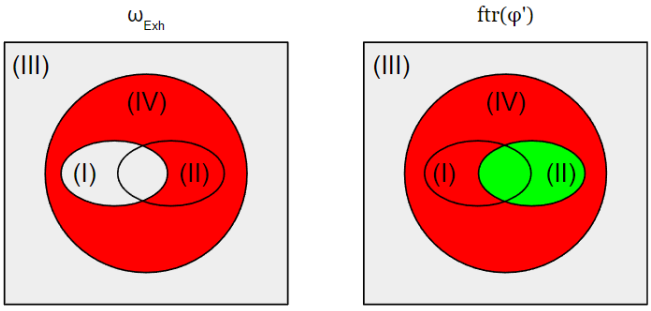


Figure 5.4: Derived information about the heaps of ω_{Exh} and $ftr(\varphi')$.

The reason we chose regions (I) to (III) is that for these sets of heap locations we can derive exactly what their permissions are in at least one of $ftr(\varphi')$ and ω_{Exh}.
For region (I) we can derive that we must have zero permission to these heap locations in ω_{Exh} using (E2) and (E3).
For region (II) we can derive that $ftr(\varphi')$ has full permission using our Hoare triple (and the fact that we do not allow local variable assignments in parallel regions).
For region (III) we can derive that both ω_{Exh} and $ftr(\varphi')$ have zero permission. For ω_{Exh} that is due to (E3). For $ftr(\varphi')$ the derivation also relies on the fact that we do not allow local variable assignments in parallel regions.

The fact that we do not allow local variable assignments is critical here. Local variable assignments also include alloc commands. Hence, as $(c,\ \varphi) \rightarrow_T^* (c',\ \varphi')$ there cannot be any heap locations that are allocated in φ' that are not allocated in φ already. Hence if a variable is not allocated in φ it cannot be allocated in φ' either. Thus, $ftr(\varphi')$ has no allocated heap locations in region (III).

What we derived so far is summarized in Figure 5.4. The grey regions we know we have zero permission for in the respective states and in the green region we know we have full permission in $ftr(\varphi')$. The red region we do not know anything about. Here we see that in regions (I) and (III) $ftr(\varphi')$ is greater than or equal to ω_{Exh}.

We still need to show $ftr(\varphi')$ is greater than or equal to ω_{Exh} in region (IV), which is the rest of the heap. This is where our frame comes into play. We use our frame to show that in region (IV) we must have full permission in $ftr(\varphi')$ and that if ω_{Exh} has non-zero permission then the heap values of $ftr(\varphi')$ and ω_{Exh} must have the same value, which is all we need to ensure that $ftr(\varphi')$ is greater than or equal to ω_{Exh} in region (IV).

The frame can be chosen to be exactly what we know to hold in region (IV) for φ. Then by applying the frame rule with this frame to our Hoare triple

we can derive that the heaps of φ and φ' are the same in region (IV). That can be shown to imply that $ftr(\varphi')$ is greater than or equal to ω_{Exh} in region (IV) using our assumptions. Then we obtain that $ftr(\varphi')$ is greater than or equal to ω_{Exh} on the entire heap. That means there exists a state ω_{Inh} that satisfies (RI1).

This proof relies on the fact that this frame can properly be defined. That is, we can define a function $heapLocs_\gamma(A)$ which for an ImpCon assertion A and an ImpCon state γ explicitly computes exactly the set of heap locations needed for A to hold in γ. Using that definition we define our frame $F(\gamma)$ to say that for all heap locations hl that are allocated in φ (i.e. do not lie in region (III)) that do not lie in the set $heapLocs_\varphi(Pre_1 * Pre_2)$ (i.e. do not lie in region (I)) γ's heap maps hl to the same value as φ does, which is exactly the frame we want to define.

Using our approach so far we can show there is a state ω_{Inh} that satisfies (RI1). Yet, we also need to show that there exists such an ω_{Inh} that also satisfies (RI2). For $\omega_{Inh} \vDash [\![Post_1]\!] * [\![Post_2]\!]$ to hold, ω_{Inh} must have full permission to all heap locations in $heapLocs_{\varphi'}(Post_1 * Post_2)$, which can be shown to correspond to region (II). That is because translated ImpCon assertions only contain access predicates requiring full permissions. In order for such a state ω_{Inh} to be joinable with ω_{Exh}, ω_{Exh} must have zero permission to all heap locations in $heapLocs_{\varphi'}(Post_1 * Post_2)$. But we can only be sure of that in regions (I) and (III).

This problem goes away, however, when we show that actually region (II) represents a subset of region (I). That is, using our heap-locs function we have

$$heapLocs_{\varphi'}(Post_1 * Post_2) \subseteq heapLocs_{\varphi}(Pre_1 * Pre_2).$$

Using that result one can then derive the existence of a state ω_{Inh} that satisfies both (RI1) and (RI2), as desired.

The reason the above result holds is again that we disallow local variable assignments and therefore also allocations in concurrent sections. Recall that

$$T \vDash \{Pre_1 * Pre_2\}\, c_1 \parallel c_2\, \{Post_1 * Post_2\}.$$

Hence, as $Post_1 * Post_2$ follows from $Pre_1 * Pre_2$ after execution of $c_1 \parallel c_2$, which cannot perform allocations, the postcondition cannot reference heap locations that are not referenced in the precondition of the Hoare triple. To illustrate this on an example, say P was the assertion $True$ and Q was the assertion $(x.f \mapsto \_)$ and were given that $T \vDash \{P\}\, c\, \{Q\}$. Say, we execute c from a state φ wich upholds T and whose heap is empty. This state φ satisfies P and so by the Hoare triple, after execution of c, Q must hold. So by executing c from φ we reach a state in which a heap location is allocated. Hence if c does not contain any allocations, it cannot be true that $T \vDash \{P\}\, c\, \{Q\}$.

That concludes our proof sketch.

The approach presented in this section required a much longer proof than the one shown in Section 4.3.3. That is because, by defining the frame explicitly as we did here, we implemented a computation that is quite similar to how the part of the state to be exhaled is chosen by Viper. When exhaling an assertion Viper computes a state ω such that the assertion holds in ω and ω is a part of the current state. To do this, Viper computes all states in which the assertion holds and chooses one that is part of the current state. Using the function $heapLocs_\gamma(A)$ we explicitly compute exactly the set of heap locations needed to satisfy assertion A and from that essentially derive the smallest state in which the assertion holds. Hence that is analogous to Viper computing ω except that we choose the smallest possible state. The proof presented in Section 4.3.3 leverages Viper's computation instead of using our own, causing the proof to be much shorter.

Furthermore, this proof is hard and in some cases even impossible to extend in case of changes to our assertion language. The current approach only works because of our currently very limited assertion language in ImpCon. If we add features to the assertion language such as logical disjunctions between points-to assertions, e. g. $(x.f \mapsto \_) \vee (y.f \mapsto \_)$, there would not even be a unique smallest state such that $(x.f \mapsto \_) \vee (y.f \mapsto \_)$ holds. In the example given we have that both states where exactly one of the corresponding heap locations (assuming they are distinct) exists would satisfy the assertion. Then, if this assertion was our precondition, no matter which set of heap locations we map our heap-locs function to, it cannot be guaranteed that all heap locations in this set are needed for the precondition to hold. In our example say we map to the set containing only the heap location corresponding to $x.f$, then the heap location corresponding to $x.f$, which lies in the set, would not really be needed to satisfy the precondition as the state might satisfy the precondition through existence of the heap location corresponding to $y.f$. Hence, if the heap location in the set is also not needed in the postcondition, it would have to be part of our frame. Hence we cannot deterministically define the frame using our heap-locs function.

Apart from extensions to the assertion language there is also the problem that this approach heavily builds on the fact that allocation is disallowed in parallel branches. Our proof in Section 4.3.3, on the other hand, only needs this assumption to ensure equality of stores, which is only necessary because we defined our encoding of the parallel command relying on this assumption.

To conclude, it is quite clear that the approach presented in Section 4.3.3 is the better one. It is shorter, cleaner, and easier to extend.

Chapter 6

An Executable for the ImpCon Front-end

In the previous chapter we have defined and proved the soundness of the translation function of our front-end once-and-forall, which was the main goal of this project. The syntax and semantics of ImpCon, the translation function and the formalisation of Viper we are translating to have all been defined in Isabelle. However, if one wishes to translate concrete ImpCon programs, then directly using the front-end translation within Isabelle would be quite cumbersome. For that reason we would instead like to have an executable version of our front-end, which parses an ImpCon program from a text file, passes the translated Viper program to the Viper verifier, and finally outputs the Viper verifier's verification result. This chapter will talk about how we created such an executable for our front-end using the translation function we defined and proved correct once-and-forall in the Isabelle Theorem Prover.

Our executable consists of three components as shown in Figure 6.1. One is the text file containing the ImCon program. The next one is the ImpCon AST, the Isabelle formalization of a Viper AST and the translation function all defined in Isabelle and the third component is the Scala Viper implementation [13] consisting of the intermediate verification language written in Scala and its verifier. To create our executable we need to connect these three components.

Remark 6.1 *To disambiguate the two, we will in this chapter refer to the Isabelle formalization of Viper as* IViper.

In Section 6.1 we explain how we extract the code needed for the translation function from Isabelle to be able to use it in our executable. Section 6.2 will then be about how we connect the extracted code to Viper. In Section 6.3 talk about the trust assumptions we have for our executable. Finally, in Section 6.4 we check whether our executable behaves as expected on some examples.

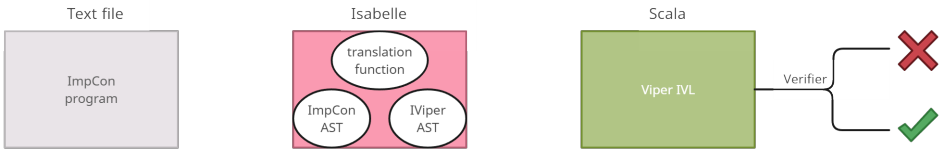


Figure 6.1: Components of the front-end.

6.1 The Extraction Process

As mentioned before, the translation function as well as the syntax of both ImpCon and IViper were all defined in Isabelle. Executing the translation function directly in Isabelle, however, is quite slow. Hence we make use of the fact that Isabelle provides a way to extract functions to executable code in a selection of other languages. These languages include, for example, Haskell, OCaml, Scala and SML. Since the Viper IVL was implemented in Scala and we need to connect the output of the translation function to the Viper implementation, we choose to extract the translation function to Scala[1].

Isabelle offers the functionality to extract code into Scala via the following command.

export_code [*functions to be extracted*] in Scala

As input for the above command we gave the translation function. Isabelle then extracts that function and all functions and definitions needed directly or indirectly by the translation function. Hence by supplying the translation function as a command we extract the definitions of the ImpCon and IViper syntax as well.

In some cases for code extraction from Isabelle to succeed one has to write alternative definitions and prove equivalence of these. Luckily, in our case code extraction of the translation function worked without further adjustments.

After code extraction we obtain a Scala file with Scala versions of the ImpCon and IViper ASTs and a translation function between both ASTs. The extracted module was almost ready to use right after extraction. Only some classes were given the same names in different capitalization, which caused compilation problems. Thus, after changing these names the module compiled without errors.

Having obtained the Scala module our situation is now as in Figure 6.2.

[1]It would have also been possible to use another language than Scala to extract to and then interact with the Viper implementation by passing a Viper text file, but it is more convenient to access the Viper API directly via Scala.

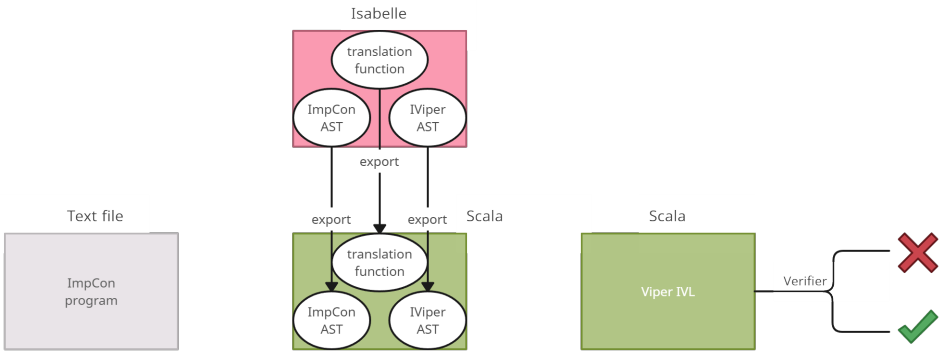


Figure 6.2: Exporting Isabelle code into Scala.

6.2 Connecting an ImpCon Text File to the Viper Implementation

Now that we have a version of the ImpCon and IViper ASTs and a translation function between both ASTs written in Scala we are in good shape to complete our front-end executable. First, we implement a parser so that we can convert ImpCon text files into Scala ImpCon ASTs. Then applying the translation function we obtain an IViper AST containing the main translation as well as a set of IViper ASTs giving the extra proof obligations. Eventually, to complete the front-end all that is left to do is to convert the IViper ASTs into ASTs of Viper's IVL, pass those to the verifier and return the verifier's output.

We have to write a translation function from IViper ASTs to Viper IVL ASTs. For most components of IViper ASTs there is a direct equivalent among the Viper IVL AST components. The only[2] exception is havocing local variables, which is currently not implemented in the Viper IVL. In the case of a havoc we therefore translate that component of the IViper AST to a Viper method call of the following abstract method, which fulfills the same purpose as havocing a reference variable.

```
method havocRef() returns (x : Ref)
```

Note that type-correct code translated from ImpCon only havocs variables of type reference as Havoc only comes up when translating Alloc-commands. Hence a method encoding the havocing of a reference variable is sufficient.

A slight annoyance when translating into the Viper IVL AST and also when writing the parser was about elementary datatypes. As both ImpCon and IViper use natural numbers (e.g., for variable names), strings (e.g., for field names), and integers (e.g. for integer literals) in the Isabelle implementation,

[2]The only exception among possible translated IViper ASTs

these data types are also needed in the Scala version. Isabelle explicitly defines them via their bit-representations. That leads to the Scala versions of the ASTs needing versions of these elementary datatypes in Scala that are also created as Scala classes during the extraction. For example chars are redefined as a class taking eight booleans as input. Therefore we had to write functions which convert usual chars to this version of chars and back in order to be able parse a written ImpCon program into the ImpCon AST and work with the IViper AST. The same goes for the other elementary datatypes mentioned.

In order to feed the Viper Scala ASTs obtained from the IViper AST to the verifier we need to convert all the ASTs into a single Viper program. We obviously include our havoc method. The body of the main method will be the Viper IVL AST resulting from the main translation and the extra proof obligations will be the bodies of other individual methods. None of these methods need any pre- or postconditions, but all the local variables have to be declared beforehand, as they are not declared in the translated ImpCon code itself. Hence we list all local variables with their types as return values for each of these methods using the type context we receive from parsing our ImpCon program.
Moreover as part of a Viper program all field names and their types have to be listed at the beginning of a Viper program. We can easily collect all field names from the IViper AST but as field types are dynamic in ImpCon it is a non-trivial task to infer their types. Thus, for the sake of simplicity we decided to for now only support fields to be of type Int and therefore list all occurring fields to be of type Int at the beginning of a Viper program. Through careful implementation this restriction of only accepting integer field types could theoretically be lifted.
Finally, through the described transformation we obtain the Viper program to verify.

The complete outline of the process is visualized in Figure 6.3. Note that we also included checks whether the obtained ImpCon AST fulfills all the extra assumptions we made in the soundness theorem (Theorem 4.4).

6.3 Trusted Code Base

The goal of this thesis is to create a Viper front-end which is formally guaranteed to be sound. While we have formally proved the translation between ImpCon and Viper correct in Isabelle, the complete executable itself contains various other components that must be trusted in order to ensure end-to-end soundness of the executable. The trust assumptions include the following:

- *CSL:* We based the proof in the case of the parallel command on the

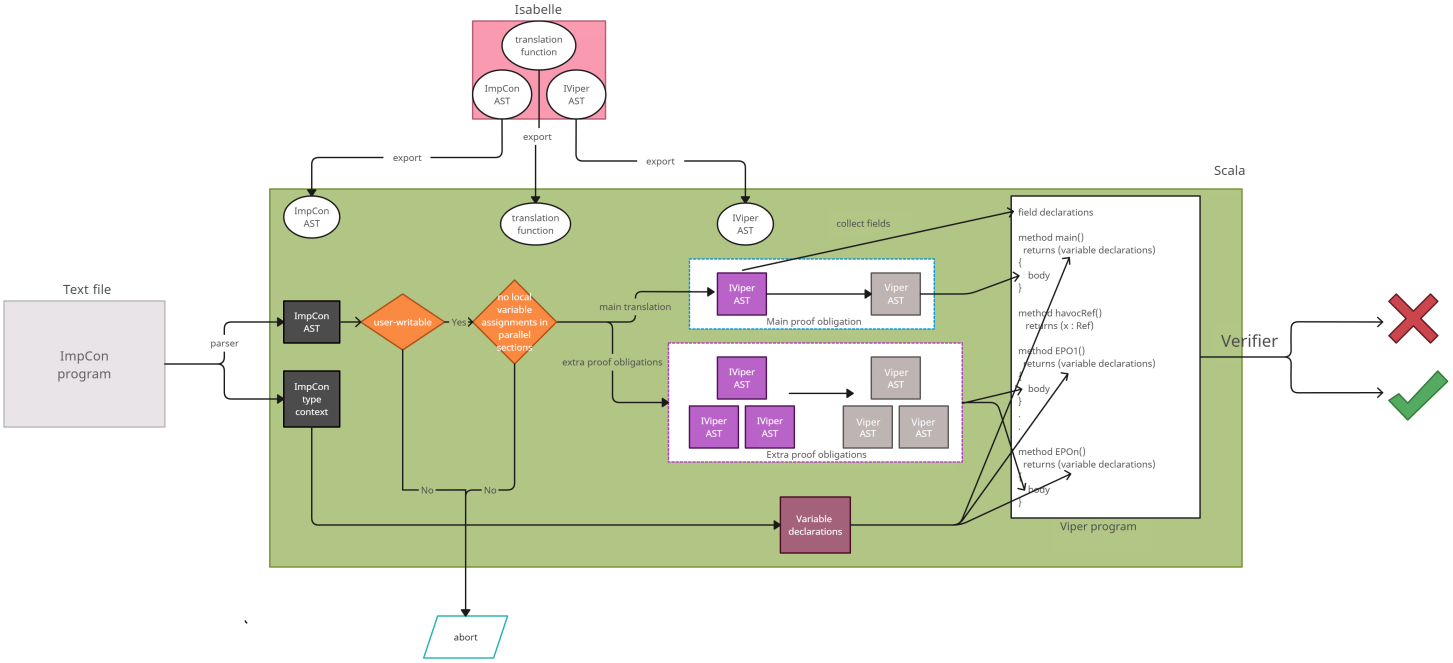


Figure 6.3: The front-end.

correctness of the CSL parallel rule and frame rule phrased in terms of the ImpCon semantics, which we have not proven formally in Isabelle. However it should be possible to prove them using the technique developed by Viktor Vafeiadis [12]. We have already proven the equivalence between our and Viktor Vafeiadis' definition of Hoare triples in the absence of resource invariants. Though here we also have to trust in our re-implementation of Viktor Vafeiadis' definition in Isabelle.

- *Isabelle:* We also trust in the correctness of Isabelle itself, as we base our trust in the soundness of the translation in the fact that we have proven the result in Isabelle.
- *Extraction into Scala:* Similarly we have to trust also in the extraction process of the Isabelle code into Scala.
- *ImpCon semantics:* Another basic component we are trusting here is the correctness of the ImpCon semantics, which we implemented ourselves.
- *IViper semantics:* We are also trusting that the IViper semantics are correct.
- *Viper:* We also trust that the Viper verifier correctly evaluates correctness of a given program written in the Viper IVL.
- *Translation from IViper to Viper in Scala:* We need to trust that we correctly

Filename	Description	Expected Outcome
testcode1.txt	variable swap	verification successful
testcode2.txt	concurrently setting two distinct allocated heap locations	verification successful
testcode3.txt	concurrently allocating and setting two distinct heap locations	arborting process due to attempt to write to local variable in parallel section
testcode4.txt	if-statement	verification successful
testcode5.txt	if-statement with final assertion being an unsatisfiable separating conjunction	verification fails due to unsatisfiable separating conjunction
testcode6.txt	parallel command with incorrect annotation	verification fails due to incorrect annotation
testcode7.txt	nested parallel command: concurrently setting three distinct allocated heap locations	verification successful

Table 6.1: Evaluation of the front-end executable on concrete examples.

implemented the translation of the IViper ASTs into a Viper program using Viper IVL ASTs.

- *Parser:* Lastly, we also have to trust that we correctly implemented the parser which converts an ImpCon text file into a Scala ImpCon AST and interpretes the type context.

Apart from the translation from IViper to Viper in Scala, the parser and the ImpCon semantics, all trusted components are part of established tools and are well-tested. The translation from IViper to Viper in Scala and the parser been developed by us ourselves and the ImpCon semantics are not well-tested. This we wish to alleviate in the next section.

6.4 Evaluation

As of the components of our executable the parser, the translation from IViper ASTs to Viper IVL ASTs, and the ImpCon semantics must be trusted and are not well-tested, we tried to gain confidence that the entire pipeline is correct by evaluating our executable on seven concrete examples. Table 6.1 presents our evaluation, in which the executable behaved as expected for all examples.

Input:

```
a : Int
b : Int
x : Ref

a := 5;
b := 8;
x := alloc(f,g);
If (a < b) {
  x.f := b;
  x.g := a;
}
else {
  x.f := a;
  x.g := b;
}
Assert (x.f ↦ 8) && (x.g ↦ 5)
```

Output:

```
field f16: Int
field f15: Int

method main()
  returns (x0: Int, x1: Int, x2: Ref)
{
  x0 := 5
  x1 := 8
  x2 := havocRef()
  inhale acc(x2.f16, write) &&
    (acc(x2.f15, write) && true)
  if (x0 < x1) {
    x2.f15 := x1
    x2.f16 := x0
  } else {
    x2.f15 := x0
    x2.f16 := x1
  }
  assert acc(x2.f15, write) && x2.f15 == 8 &&
    (acc(x2.f16, write) && x2.f16 == 5)
}

method havocRef()
  returns (havoc_Out: Ref)

Verification successful.
```

Listing 6: Running the executable on testcode4.txt.

Two of the examples we tested will be showcased here. For the other examples we refer the reader to Section A.1 of the appendix.

In Listing 6 you can see a simple conditional statement where we assign the smaller of two variables to the allocated field g of a reference variable x and the larger of the two to the allocated field f of the same reference variable x. Then, finally we assert the fields of x have been assigned the correct values.

Listing 7 shows an example of a nested parallel command. We allocate three fields $x.f$, $y.f$ and $y.g$, and then concurrently set all three fields and finally assert that the fields have been set correctly. As alloc commands in ImpCon guarantee that x and y are assigned distinct memory addresses this ImpCon program is correct. As expected, our front-end returns a successful verification result.

Input:

```
x : Ref
y : Ref

x := alloc(f);
y := alloc(f,g);
Parallel {
 {x.f ↦ _}    {(y.f ↦ _) * (y.g ↦ _)}
            ‖  Parallel {
            ‖    {y.f ↦ _}   {y.g ↦ _}
  x.f := 1  ‖      y.f := 2 ‖ y.g := 3
            ‖    {y.f ↦ _}   {y.g ↦ _}
            ‖  };
 {x.f ↦ 1}    {(y.f ↦ 2) * (y.g ↦ 3)}
};
Assert (x.f ↦ 1) && (y.f ↦ 2)
```

Output:

```
field f15: Int
field f16: Int

method main()
  returns (x0: Ref, x1: Ref)
{
  x0 := havocRef()
  inhale acc(x0.f15, write) && true
  x1 := havocRef()
  inhale acc(x1.f16, write) &&
    (acc(x1.f15, write) && true)
  exhale acc(x0.f15, write)
  exhale acc(x1.f15, write) && acc(x1.f16, write)
  inhale acc(x0.f15, write) && x0.f15 == 1
  inhale acc(x1.f15, write) && x1.f15 == 2 &&
    (acc(x1.f16, write) && x1.f16 == 3)
  assert acc(x0.f15, write) && x0.f15 == 1 &&
    (acc(x1.f15, write) && x1.f15 == 2 &&
    (acc(x1.f16, write) && x1.f16 == 3))
}

method havocRef()
  returns (havoc_Out: Ref)

method Extra_Proof_Obligation1()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x1.f15, write) && acc(x1.f16, write)
  exhale acc(x1.f15, write)
  exhale acc(x1.f16, write)
  inhale acc(x1.f15, write) && x1.f15 == 2
  inhale acc(x1.f16, write) && x1.f16 == 3
  exhale acc(x1.f15, write) && x1.f15 == 2 &&
    (acc(x1.f16, write) && x1.f16 == 3)
}

method Extra_Proof_Obligation2()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x0.f15, write)
  x0.f15 := 1
  exhale acc(x0.f15, write) && x0.f15 == 1
}

method Extra_Proof_Obligation3()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x1.f15, write)
  x1.f15 := 2
  exhale acc(x1.f15, write) && x1.f15 == 2
}

method Extra_Proof_Obligation4()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x1.f16, write)
  x1.f16 := 3
  exhale acc(x1.f16, write) && x1.f16 == 3
}

Verification successful.
```

Listing 7: Running the executable on testcode7.txt.

Chapter 7

Conclusion

In this thesis, we have implemented and formally verified a prototype Viper front-end for concurrent programs using the Isabelle theorem prover [10]. As opposed to other front-ends [4, 5, 6, 7, 8, 9], our prototype has been mechanically proven correct and formalised. Our work is a proof of concept that it is possible to reliably prove Viper front-ends sound once-and-forall. Therefore, future (and existing) Viper front-ends would benefit from adopting this approach.

We have designed a simple imperative programming language ImpCon which supports concurrency and heap operations, and defined a translation from ImpCon programs into Viper programs. We have shown that this translation is sound, in the following sense: If the Viper program obtained via the translation function verify then the ImpCon program used as input is correct with respect to its specification. The soundness of the translation we have proved formally once-and-forall in the Isabelle theorem prover, leveraging well-known results from concurrent separation logic [2, 12].

Eventually, using our sound translation function we have implemented an executable of our Viper front-end for ImpCon. We did so by exporting our translation function from Isabelle and connecting it to the Viper verifier. When evaluating our front-end executable on a few code samples we have seen that the output of the executable was as expected, giving us some confidence in the correctness of the implementation of the executable.

7.1 Future Work

ImpCon, as it is now, is a very simple language which apart from concurrency only offers conditional statements, basic heap operations and variable assignments. It does not support heap-dependent expressions causing us to have to work around this issue through additional operations. Furthermore, the

absence of fractional permissions means that we cannot distinguish between read and write permission, which implies that we cannot verify concurrent reads by splitting permissions between threads. Hence there are a lot of options to extend ImpCon, some of which we list here.

- **Extending the ImpCon language.** Currently there exist quite a few common features which are not reflected in the syntax of ImpCon. These features include for example while-loops and method calls. In these cases we could leverage the already existing framework for verifying while-loops and method calls in Viper in the translation. One could also extend ImpCon expressions to allow heap-dependent expressions, which would give the user more flexibility as they do not have to first save a field value to a local variable to be able to use it in an expression as they do now. Also, lifting our restriction that disallows local variable assignments in concurrently executed code would similarly give more flexibility to the user.
- **Extending the assertion language.** Adding support for heap-dependent expressions will not only extend the ImpCon language but also its assertion language. Another idea would be to add the capability to use heap-dependent functions within assertions, a feature which already exists in Viper. One could also add support for fractional permissions. By adding a permission mask to the ImpCon state and allowing fractional permissions we could distinguish between read access and write access by adding assertions expressing this. That would also enable us to be more permissive in parallel sections by allowing concurrent reads. Currently attempting to read from the same heap location concurrently causes verification to fail even if there is no actual data race due to none of the parallel sections attempting to write to that heap location.
  When adding a permission mask one could also go even further and add recursive predicates to the ImpCon assertion language. Recursive predicates are used in Viper in assertions to talk about permissions to larger objects such as a linked list. Thus, implementing them in ImpCon would allow us to also abstractly express and reason about some common data structures.
- **Generalizing the framework.** Currently our proofs are tailored to ImpCon's state model and semantics. It would be ideal if we could generalize our proof such that if we would like to prove a different front-end correct we could reuse our proof by only proving some key properties. That is, from these key properties we can derive that our proof holds for this different front-end as well. To give an example, say hypothetically our proof could be shown to hold for all front-ends whose state model can be mapped back and forth from the Viper state in a way that the properties we have shown for our state translation

hold. Then we would like to make it sufficient to prove these properties for another front-end in order to automatically apply our proof.

- **Adding support for more advanced concurrent separation logic features.** O'Hearn [2] and Vafeiadis [12] present some more advanced concepts of concurrent separation logic to reason about concurrent programs. These include *resource invariants* and *named critical regions*. Using such features one can make even more fine-grained statements about concurrent sections, making it possible to verify even more concurrent programs which might also contain features such as locks. Thus, enabling the use of advanced concurrent separation logic would be highly beneficial.

Bibliography

[1] J. Reynolds, "Separation logic: A logic for shared mutable data structures", in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[2] P. W. O'Hearn, "Resources, concurrency, and local reasoning", *Theoretical Computer Science*, vol. 375, no. 1, pp. 271–307, 2007, Festschrift for John C. Reynolds's 70th birthday, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2006.12.035. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030439750600925X.

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning", in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, B. Jobstmann and K. R. M. Leino, Eds., ser. LNCS, vol. 9583, Springer-Verlag, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2.

[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 *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., Cham: Springer International Publishing, 2021, pp. 367–379, ISBN: 978-3-030-81685-8.

[5] M. Eilers and P. Müller, "Nagini: A static verifier for Python", in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 596–603, ISBN: 978-3-319-96145-3.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360573. [Online]. Available: https://doi.org/10.1145/3360573.

[7] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors tool set: Verification of parallel and concurrent software", in *Integrated Formal Methods*, N. Polikarpova and S. Schneider, Eds., Cham: Springer International Publishing, 2017, pp. 102–110, ISBN: 978-3-319-66845-1.

[8] C. M. Gössi, "A formal semantics for viper", Master's thesis, Swiss Federal Institute of Technology Zurich, 2016. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Cyrill_Goessi_MA_report.pdf.

[9] F. A. Wolf, M. Schwerhoff, and P. Müller, "Concise outlines for a complex logic: A proof outline checker for TaDA", in *Formal Methods (FM)*, M. Huisman, C. S. Păsăreanu, and N. Zhan, Eds., ser. LNCS, vol. 13047, Springer, 2021, pp. 407–426. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-90870-6_22.

[10] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., *Isabelle/HOL*. Springer Berlin, Heidelberg, 2002, ISBN: 978-3-540-43376-7. DOI: https://doi.org/10.1007/3-540-45949-9.

[11] J. Boyland, "Checking interference with fractional permissions", in *Static Analysis*, R. Cousot, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 55–72, ISBN: 978-3-540-44898-3.

[12] V. Vafeiadis, "Concurrent separation logic and operational semantics", *MFPS 2011. Electronic Notes in Theoretical Computer Science*, vol. 276, pp. 335–351, 2011.

[13] Viperproject, *Viperproject/silver: Definition of the viper intermediate verification language.* [Online]. Available: https://github.com/viperproject/silver/tree/master (visited on 05/17/2023).

[14] T. Nipkow and G. Klein, *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014, ISBN: 3319105418.

Appendix A

Appendix

A.1 Evaluated Code Samples

In the following we list the code samples referred to in Table 6.1 that have not been listed in Section 6.4 together with the output returned by our front-end executable.

Input:

```
x : Int
y : Int
z : Int

x := 5;
y := 8;

z := x;
x := y;
y := z;

Assert (8 == x);
Assert (5 == y);
```

Output:

```
method main()
  returns (x0: Int, x1: Int, x2: Int)
{
    x0 := 5
    x1 := 8
    x2 := x0
    x0 := x1
    x1 := x2
    assert 8 == x0
    assert 5 == x1
}

method havocRef()
  returns (havoc_Out: Ref)

Verification successful.
```

Listing 8: Running the executable on testcode1.txt.

Input:

```
x : Ref
y : Ref

x := alloc(f);
y := alloc(f);
Parallel {
 {x.f ↦ _}    {y.f ↦ _}
  x.f := 2  ‖  y.f := 3
 {x.f ↦ 2}    {y.f ↦ 3}
};
Assert (x.f ↦ 2) && (y.f ↦ 3)
```

Output:

```
field f15: Int

method main()
  returns (x0: Ref, x1: Ref)
{
  x0 := havocRef()
  inhale acc(x0.f15, write) && true
  x1 := havocRef()
  inhale acc(x1.f15, write) && true
  exhale acc(x0.f15, write)
  exhale acc(x1.f15, write)
  inhale acc(x0.f15, write) && x0.f15 == 2
  inhale acc(x1.f15, write) && x1.f15 == 3
  assert acc(x0.f15, write) && x0.f15 == 2 &&
    (acc(x1.f15, write) && x1.f15 == 3)
}

method havocRef()
  returns (havoc_Out: Ref)

method Extra_Proof_Obligation1()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x1.f15, write)
  x1.f15 := 3
  exhale acc(x1.f15, write) && x1.f15 == 3
}

method Extra_Proof_Obligation2()
  returns (x0: Ref, x1: Ref)
{
  inhale acc(x0.f15, write)
  x0.f15 := 2
  exhale acc(x0.f15, write) && x0.f15 == 2
}

Verification successful.
```

Listing 9: Running the executable on testcode2.txt.

Input:

```
x : Ref
y : Ref

Parallel {
      {True}          {True}
  x := alloc(f);‖ y := alloc(f);
  x.f := 2      ‖ y.f := 3
    {x.f ↦ 2}       {y.f ↦ 3}
};
Assert (x.f ↦ 2) && (y.f ↦ 3)
```

Output:

```
Local Variable assignments in Parallel
Section detected! No guarantees on
correctness of verfication result!
Aborting...
```

Listing 10: Running the executable on testcode3.txt.

Input:

```
a : Int
b : Int
x : Ref

a := 5;
b := 8;
x := alloc(f,g);
If (a < b) {
  x.f := b;
  x.g := a;
}
else {
  x.f := a;
  x.g := b;
}
Assert (x.g ↦ _) && (x.g ↦ 5)
```

Output:

```
field f16: Int
field f15: Int

method main()
  returns (x0: Int, x1: Int, x2: Ref)
{
  x0 := 5
  x1 := 8
  x2 := havocRef()
  inhale acc(x2.f16, write) &&
    (acc(x2.f15, write) && true)
  if (x0 < x1) {
    x2.f15 := x1
    x2.f16 := x0
  } else {
    x2.f15 := x0
    x2.f16 := x1
  }
  assert acc(x2.f16, write) &&
    (acc(x2.f16, write) && x2.f16 == 5)
}

method havocRef()
  returns (havoc_Out: Ref)

Verification failed with 1 errors:
  [assert.failed:insufficient.permission]
  Assert might fail. There might be insufficient
  permission to access x2.f16
}
```

Listing 11: Running the executable on testcode5.txt.

Input:

```
x : Ref
y : Ref
a : Int
b : Int

x := alloc(f);
y := alloc(f);
x.f := a;
y.f := b;
Parallel {
{(x.f ↦ _) * (y.f ↦ _)}    {y.f ↦ _}
  x := alloc(f);    ‖ y := alloc(f);
  x.f := 2          ‖ y.f := 3
  y.f := 5          ‖
{(x.f ↦ 2) * (y.f ↦ 5)}    {y.f ↦ 3}
};
a := x.f;
b := y.f;
Assert (a < b);
```

Output:

```
field f15: Int

method main()
  returns (x0: Ref, x1: Ref, x2: Int, x3: Int)
{
  x0 := havocRef()
  inhale acc(x0.f15, write) && true
  x1 := havocRef()
  inhale acc(x1.f15, write) && true
  x0.f15 := x2
  x1.f15 := x3
  exhale acc(x0.f15, write) && acc(x1.f15, write)
  exhale acc(x1.f15, write)
  inhale acc(x0.f15, write) && x0.f15 == 2 &&
    (acc(x1.f15, write) && x1.f15 == 5)
  inhale acc(x1.f15, write) && x1.f15 == 3
  x2 := x0.f15
  x3 := x1.f15
  assert x2 < x3
}

method havocRef()
  returns (havoc_Out: Ref)

method Extra_Proof_Obligation1()
  returns (x0: Ref, x1: Ref, x2: Int, x3: Int)
{
  inhale acc(x1.f15, write)
  x1.f15 := 3
  exhale acc(x1.f15, write) && x1.f15 == 3
}
method Extra_Proof_Obligation2()
  returns (x0: Ref, x1: Ref, x2: Int, x3: Int)
{
  inhale acc(x0.f15, write) && acc(x1.f15, write)
  x0.f15 := 2
  x1.f15 := 5
  exhale acc(x0.f15, write) && x0.f15 == 2 &&
    (acc(x1.f15, write) && x1.f15 == 5)
}

Verification failed with 1 errors:
  [exhale.failed:insufficient.permission]
  Exhale might fail. There might be insufficient
  permission to access x1.f15
```

Listing 12: Running the executable on testcode6.txt

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

A Formally Verified Automatic Verifier for Concurrent Programs

Authored by (in block letters):

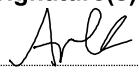
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Arlt	Ellen Joelle

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Zürich, 17.05.2023	

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