Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Formal Semantics for Viper

Master Thesis

Cyrill Martin Gössi

November 24, 2016

Advisors: Dr. Alexander J. Summers and Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science, ETH Zurich

Abstract

The Verification Infrastructure for Permission-based Reasoning (Viper) [1] is a suite of tools developed by the Chair of Programming Methodology research group at ETH Zurich[1]. Viper includes an intermediate language which is based on a flexible permission model and aims to allow for simple encodings of permission-based reasoning techniques implemented in front-end tools such as Chalice [2]. As of today, the intermediate language of Viper lacks a formal semantics. Thus, encodings of front-end languages into Viper cannot formally be reasoned about and the verification of a program encoded into Viper cannot rigorously be argued to be sound with respect to the semantics of the front-end tool language.

This thesis presents the first formal semantics for a chosen subset of Viper. Moreover, an encoding $[\![\_]\!]$ of a subset of Chalice into Viper is defined and proven sound: if an encoded Chalice program $[\![p]\!]$ verifies with respect to the semantics of Viper then p verifies with respect to the semantics of Chalice.

[1] pm.inf.ethz.ch/research/viper.html. Last accessed: November 24, 2016

Acknowledgements

First and foremost, I would like to thank my primary supervisor Dr. Alexander J. Summers for his great support throughout the whole duration of the thesis. His ingenious way of thinking about complex problems was inspirational to witness and his dedication to the role as a supervisor a privilege to have when conducting a final thesis. Thank you Alex for all your efforts. Then I would like to thank Prof. Dr. Peter Müller for having given me the opportunity to carry out this interesting thesis project within the Programming Methodology group. At last, I also sincerely acknowledge the efforts of my proofreaders Fabian, Felix, Valentin and Marco, even though I'm sure some of you didn't actually read the proofs.

Contents

Chapter 1

Introduction

Separation logic [3][4] and implicit dynamic frames [5] are popular permission logics used for the specification and verification of concurrent programs. Automated front-end verifiers, such as the permission logic based Chalice [2], translate a program together with its specification into an intermediate language, such as Boogie [6] or Why [7], where it ultimately is encoded as a logical formula and then solved by a theorem prover, such as Z3 [8]. Boogie or Why, however, have no direct support for permissions and the translation of a permission logic based front-end into one of these intermediate languages is involved.

The Verification Infrastructure for Permission-based Reasoning (Viper) [1], developed by the Chair of Programming Methodology research group at ETH Zurich[1], contains an intermediate language, also called Viper, which supports many high-level programming language features, such as method-calls and loops, and is based on a flexible permission model. As such, it allows for simple encodings of permission-based reasoning techniques and is targeted by multiple front-end tools such as Chalice.

To mathematically reason about the encoding of a front-end language into the Viper intermediate language requires the languages involved to have a formal semantics. As of today, Viper has no formal semantics. Thus, any encoding into Viper cannot be mathematically reasoned about and any verification in Viper cannot be proven sound with respect to the semantics of the front-end language.

This thesis presents the first formal semantics for a substantial subset of Viper. With this, it lays the foundation for a more rigorous treatment of the verification process within the Viper infrastructure. Moreover, an encoding of a subset of Chalice into Viper is defined and proven sound. This soundness proof demonstrates that whenever a Chalice program is translated into

[1] pm.inf.ethz.ch/research/viper.html. Last accessed: November 24, 2016

Viper and the translated program verifies within Viper, then the original Chalice program verifies with respect to the semantics of Chalice.

The structure of this thesis is as follows: chapter 2 presents the abstract syntax of the chosen subset of Viper and presents the first formal semantics for Viper on this subset. Chapter 3 presents the abstract syntax of a chosen Chalice subset as well as an extension of a pre-existing semantics for Chalice first defined in [9]. Chapter 4 defines an encoding of the chosen Chalice subset into Viper and chapter 5 proves this encoding to be sound with respect to the formally defined semantics of Viper and Chalice.

Chapter 2

Viper

Contents

2.1 Syntax

This section presents an abstract syntax of a substantial subset of the Viper intermediate language [1], short Viper, and it is this subset for which the subsequent section 2.2 will define a semantics for. The concrete syntax of Viper programs is given in appendix C. Where a concrete syntax is concerned with defining which strings of characters constitute a valid program, an abstract syntax *abstracts* away details related to parsing and lexing of character strings and focuses on representing the syntactic core domains of a program. The core domains relevant to present a semantics for Viper are *expressions*, *assertions*, *statements*, *functions*, *methods* and *predicates*.

These domains are presented using an Extended-Backus-Naur Form (EBNF)-like notation where a *vertical bar* "|" (as in "$a \mid b$") represents a choice (between a and b) and an *overline* (as in "$\overline{x_i}$") represents a finite repetition of elements ("$x_1, ..., x_n$").

Definition 2.1 *(Syntactic domain of expressions)* The syntactic domain of *expressions* is denoted by E and has metavariable e associated. The domain of *expressions* is defined as follows.

$$
\begin{aligned}
E \coloneqq\ & \mathit{null} \mid b \mid n \mid x \mid x.f \mid \text{-}\, e \mid e + e \mid e \,/\, e \mid \neg\, e \mid e = e \mid \mathit{fun}(\bar{e}) \mid \\
& \textbf{unfolding}\ \textbf{acc}(p(\bar{e}))\ \textbf{in}\ e \mid \textbf{perm}(x.f) \mid \textbf{perm}(p(\bar{e})) \mid \textbf{old}(e) \mid \\
& \textbf{old}[l](e) \mid \textbf{applying}\ a \mathrel{-\!\!*} a\ \textbf{in}\ e
\end{aligned}
$$

Above, b is the metavariable ranging over the syntactic domain of booleans B and n ranges over numerals N. Furthermore, x identifies a *variable* and is the metavariable associated with the syntactic domain of variable identifiers *Var*. Similarly, f, *fun*, p and l are associated with the domains F, *Fun*, P and *Label* and identify *fields*, *functions*, *predicates* and *labels*. At last, a is associated with the domain of *assertions* A, which is defined below.

Note that Viper knows many more unary operators and binary operators applicable to expressions besides the once listed above. However, the operators listed above suffice to subsequently demonstrate the semantics of Viper. The full list of operators available in Viper can be found in appendix C.

Definition 2.2 *(Syntactic domain of assertions)* The syntactic domain of *assertions* is denoted by A and has metavariable a associated. The domain of *assertions* is defined as follows.

$$
\begin{aligned}
A \coloneqq\ & e \mid \textbf{acc}(x.f, q) \mid \textbf{acc}(p(\bar{e}), q) \mid e \rightarrow a \mid a\ \&\&\ a \mid a \mathrel{-\!\!*} a \mid \\
& \textbf{forall}\ x : t\ ::e \mid \textbf{exists}\ x : t\ ::e \mid [a, a] \mid \textbf{forperm}[\bar{f}]\ x :: a
\end{aligned}
$$

Above, q is the metavariable ranging over the syntactic domain of *permissions* Q and t ranges over the domain of *types* T.

Definition 2.3 *(Syntactic domain of statements)* The syntactic domain of *statements* is denoted by S and has metavariable s associated. The domain of *statements* is defined as follows.

$$\begin{aligned} S \coloneqq\ & \varepsilon \mid \textbf{var}\ x : t \mid x := \textbf{new}(\bar{f}) \mid x := e \mid x.f := x \mid \overline{x} := m(\overline{e}) \mid \textbf{label}\ l \mid \\ & \textbf{assert}\ a \mid \textbf{assume}\ a \mid \textbf{inhale}\ a \mid \textbf{exhale}\ a \mid \textbf{fold}\ \textbf{acc}(p(\overline{e}), q) \mid \\ & \textbf{unfold}\ \textbf{acc}(p(\overline{e}), q) \mid \textbf{apply}\ a \mathrel{-\!\!*} a \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \\ & \textbf{while}\ e\ \textbf{invariant}\ a\ \textbf{do}\ s \mid s; s \end{aligned}$$

Above, ε denotes the *empty* statement and m identifies a *method* and is the metavariable associated with the syntactic domain of method identifiers *Meth*.

Definition 2.4 *(Syntactic domain of fields)* The syntactic domain of *fields* is denoted by *Field* and has metavariable *field* associated. The domain of *fields* is defined as follows.

$$Field \coloneqq \textbf{field}\ f : t$$

Definition 2.5 *(Syntactic domain of domains)* The syntactic domain of *domains* is denoted by *Dom* and has metavariable *dom* associated. The domain of *domains* is defined as follows.

$$Dom \coloneqq \textbf{domain}\ dom[\bar{t}]\ \{\ \overline{\textbf{function}\ fun(\overline{x : t}) : t}\ \ \overline{\textbf{axiom}\ ax\ \{\ a\ \}}\ \}$$

Definition 2.6 *(Syntactic domain of functions)* The syntactic domain of *functions* is denoted by *Fun* and has metavariable *fun* associated. The domain of *functions* is defined as follows.

$$Fun \coloneqq \textbf{function}\ fun(\overline{x : t}) : t\ \ \overline{\textbf{requires}\ a}\ \ \overline{\textbf{ensures}\ a}\ \{\ e\ \}$$

Definition 2.7 *(Syntactic domain of predicates)* The syntactic domain of *predicates* is denoted by P and has metavariable p associated. The domain of *predicates* is defined as follows.

$$P \coloneqq \textbf{predicate}\ p(\overline{x : t})\ \{\ a\ \}$$

Definition 2.8 *(Syntactic domain of methods)* The syntactic domain of *methods* is denoted by *Meth* and has metavariable *meth* associated. The domain of *methods* is defined as follows.

$$Meth \coloneqq \textbf{method}\ m(\overline{x : t})\ \textbf{returns}\ (\overline{x : t})\ \overline{\textbf{requires}\ a}\ \ \overline{\textbf{ensures}\ a}\ \{\ s\ \}$$

Definition 2.9 *(Syntactic domain of declarations)* The syntactic domain of *declarations* is denoted by *Decl* and has metavariable *decl* associated. The domain of *declarations* is defined as follows.

$$Decl \coloneqq dom \mid fun \mid p \mid meth$$

Definition 2.10 *(Syntactic domain of Viper programs)* The syntactic domain of *Viper programs* is denoted by *Prog* and has metavariable *prog* associated. The domain of *Viper programs* is defined as follows.

$Prog \coloneqq \overline{decl}$

2.2 Semantics

2.2.1 Preliminary

Unless otherwise specified, all functions defined in this preliminary are assumed to be total.

Definition 2.11 *(Semantic domain of Booleans)* The semantic domain of *Booleans* is denoted by $\mathbf{B}$, has metavariable $\mathbf{b}$ associated and is identified with the set of mathematical booleans. The domain of *Booleans* is given as: $\mathbf{B} = \{\, \text{False}, \text{True} \,\}$. Furthermore, a function mapping the syntactic *true* to the semantic True and mapping the syntactic *false* to the semantic False is assumed. The image of b under this function is subsequently denoted as $\mathbf{B}_b$.

Definition 2.12 *(Semantic domain of Integers)* The semantic domain of *Integers* is denoted by $\mathbf{Z}$, has metavariable $\mathbf{z}$ associated and is identified with the set of mathematical integers. A function mapping syntactic numerals n to appropriate integers $\mathbf{z}$ is assumed. The image of n under this function is subsequently denoted as $\mathbf{Z}_n$.

Definition 2.13 *(Semantic domain of objects)* Without explicit definition, an infinite semantic domain of *objects* is assumed. The domain of *objects* is denoted by $\mathbf{O}$ and has metavariable $\mathbf{o}$ associated. Parts of the subsequently presented semantics require to obtain new objects which are unique with respect to a given trace λ. For this, a function is assumed, which, given a trace λ, computes the set of all objects used within λ. A call to this function will be denoted as $\mathbf{O}(\lambda)$.

Definition 2.14 *(Semantic domain of permissions)* The semantic domain of *permissions* is denoted by $\mathbf{Q}$, has metavariable $\mathbf{q}$ associated and is identified with the set of non-negative mathematical rationals, i.e. $\mathbf{q} \geq 0$. Addition of permissions, for example, is then given by the addition over mathematical rationals. Furthermore, a function mapping syntactic permissions q to appropriate permissions $\mathbf{q}$ is assumed. The image of q under this function is subsequently denoted as $\mathbf{Q}_q$.

Definition 2.15 *(Semantic domain of errors)* An *error* is an assumed semantic value used to indicate an erroneous operation. The semantic domain of *errors* is denoted by $\mathbf{X}$, has metavariable $\mathbf{x}$ associated and is defined as follows: $\mathbf{X} = \{\, \mathbf{x}_{\text{div}}, \mathbf{x}_{\text{null}}, \mathbf{x}_{\text{ass}}, \mathbf{x}_{\text{perm}}, \mathbf{x}_{\text{inf}}, \mathbf{x}_{\text{wand}} \,\}$. The semantic values of $\mathbf{X}$

indicate the following: $\mathbf{x}_{\text{div}}$ indicates a division-by-zero, $\mathbf{x}_{\text{null}}$ indicates that the receiver of a field-lookup was null, $\mathbf{x}_{\text{ass}}$ indicates an assertion not to hold, $\mathbf{x}_{\text{perm}}$ indicates that a permission related error occurred, $\mathbf{x}_{\text{inf}}$ indicates an error related to an infinite unrolling of either a predicate or a function and $\mathbf{x}_{\text{wand}}$ indicates an error related to the usage of wands.

Definition 2.16 *(Semantic domain of values)* The semantic domain of *values* is denoted by $\mathbf{V}$ and has metavariable $\mathbf{v}$ associated. $\mathbf{V}$ is the *disjoint* union of the semantic domains of *Integers*, *Booleans*, *objects*, *permissions* and additionally contains the special value null, i.e. $\mathbf{V} = \mathbf{B} \cup \mathbf{Z} \cup \mathbf{O} \cup \mathbf{Q} \cup \{\,\text{null}\,\}$. Furthermore, the function $\mathbf{D} : (\mathbf{V} \mapsto \{\,\mathbf{B}, \mathbf{Z}, \mathbf{O} \cup \{\text{null}\}, \mathbf{Q}\,\}) \cup (T \mapsto \{\,\mathbf{B}, \mathbf{Z}, \mathbf{O} \cup \{\text{null}\}, \mathbf{Q}\,\}) \cup (F \mapsto \{\,\mathbf{B}, \mathbf{Z}, \mathbf{O} \cup \{\text{null}\}, \mathbf{Q}\,\})$ is assumed. $\mathbf{D}$ either maps semantic values $\mathbf{v}$ to the domain they originate from, denoted by $\mathbf{D}_{\mathbf{v}}$, maps syntactic types t to the corresponding semantic value sub-domain, denoted by $\mathbf{D}_t$, or maps syntactic field identifiers f to the semantic value sub-domain corresponding to the type of f, denoted by $\mathbf{D}_f$. Important example: $\mathbf{D}_{Ref} = \mathbf{O} \cup \{\,\text{null}\,\}$.

Definition 2.17 *(Semantic domain of predicate identifiers)* An infinite semantic domain of *predicate identifiers* is assumed, is denoted by $\mathbf{P}$ and has metavariable $\mathbf{p}$ associated. Identification of a syntactic predicate with a semantic counterpart is involved as the identification depends on the input a predicate is applied with: $p(0)$ and $p(1)$ should map to different semantic entities. Thus, the following parametrised family of injective functions is assumed: $\{\,\mathbf{P}_n \mid n \in \mathbb{N}\,\}$ where $\mathbf{P}_n : (P \times \mathbf{V}^n) \mapsto \mathbf{P}$. To continue the example, $p(0)$ is then identified with $\mathbf{p} = \mathbf{P}_1(p, \mathbf{Z}_0)$ and $p(1)$ is identified with $\mathbf{p}_1 = \mathbf{P}_1(p, \mathbf{Z}_1)$.

Definition 2.18 *(Semantic domain of wand identifiers)* An infinite semantic domain of *wand identifiers* is assumed, is denoted by $\mathbf{W}$ and has metavariable $\mathbf{w}$ associated. Identification of wands is similar to the identification of predicates and is based on extracting expressions from the wand to be identified and then evaluating these expressions to values. As an example, consider the wand $\mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.f, 1/1)$ && $\mathbf{acc}(y.f, 1/3)$. If the value of x is $\mathbf{o}$ and the value of y is $\mathbf{o}_1$, then the extracted tuple of values is $(\mathbf{o}, \mathbf{Q}_{1/2}, \mathbf{o}, \mathbf{Q}_{1/1}, \mathbf{o}_1, \mathbf{Q}_{1/3})$. However, extracting tuples of values itself is not sufficient to identify a wand as different wands can result in the same tuple of values: $\mathbf{acc}(x.f, 1/2)$ && $\mathbf{acc}(x.f, 1/1) \mathrel{-\!\!*} \mathbf{acc}(y.f, 1/3)$ is a different wand then the wand presented first but results in the same tuple of values. Thus, the identification of a wand also has to take the shape of the wand into account. Moreover, the left-hand side and right-hand side of a wand might contain ghost-expressions like **folding** and **unfolding**. To identify a wand, only the ghost-expression free assertions nested inside the left-hand side and right-hand side of the wand have to be considered during the extraction of expression values. An injective *shape function* mapping wands, based on their shapes, to unique identifiers in $\mathbb{N}$ is now assumed as

well as the following parametrised family of injective functions: $\{ \mathbf{W}_n \mid n \in \mathbb{N} \}$ where $\mathbf{W}_n : (\mathbb{N} \times \mathbf{V}^n) \mapsto \mathbf{W}$. A syntactic wand is then identified by first computing its shape, extracting the ghost-expression free assertions nested inside the left-hand side and right-hand side of the wand, extracting the expressions from these ghost-expression free sub-assertions, evaluating these expressions to values and then using the appropriate injective function from the family to get a value from $\mathbf{W}$. As an example, consider the two wands introduced before: $\mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.f, 1/1)$ && $\mathbf{acc}(y.f, 1/3)$ and $\mathbf{acc}(x.f, 1/2)$ && $\mathbf{acc}(x.f, 1/1) \mathrel{-\!\!*} \mathbf{acc}(y.f, 1/3)$. Note that the left-hand sides and right-hand sides of both wands are already ghost-expression free. Then assume the shape function to compute the shape identifier 1 for the first wand and the shape identifier 2 for the second wand. If the value of x is $\mathbf{o}$ and the value of y is $\mathbf{o}_1$ then the two wands are identified with $\mathbf{W}_6(1, \mathbf{o}, \mathbf{Q}_{1/2}, \mathbf{o}, \mathbf{Q}_{1/1}, \mathbf{o}_1, \mathbf{Q}_{1/3})$ and $\mathbf{W}_6(2, \mathbf{o}, \mathbf{Q}_{1/2}, \mathbf{o}, \mathbf{Q}_{1/1}, \mathbf{o}_1, \mathbf{Q}_{1/3})$. For a concise description of the subsequent semantics, the function $\mathbf{W}_n$ is now assumed as: $\mathbf{W}_n : A \times \Lambda \mapsto \mathbf{W} \cup \mathbf{X}$, where Λ is the domain of traces and defined in 2.28. In a trace λ, the first wand defined above is then identified with $\mathbf{W}_n(\mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.f, 1/1)$ && $\mathbf{acc}(y.f, 1/3), \lambda)$. $\mathbf{W}_n$ is thus assumed to compute the shape of the given wand, extract the ghost-expression free sub-assertions from the left-hand side and right-hand side, extract all expressions from these ghost-expression free sub-assertions, evaluate these expressions in the trace and then pick the appropriate injective function to map the shape identifier together with the tuple of values into a semantic wand identifier $\mathbf{w}$ or an error $\mathbf{x}$ in case any expression evaluation resulted in an error.

Definition 2.19 *(Semantic domain of heaps)* The semantic domain of *heaps* is denoted by $\mathbf{H}$ and is a domain of functions mapping tuples of semantic objects $\mathbf{o}$ and syntactic field identifiers f to semantic values $\mathbf{v}$. The metavariable associated with the domain of *heaps* is $\mathbf{h}$ and hence $\mathbf{h} : \mathbf{O} \times F \mapsto \mathbf{V}$.

Definition 2.20 *(Semantic domain of stores)* The semantic domain of *stores* is denoted by Σ, has metavariable σ associated and is a domain of functions mapping syntactic variable identifiers x to semantic values $\mathbf{v}$. Hence, $\sigma : \mathit{Var} \mapsto \mathbf{V}$.

Definition 2.21 *(Semantic domain of field-permission masks)* The semantic domain of *field-permission masks* is denoted by $\Pi_{\mathbf{F}}$, has metavariable $\pi^{\mathbf{F}}$ associated and is a domain of functions mapping tuples of semantic objects $\mathbf{o}$ and syntactic field identifiers f to semantic permissions $\mathbf{q}$. Hence, $\pi^{\mathbf{F}} : \mathbf{O} \times F \mapsto \mathbf{Q}$. Furthermore, the field-permission mask mapping all pairs of objects and fields to 0 is denoted by $\varnothing^{\mathbf{F}}$. Addition of field-permission masks is defined as follows: $(\pi^{\mathbf{F}} + \pi_1^{\mathbf{F}})(\mathbf{o}, f) = \pi^{\mathbf{F}}(\mathbf{o}, f) + \pi_1^{\mathbf{F}}(\mathbf{o}, f)$. Equality of field-permission masks is defined as follows: $\pi^{\mathbf{F}} = \pi_1^{\mathbf{F}} \Leftrightarrow \forall(\mathbf{o}, f).\, \pi^{\mathbf{F}}(\mathbf{o}, f) = \pi_1^{\mathbf{F}}(\mathbf{o}, f)$.

Definition 2.22 *(Semantic domain of predicate-permission masks)* The semantic domain of *predicate-permission masks* is denoted by $\Pi_{\mathbf{P}}$, has metavariable $\pi^{\mathbf{P}}$ associated and is a domain of functions mapping semantic predicate identifiers $\mathbf{p}$ to semantic permissions $\mathbf{q}$. Hence, $\pi^{\mathbf{P}} : \mathbf{P} \mapsto \mathbf{Q}$. Furthermore, the predicate-permission mask mapping all semantic predicate identifiers to 0 is denoted by $\varnothing^{\mathbf{P}}$. Addition of predicate-permission masks is defined as follows: $(\pi^{\mathbf{P}} + \pi_1^{\mathbf{P}})(\mathbf{p}) = \pi^{\mathbf{P}}(\mathbf{p}) + \pi_1^{\mathbf{P}}(\mathbf{p})$. Equality of predicate-permission masks is defined as follows: $\pi^{\mathbf{P}} = \pi_1^{\mathbf{P}} \Leftrightarrow \forall \mathbf{p}.\, \pi^{\mathbf{P}}(\mathbf{p}) = \pi_1^{\mathbf{P}}(\mathbf{p})$.

Definition 2.23 *(Semantic domain of wand-permission masks)* The semantic domain of *wand-permission masks* is denoted by $\Pi_{\mathbf{W}}$, has metavariable $\pi^{\mathbf{W}}$ associated and is a domain of functions mapping semantic wand identifiers $\mathbf{w}$ to semantic permissions $\mathbf{q}$. Hence, $\pi^{\mathbf{W}} : \mathbf{W} \mapsto \mathbf{Q}$. Furthermore, the wand-permission mask mapping all semantic wand identifiers to 0 is denoted by $\varnothing^{\mathbf{W}}$. Addition of wand-permission masks is defined as follows: $(\pi^{\mathbf{W}} + \pi_1^{\mathbf{W}})(\mathbf{w}) = \pi^{\mathbf{W}}(\mathbf{w}) + \pi_1^{\mathbf{W}}(\mathbf{w})$. Equality of wand-permission masks is defined as follows: $\pi^{\mathbf{W}} = \pi_1^{\mathbf{W}} \Leftrightarrow \forall \mathbf{w}.\, \pi^{\mathbf{W}}(\mathbf{w}) = \pi_1^{\mathbf{W}}(\mathbf{w})$.

Definition 2.24 *(Semantic domain of permission masks)* The semantic domain of *permission masks* is denoted by Π, has metavariable π associated and is a domain of triples over the domains of field-permission masks $\Pi_{\mathbf{F}}$, predicate-permission masks $\Pi_{\mathbf{P}}$ and wand-permission masks $\Pi_{\mathbf{W}}$. Hence, $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}})$. Furthermore, the permission mask consisting of $\varnothing^{\mathbf{F}}$, $\varnothing^{\mathbf{P}}$ and $\varnothing^{\mathbf{W}}$ is denoted by $\pi_\varnothing$, i.e. $\pi_\varnothing = (\varnothing^{\mathbf{F}}, \varnothing^{\mathbf{P}}, \varnothing^{\mathbf{W}})$. Addition of permission masks is defined as follows: $(\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) + (\pi_1^{\mathbf{F}}, \pi_1^{\mathbf{P}}, \pi_1^{\mathbf{W}}) = (\pi^{\mathbf{F}} + \pi_1^{\mathbf{F}}, \pi^{\mathbf{P}} + \pi_1^{\mathbf{P}}, \pi^{\mathbf{W}} + \pi_1^{\mathbf{W}})$. Equality of permission masks is defined as follows: $(\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) = (\pi_1^{\mathbf{F}}, \pi_1^{\mathbf{P}}, \pi_1^{\mathbf{W}}) \Leftrightarrow \pi^{\mathbf{F}} = \pi_1^{\mathbf{F}} \wedge \pi^{\mathbf{P}} = \pi_1^{\mathbf{P}} \wedge \pi^{\mathbf{W}} = \pi_1^{\mathbf{W}}$.

Definition 2.25 *(Heap equivalence on permission masks)* The judgment of a heap $\mathbf{h}$ and a heap $\mathbf{h}_1$ to be equal on permission mask $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}})$ is expressed through predicate $\mathbf{h} \overset{\pi}{\equiv} \mathbf{h}_1$ and defined as follows: $\mathbf{h} \overset{\pi}{\equiv} \mathbf{h}_1 \Leftrightarrow \forall(\mathbf{o}, f).\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \Rightarrow \mathbf{h}(\mathbf{o}, f) = \mathbf{h}_1(\mathbf{o}, f)$.

Definition 2.26 *(Evaluation context)* The evaluation strategy of the Viper semantics on a sequence of statements $s_1; s_2$ is to evaluate s_1 before s_2. To formally capture this strategy, the *evaluation context* C is introduced and defined by the following production rule.

$$\mathrm{C} := \bullet \mid \mathrm{C}; s$$

The result of plugging a statement s into context C is then given by the following equations.

$$\mathrm{C}[\varepsilon] = \begin{cases} \varepsilon & \text{if } \mathrm{C} \equiv \bullet \\ s & \text{if } \mathrm{C} \equiv \bullet; s \\ \mathrm{C}_1[\varepsilon]; s & \text{if } \mathrm{C} \equiv \mathrm{C}_1; s \end{cases}$$

$$\mathrm{C}[s] = \begin{cases} s & \text{if } \mathrm{C} \equiv \bullet \\ \mathrm{C}_1[s]; s_1 & \text{if } \mathrm{C} \equiv \mathrm{C}_1; s_1 \end{cases}$$

Definition 2.27 *(Semantic domain of configurations)* The semantic domain of *configurations* is denoted by Φ, has metavariable φ associated and is a domain of triples over the domains of heaps $\mathbf{H}$, stores Σ and permission masks Π. Hence, $\varphi = (\mathbf{h}, \sigma, \pi)$.

Definition 2.28 *(Semantic domain of traces)* The semantic domain of *traces* is denoted by Λ, has metavariable λ associated and is a domain of tuples of sequences of semantic configurations $\overline{\varphi}$ and a *label-mapping* function $\mathrm{L} : \mathit{Label} \mapsto \mathbb{N}$. Hence, $\lambda = (\overline{\varphi}, \mathrm{L})$. The purpose of L is to record the length of $\overline{\varphi}$ when λ reaches a label statement. This will allow for labelled-old expressions to be evaluated in the last configuration of $\overline{\varphi}$ if $\overline{\varphi}$ was the sequence of configurations when λ reached the label. The domain of traces Λ has a distinct sub-domain of *error traces* Λ_{Err} with Λ_{Err} containing one dedicated trace for each of the error values defined in 2.15: $\Lambda_{\mathrm{Err}} = \{ \lambda_{\mathrm{div}}, \lambda_{\mathrm{null}}, \lambda_{\mathrm{ass}}, \lambda_{\mathrm{perm}}, \lambda_{\mathrm{inf}}, \lambda_{\mathrm{wand}} \}$. Over the domain of traces Λ, the following operations are defined: *appending* a configuration φ_n to a trace $\lambda = ([\varphi_1, ..., \varphi_{n-1}], \mathrm{L})$ is denoted as $\lambda{:}\varphi_n$ and results in a trace $([\varphi_1, ..., \varphi_n], \mathrm{L})$, computing the *length* of a trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ is denoted as $|\lambda|$ and results in n, computing a sub-sequence of a trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ is denoted as $\lambda[i, j]$, for $i \leq j$, and results in a trace $([\varphi_i, ..., \varphi_j], \mathrm{L})$, *applying* the label-mapping function L of trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ at point l, i.e. $\mathrm{L}(l)$, is denoted as $\lambda(l)$, *updating* the label-mapping function L of trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ at point l to map to $n \in \mathbb{N}$, i.e. $\mathrm{L}[l \mapsto n]$, is denoted as $\lambda[l \mapsto n]$, *retrieving* the first configuration of trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ is denoted by $\mathrm{FIRST}(\lambda)$ and results in φ_1 and *retrieving* the last configuration of trace $\lambda = ([\varphi_1, ..., \varphi_n], \mathrm{L})$ is denoted by $\mathrm{LAST}(\lambda)$ and results in φ_n. Subsequently, traces will be used as the state with respect to which the semantics of Viper language constructs will be defined. As such, tuples $(\mathrm{C}[s], \lambda)$ will be considered and λ will be called a *complete trace* if $\mathrm{C}[s] = \varepsilon$ and λ will be called a *partial trace* if $\mathrm{C}[s] \neq \varepsilon$.

Definition 2.29 *(Local permission collection function)* The semantics will often require to collect all permissions explicitly contained in an assertion a. The semantic function $\mathbf{Q} : (A \times \Lambda \times \mathbf{B}) \mapsto \Pi \cup \mathbf{X}$, defined as algorithm 1 in appendix A, takes care of this and collects all permissions contained inside assertion a under a given trace λ. The boolean flag, given as the third argument to $\mathbf{Q}$, indicates whether the permissions inside assertion a should be collected *iso-recursively* or *equi-recursively* [9]: in case a contains a predicate p then an *iso-recursive* collection simply collects the permissions mentioned in connection with p whereas an *equi-recursive* collection fully unrolls p and collects all permissions contained within this full unrolling. An equi-recursive

collection of permission renders $\mathbf{Q}$ *partial* in case recursively defined predicates have an infinite unrolling. Note that algorithm 1 makes use of the expression evaluation function $[\![e, \lambda]\!]$ which is defined in 2.58. In order to have a semantics of total functions, $\mathbf{Q}$ is now totalised by defining the function $\mathbf{Q}^{\mathrm{T}}(\_, \_, \_) : (A \times \Lambda \times \mathbf{B}) \mapsto \Pi \cup \mathbf{X}$. This function computes, even in the presence of predicates with an infinite unrolling, either a permission mask reflecting the permissions contained in an assertion, or an error.

$$\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b}) = \begin{cases} \pi & \text{if } \mathbf{Q}(a, \lambda, \mathbf{b}) = \pi \\ \mathbf{x} & \text{if } \mathbf{Q}(a, \lambda, \mathbf{b}) = \mathbf{x} \\ \mathbf{x}_{\mathrm{inf}} & \text{if } \nexists \pi.\, \mathbf{Q}(a, \lambda, \mathbf{b}) = \pi \wedge \nexists \mathbf{x}.\, \mathbf{Q}(a, \lambda, \mathbf{b}) = \mathbf{x} \end{cases}$$

The following lemmas demonstrate that the totalised permission collection function $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ is deterministic.

Lemma 2.30 *(Totalised local permission collection is total)*
$\forall a, \lambda, \mathbf{b}. \exists \mathbf{u} \in \Pi \cup \mathbf{X}.\, \mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b}) = \mathbf{u}$

Proof Follows from totalising conditions in definition of $\mathbf{Q}^{\mathrm{T}}(\_, \_, \_)$. □

Lemma 2.31 *(Local permission collection is deterministic)*
$\forall a, \lambda, \mathbf{b}, \forall \mathbf{u}_1, \mathbf{u}_2 \in \Pi \cup \mathbf{X}.\, \mathbf{Q}(a, \lambda, \mathbf{b}) = \mathbf{u}_1 \ \wedge \ \mathbf{Q}(a, \lambda, \mathbf{b}) = \mathbf{u}_2 \ \Rightarrow \mathbf{u}_1 = \mathbf{u}_2$

Proof By structural induction on a and assuming the expression evaluation $[\![e, \lambda]\!]$ deterministic. □

Corollary 2.32 *(Totalised local permission collection is deterministic)*
$\forall a, \lambda, \mathbf{b}.\, \mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b}) = \mathbf{u}_1 \ \wedge \ \mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b}) = \mathbf{u}_2 \ \Rightarrow \mathbf{u}_1 = \mathbf{u}_2$

Proof Follows from lemmas 2.30 and 2.31. □

Definition 2.33 *(Global permission collection function)* The semantics will also require to collect all permissions to heap locations either directly given by a permission mask or indirectly given by a full unrolling of predicates to which a permission mask has access to. These permissions are collected by the semantic function $\mathbf{G}(\_, \_) : (\Pi \times \Lambda) \mapsto \Pi \cup \mathbf{X}$, which is defined as algorithm 2 in appendix A. In case of predicates with an infinite unrolling, $\mathbf{G}$ won't terminate. In order to have a semantics of total functions, the function $\mathbf{G}^{\mathrm{T}}(\_, \_) : (\Pi \times \Lambda) \mapsto \Pi \cup \mathbf{X}$ is now defined and computes, even in the presence of predicates with an infinite unrolling, either a permission mask, reflecting all directly and indirectly accessible heap locations, or an error.

$$\mathbf{G}^{\mathrm{T}}(\pi, \lambda) = \begin{cases} \pi_1 & \text{if } \mathbf{G}(\pi, \lambda) = \pi_1 \\ \mathbf{x} & \text{if } \mathbf{G}(\pi, \lambda) = \mathbf{x} \\ \mathbf{x}_{\mathrm{inf}} & \text{if } \nexists \pi_1.\, \mathbf{G}(\pi, \lambda) = \pi_1 \wedge \nexists \mathbf{x}.\, \mathbf{G}(\pi, \lambda) = \mathbf{x} \end{cases}$$

The following lemmas demonstrate that the totalised global permission collection function $\mathbf{G}^{\mathrm{T}}(\pi, \lambda)$ is deterministic.

Lemma 2.34 *(Totalised global permission collection is total)*
$\forall \pi, \lambda. \exists \mathbf{u} \in \Pi \cup \mathbf{X}. \mathbf{G}^{\mathrm{T}}(\pi, \lambda) = \mathbf{u}$

Proof Follows from totalising conditions in definition of $\mathbf{G}^{\mathrm{T}}(\_, \_)$. □

Lemma 2.35 *(Global permission collection is deterministic)*
$\forall \pi, \lambda, \forall \mathbf{u}_1, \mathbf{u}_2 \in \Pi \cup \mathbf{X}. \mathbf{G}(\pi, \lambda) = \mathbf{u}_1 \ \wedge \ \mathbf{G}(\pi, \lambda) = \mathbf{u}_2 \ \Rightarrow \ \mathbf{u}_1 = \mathbf{u}_2$

Proof Follows from a full unrolling of the premises and lemma 2.32. □

Corollary 2.36 *(Totalised global permission collection is deterministic)*
$\forall \pi, \lambda. \mathbf{G}^{\mathrm{T}}(\pi, \lambda) = \mathbf{u}_1 \ \wedge \ \mathbf{G}^{\mathrm{T}}(\pi, \lambda) = \mathbf{u}_2 \ \Rightarrow \ \mathbf{u}_1 = \mathbf{u}_2$

Proof Follows from lemmas 2.34 and 2.35. □

2.2.2 Evaluation of Expressions and Assertions

Assertion a, which can also be an expression e, evaluates in trace λ to $\mathbf{u} \in \mathbf{V} \cup \mathbf{X}$ if the subsequently inductively defined semantic function $\Downarrow$ maps a, given λ, to $\mathbf{u}$. If $\Downarrow$ maps a, given λ, to $\mathbf{u}$, the notation $a \Downarrow_\lambda \mathbf{u}$ will be used.

Definition 2.37 *(Evaluation in error-trace)* All assertions evaluate to False within an error-trace.

$$\frac{\lambda \in \Lambda_{\mathrm{Err}}}{a \Downarrow_\lambda \text{False}}$$

All subsequent evaluation rules assume an initial non-error trace.

Definition 2.38 *(Evaluation of null, booleans and numerals)* The syntactic null value *null*, booleans b and numerals n are unconditionally evaluated according to the following rules.

$$\frac{}{\mathit{null} \Downarrow_\lambda \text{null}} \qquad \frac{}{b \Downarrow_\lambda \mathbf{B}_b} \qquad \frac{}{n \Downarrow_\lambda \mathbf{Z}_n}$$

Definition 2.39 *(Evaluation of variable identifiers)* Variable x evaluates to the value contained for it in store σ of the last configuration of trace λ.

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda)}{x \Downarrow_\lambda \sigma(x)}$$

Definition 2.40 *(Evaluation of field-lookups)* Evaluating a field-lookup $x.f$ requires receiver x not to be null. If receiver x is null, the error value $\mathbf{x}_{\text{null}}$ is generated. Else, if no permission to field f of receiver x is available, the error value $\mathbf{x}_{\text{perm}}$ is generated. Otherwise, the semantic value stored in heap $\mathbf{h}$ for the field f of receiver x is returned.

$$\frac{(\mathbf{h}, \sigma, \pi) = \text{LAST}(\lambda) \quad \sigma(x) = \text{null}}{x.f \Downarrow_\lambda \mathbf{x}_{\text{null}}}$$

$$\frac{\begin{array}{c}(\mathbf{h}, \sigma, \pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \sigma(x) = \mathbf{o} \quad \pi^{\mathbf{F}}(\mathbf{o}, f) = 0\end{array}}{x.f \Downarrow_\lambda \mathbf{x}_{\text{perm}}}$$

$$\frac{\begin{array}{c}(\mathbf{h}, \sigma, \pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \sigma(x) = \mathbf{o} \quad \pi^{\mathbf{F}}(\mathbf{o}, f) > 0\end{array}}{x.f \Downarrow_\lambda \mathbf{h}(\mathbf{o}, f)}$$

Definition 2.41 *(Evaluation of unary operations)* Evaluation of unary operations is shown for the case of evaluating an unary negation operation. The remaining cases evaluate analogous.

$$\frac{e \Downarrow_\lambda \mathbf{x}}{\neg e \Downarrow_\lambda \mathbf{x}} \qquad \frac{e \Downarrow_\lambda \mathbf{b}}{\neg e \Downarrow_\lambda \neg \mathbf{b}}$$

Definition 2.42 *(Evaluation of binary operations)* Evaluation of binary operation expressions is shown for the case of evaluating a division operation. This case is special as it potentially generates a division-by-zero error. Evaluation of the remaining cases is analogous with the exception of not having to capture the division-by-zero error.

$$\frac{e \Downarrow_\lambda \mathbf{x}}{e \,/\, e_1 \Downarrow_\lambda \mathbf{x}} \qquad \frac{e \Downarrow_\lambda \mathbf{v} \quad e_1 \Downarrow_\lambda \mathbf{x}}{e \,/\, e_1 \Downarrow_\lambda \mathbf{x}} \qquad \frac{e \Downarrow_\lambda \mathbf{v} \quad e_1 \Downarrow_\lambda 0}{e \,/\, e_1 \Downarrow_\lambda \mathbf{x}_{\text{div}}} \qquad \frac{e \Downarrow_\lambda \mathbf{v} \quad e_1 \Downarrow_\lambda \mathbf{v}_1}{e \,/\, e_1 \Downarrow_\lambda \mathbf{v} \,/\, \mathbf{v}_1}$$

Definition 2.43 *(Evaluation of function calls)* The semantics of function-calls renders the expression evaluation function $\Downarrow$ partial in case the function to evaluate recurses infinitely. Definition 2.58 below will take care of this issue and establishes for Viper a semantics of total functions.

$$\frac{\exists i.\, e_i \Downarrow_\lambda \mathbf{x}}{fun(\overline{e_i}) \Downarrow_\lambda \mathbf{x}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathbf{x}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathrm{False}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}_{\mathrm{ass}}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathrm{True} \quad \mathrm{Body}(fun) \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathbf{x}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathrm{True} \quad \mathrm{Body}(fun) \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathbf{v} \quad a_1 = \mathrm{Post}(fun) \quad a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}][result \mapsto \mathbf{v}],\pi)} \mathbf{x}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathrm{True} \quad \mathrm{Body}(fun) \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathbf{v} \quad a_1 = \mathrm{Post}(fun) \quad a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}][result \mapsto \mathbf{v}],\pi)} \mathrm{False}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}_{\mathrm{ass}}}$$

$$\frac{\overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad a = \mathrm{Pre}(fun) \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathrm{True} \quad \mathrm{Body}(fun) \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}],\pi)} \mathbf{v} \quad a_1 = \mathrm{Post}(fun) \quad a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma[\overline{x_i \mapsto \mathbf{v}_i}][result \mapsto \mathbf{v}],\pi)} \mathrm{True}}{fun(\overline{e_i}) \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}}$$

Definition 2.44 *(Evaluation of predicate unfoldings)* An unfolding expression **unfolding acc**$(p(\overline{e_i}))$ **in** e first requires sufficient permission to unfold predicate p. If not enough permission is available, a permission-error results. Otherwise, the current trace is extended with all permissions implied by a *single* unrolling of predicate p and then continues to evaluate e with these additional permissions. Recall from 2.29, that $\mathbf{Q}^{\mathrm{T}}(\_,\_,\_)$ with the third argument set to True collects the permissions inside an assertion iso-recursively.

$$\frac{\exists i.\, e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}{\textbf{unfolding acc}(p(\overline{e_i}))\ \textbf{in}\ e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) < 1}{\textbf{unfolding acc}(p(\overline{e_i}))\ \textbf{in}\ e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}_{\mathrm{perm}}}$$

$$\frac{\begin{array}{c}\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \\ \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq 1 \quad \mathbf{x} = \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True})\end{array}}{\textbf{unfolding acc}(p(\overline{e_i}))\ \textbf{in}\ e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\begin{array}{c}\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{e_i \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}_i} \\ \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq 1 \quad \pi_1 = \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) \\ \pi_2 = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}[\mathbf{p} \mapsto \pi^{\mathbf{P}}(\mathbf{p}) - 1], \pi^{\mathbf{W}}) + \pi_1 \quad e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi_2)} \mathbf{u}\end{array}}{\textbf{unfolding acc}(p(\overline{e_i}))\ \textbf{in}\ e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{u}}$$

Definition 2.45 *(Evaluation of field-permission lookups)* Evaluation of a field-perm expression requires to retrieve the permission stored in the field-permission mask $\pi^{\mathbf{F}}$ contained in the head configuration of the current trace.

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \sigma(x) = \mathrm{null}}{\textbf{perm}(x.f) \Downarrow_{\lambda} \mathbf{x}_{\mathrm{null}}}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o}}{\textbf{perm}(x.f) \Downarrow_{\lambda} \pi^{\mathbf{F}}(\mathbf{o}, f)}$$

Definition 2.46 *(Evaluation of predicate-permission lookups)* Assume predicate p contained in expression **perm**$(p(\overline{e_i}))$ to be of arity $n \in \mathbb{N}$. First, the predicate is identified based on the evaluation of the n subexpressions e_i. Then

the permission value for the identified predicate stored in the predicate-permission mask $\pi^{\mathbf{P}}$ contained in the head configuration of the current trace is returned.

$$\frac{\exists\, i :\ e_i \Downarrow_\lambda \mathbf{x}}{\mathbf{perm}(p(\overline{e_i})) \Downarrow_\lambda \mathbf{x}}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{e_i \Downarrow_\lambda \mathbf{v}_i} \quad \mathbf{p} \ =\ \mathbf{P}_n(p, \overline{\mathbf{v}_i})}{\mathbf{perm}(p(\overline{e_i})) \Downarrow_\lambda \pi^{\mathbf{P}}(\mathbf{p})}$$

Definition 2.47 *(Evaluation of old value lookups)* Evaluating $\mathbf{old}(e)$ amounts to evaluating e in the first configuration of the given trace λ.

$$\frac{\lambda_1 = \ \lambda[1,1] \quad e \Downarrow_{\lambda_1} \mathbf{u}}{\mathbf{old}(e) \Downarrow_\lambda \mathbf{u}}$$

Definition 2.48 *(Evaluation of labelled-old lookups)* Evaluating $\mathbf{old}[l](e)$ amounts to evaluating e in the configuration that was last in λ when λ reached the label statement for l.

$$\frac{\lambda_1 = \ \lambda[1, \lambda(l)] \quad e \Downarrow_{\lambda_1} \mathbf{u}}{\mathbf{old}[l](e) \Downarrow_\lambda \mathbf{u}}$$

Definition 2.49 *(Evaluation of wand applications)* Evaluating **applying** $a \mathrel{-\!\!*} a_1$ **in** e first requires the current permission mask π to have permissions to the actual wand instance $a \mathrel{-\!\!*} a_1$. Moreover, a has to hold in the current trace $\lambda{:}(\mathbf{h}, \sigma, \pi)$. If so, $\lambda{:}(\mathbf{h}, \sigma, \pi)$ temporarily removes the permissions π_1 contained in a, adds the permissions π_2 contained in a_1, assumes with $\mathbf{h}_1$ all heap information contained in a_1 and then evaluates the expression e with these temporary modifications. Recall $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ to denote the totalised permission collection function as defined in 2.29. If $\mathbf{b}$ is True then permissions are collected *iso-recursively*. Below, $\mathbf{H}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ is an assumed *total* and *deterministic* function computing a list of heap locations and values explicitly mentioned within a. Moreover, the heap locations returned are locations to which λ has non-zero access. If $\mathbf{b}$ is True then the list is computed without unrolling of predicates contained in a. If $\mathbf{b}$ is False then the list is computed with unrolling of predicates contained in a.

$$\frac{\begin{array}{c} a \multimap a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True} \quad a \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True} \quad \mathbf{Q}^{\mathrm{T}}(a, \lambda{:}(\mathbf{h},\sigma,\pi), \text{True}) = \pi_1 \\ \mathbf{Q}^{\mathrm{T}}(a_1, \lambda{:}(\mathbf{h},\sigma,\pi), \text{True}) = \pi_2 \quad \overline{\mathbf{Q}^{\mathrm{T}}(a \multimap a_1, \lambda{:}(\mathbf{h},\sigma,\pi), \text{True}) = \pi_3} \\ \mathbf{H}^{\mathrm{T}}(a_1, \lambda{:}(\mathbf{h},\sigma,\pi), \text{True}) = \overline{(\mathbf{o}_i, f_i, \mathbf{v}_i)} \quad \mathbf{h}_1 = \mathbf{h}\overline{[(\mathbf{o}_i, f_i) \mapsto \mathbf{v}_i]} \\ \pi_4 = \pi - \pi_1 + \pi_2 - \pi_3 \quad e \Downarrow_{\lambda:(\mathbf{h}_1,\sigma,\pi_4)} \mathbf{v} \end{array}}{\textbf{applying } a \multimap a_1 \textbf{ in } e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{v}}$$

Definition 2.50 *(Evaluation of field-access assertions)* Evaluating assertion $\mathbf{acc}(x.f, q)$ requires receiver x not to be null. If x is null, a null-error is generated. Otherwise, the evaluation is True if the current field-permission mask $\pi^{\mathbf{F}}$ contains at least permission q to field f of receiver x, and False otherwise.

$$\frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \sigma(x) = \text{null}}{\mathbf{acc}(x.f,\, q) \Downarrow_\lambda \mathbf{x}_{\text{null}}}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q < 0}{\mathbf{acc}(x.f,\, q) \Downarrow_\lambda \mathbf{x}_{\text{perm}}}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q \geq 0}{\mathbf{acc}(x.f,\, q) \Downarrow_\lambda \pi^{\mathbf{F}}(\mathbf{o}, f) \geq \mathbf{Q}_q}$$

Definition 2.51 *(Evaluation of predicate-access assertions)* Evaluating assertion $\mathbf{acc}(p(\overline{e_i}),\, q)$ is similar to evaluating a field-access assertion except to first computing the actual semantic predicate $\mathbf{p}$.

$$\frac{\exists\, i : e_i \Downarrow_\lambda \mathbf{x}}{\mathbf{acc}(p(\overline{e_i}),\, q) \Downarrow_\lambda \mathbf{x}} \qquad \frac{\overline{e_i \Downarrow_\lambda \mathbf{v}_i} \quad \mathbf{Q}_q < 0}{\mathbf{acc}(p(\overline{e_i}),\, q) \Downarrow_\lambda \mathbf{x}_{\text{perm}}}$$

$$\frac{\begin{array}{c} (\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \overline{e_i \Downarrow_\lambda \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} \;=\; \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \end{array}}{\mathbf{acc}(p(\overline{e_i}),\, q) \Downarrow_\lambda \pi^{\mathbf{P}}(\mathbf{p}) \geq \mathbf{Q}_q}$$

Definition 2.52 *(Evaluation of conditional assertions)* Assertion $e \rightarrow a$ evaluates to True in case the conditional expression e evaluates to False. Otherwise, it evaluates to the value to which a evaluates.

$$\frac{e \Downarrow_\lambda \mathbf{x}}{e \rightarrow a \Downarrow_\lambda \mathbf{x}} \qquad \frac{e \Downarrow_\lambda \text{False}}{e \rightarrow a \Downarrow_\lambda \text{True}} \qquad \frac{e \Downarrow_\lambda \text{True} \quad a \Downarrow_\lambda \mathbf{u}}{e \rightarrow a \Downarrow_\lambda \mathbf{u}}$$

Definition 2.53 *(Evaluation of separating conjunction assertions)* Assertion a && a_1 can only evaluate to True if the current permission mask π can be split into π_1 and π_2 s.t. a evaluates to True with π_1 and a_1 evaluates to True with π_2. This semantics reflects the semantics of separating conjunctions as it was originally given in [4].

$$\frac{\exists \pi_1, \pi_2 : \pi = \pi_1 + \pi_2 \ \wedge \ a \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi_1)} \text{True} \ \wedge \ a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi_2)} \text{True}}{a \ \&\& \ a_1 \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True}}$$

If there is no such split of π, then the only requirement the semantics imposes is the assertion a && a_1 not to evaluate to True. Hence, an implementation is free to chose how to handle the complementary case and letting it either result in False or, if a sub-assertion is evaluated, possibly in some error $\mathbf{x}$. The semantics at the level of traces will then either result in λ_{ass}, in case the implementation handles the complementary case by letting it result in False, or it results in $\lambda_{\mathbf{x}}$. In any case, the verification of a program will fail.

Definition 2.54 *(Evaluation of magic-wand assertions)*

$$\frac{\mathbf{W}_n(a \mathrel{-\!\!*} a_1, \lambda) = \mathbf{x}}{a \mathrel{-\!\!*} a_1 \Downarrow_\lambda \mathbf{x}}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \mathbf{W}_n(a \mathrel{-\!\!*} a_1, \lambda) = \mathbf{w}}{a \mathrel{-\!\!*} a_1 \Downarrow_\lambda \pi^{\mathbf{W}}(\mathbf{w}) \geq 1}$$

Definition 2.55 *(Evaluation of all-quantified permission assertions)* An all-quantified permission assertion **forall** $x : t :: e$ requires e to evaluate to True if any value from the domain of type t substitutes for x inside e.

$$\frac{\exists \mathbf{v} \in \mathbf{D}_t . \, e[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{v}],\pi)} \mathbf{x}}{\textbf{forall} \ x : t :: e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\exists \mathbf{v} \in \mathbf{D}_t . \, e[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{v}],\pi)} \text{False}}{\textbf{forall} \ x : t :: e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{False}}$$

$$\frac{\forall \mathbf{v} \in \mathbf{D}_t . \, e[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{v}],\pi)} \text{True}}{\textbf{forall} \ x : t :: e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True}}$$

Definition 2.56 *(Evaluation of existentially-quantified permission assertions)* An existentially-quantified permission assertion **exists** $x : t :: e$ requires the domain of type t to have a value $\mathbf{v}$ s.t. e evaluates to True if $\mathbf{v}$ substitutes for x inside e.

$$\frac{\exists \mathbf{v} \in \mathbf{D}_t.\, e[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{v}],\pi)} \text{True}}{\textbf{exists } x : t :: e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True}}$$

$$\frac{\nexists \mathbf{v} \in \mathbf{D}_t.\, e[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{v}],\pi)} \text{True}}{\textbf{exists } x : t :: e \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{False}}$$

Definition 2.57 *(Evaluation of forperm assertions)* A forperm assertion **forperm**$[f]$ $x :: a$ requires assertion a, which is a function of x, to evaluate to True whenever x has non-zero access permissions to field f.

$$\frac{\begin{array}{c}\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \exists \mathbf{o}.\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \wedge a[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{o}],\pi)} \mathbf{x}\end{array}}{\textbf{forperm}[f]\ x :: a \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \mathbf{x}}$$

$$\frac{\begin{array}{c}\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \exists \mathbf{o}.\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \wedge a[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{o}],\pi)} \text{False}\end{array}}{\textbf{forperm}[f]\ x :: a \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{False}}$$

$$\frac{\begin{array}{c}\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \\ \forall \mathbf{o}.\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \Rightarrow a[x_1/x] \Downarrow_{\lambda:(\mathbf{h},\sigma[x_1 \mapsto \mathbf{o}],\pi)} \text{True}\end{array}}{\textbf{forperm}[f]\ x :: a \Downarrow_{\lambda:(\mathbf{h},\sigma,\pi)} \text{True}}$$

Definition 2.58 *(Value of assertions)* The assertion evaluation function $\Downarrow$ defined above is *partial* as a function-call evaluation won't terminate if the evaluated function recurses infinitely. In order to have a semantics of total functions, the function $[\![\_, \_]\!] : A \times \Lambda \mapsto \mathbf{V} \cup \mathbf{X}$ is now defined and computes, even in the presence of infinitely recursing functions, either a value of an assertion or an error.

$$[\![a, \lambda]\!] = \begin{cases} \mathbf{v} & \text{if } a \Downarrow_\lambda \mathbf{v} \\ \mathbf{x} & \text{if } a \Downarrow_\lambda \mathbf{x} \\ \mathbf{x}_{\text{inf}} & \text{if } \nexists \mathbf{v}.\, a \Downarrow_\lambda \mathbf{v} \wedge \nexists \mathbf{x}.\, a \Downarrow_\lambda \mathbf{x} \end{cases}$$

The following lemmas demonstrate that the totalised assertion evaluation function $[\![a, \lambda]\!]$ is deterministic.

Lemma 2.59 *(Totalised assertion evaluation is total)*
$\forall a, \lambda. \exists \mathbf{u} \in \mathbf{V} \cup \mathbf{X}. [\![a, \lambda]\!] = \mathbf{u}$

Proof Follows from totalising conditions in definition of $[\![\_, \_]\!]$. □

Lemma 2.60 *(Assertion evaluation is deterministic)*
$\forall a, \lambda, \forall \mathbf{u}_1, \mathbf{u}_2 \in \mathbf{V} \cup \mathbf{X}. a \Downarrow_\lambda \mathbf{u}_1 \ \wedge \ a \Downarrow_\lambda \mathbf{u}_2 \ \Rightarrow \ \mathbf{u}_1 = \mathbf{u}_2$

Proof By structural induction on a and assuming the totalised local permission collection $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ deterministic. □

Corollary 2.61 *(Totalised assertion evaluation is deterministic)*
$\forall a, \lambda. [\![a, \lambda]\!] = \mathbf{u}_1 \ \wedge \ [\![a, \lambda]\!] = \mathbf{u}_2 \ \Rightarrow \mathbf{u}_1 = \mathbf{u}_2$

Proof Follows from lemmas 2.59 and 2.60. □

2.2.3 Trace Semantics of Statements

The semantics of a statement s is defined as the change of a state represented by a trace λ. The semantics of statement s is thus defined with respect to a partial trace (s, λ) and its evolution into a partial trace (s_1, λ_1). The partial trace (s, λ) evolves into (s_1, λ_1) if the subsequently inductively defined binary semantic function $\rightsquigarrow$ maps (s, λ) to (s_1, λ_1). To denote that $\rightsquigarrow$ maps (s, λ) to (s_1, λ_1), the notation $(s, \lambda) \rightsquigarrow (s_1, \lambda_1)$ will be used. Note that due to the following lemma, no explicit semantic rule for sequences of statements will be required.

Lemma 2.62 *(Superfluous plugging composition)*
$\forall \mathrm{C}, \mathrm{C}_1, s. \exists \mathrm{C}_2. \mathrm{C}_1[\mathrm{C}[s]] = \mathrm{C}_2[s]$

Proof By induction on C_1. □

Definition 2.63 *(Unconditional error-trace propagation)* The semantics of any statement s is to unconditionally propagate an error-trace.

$$\frac{\lambda \in \Lambda_{\mathrm{Err}}}{(\mathrm{C}[s], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda)}$$

All subsequent rules of the trace semantics assume an initial non-error trace.

Definition 2.64 *(Trace semantics of variable declaration statements)* The semantics of statement **var** $x : t$ is to append to trace λ a new configuration where for x an arbitrary value from the domain of type t is assumed. Note that this semantics introduces non-determinism at the level of single traces and recall that $\mathbf{D}_{Ref} = \mathbf{O} \cup \{ \mathrm{null} \}$.

$$\frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \mathbf{v} \in \mathbf{D}_t}{(\mathrm{C}[\mathbf{var}\ x : t],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h},\sigma[x \mapsto \mathbf{v}],\pi))}$$

Definition 2.65 *(Trace semantics of new statements)* The semantics of statement $x := \mathbf{new}(\overline{f_i})$ is to let a trace λ assume for x an arbitrary object $\mathbf{o}$ which is not yet used within λ. Note that this semantics introduces non-determinism on the level of single traces. Furthermore, λ will subsequently have full permission to all fields $\overline{f_i}$.

$$\frac{\begin{array}{c}(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \quad \overline{\mathbf{v}_i \in \mathbf{D}_{f_i}} \\ \mathbf{o} \notin \mathbf{O}(\lambda) \quad \mathbf{h}_1 = \mathbf{h}[\overline{(\mathbf{o},f_i) \mapsto \mathbf{v}_i}] \quad \sigma' = \sigma[x \mapsto \mathbf{o}] \quad \pi_1^{\mathbf{F}} = \pi^{\mathbf{F}}[\overline{(\mathbf{o},f_i) \mapsto 1}]\end{array}}{(\mathrm{C}[x := \mathbf{new}(\overline{f_i})],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h}_1,\sigma',(\pi_1^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}})))}$$

Definition 2.66 *(Trace semantics of variable assignment statements)* The semantics of statement $x := e$ is to append to a trace λ a new configuration which is equal to the current last configuration of λ except to contain an updated store which maps x to the value to which e evaluates to.

$$\frac{[\![e,\lambda]\!] = \mathbf{x}}{(\mathrm{C}[x := e],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathbf{x}})} \qquad \frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad [\![e,\lambda]\!] = \mathbf{v}}{(\mathrm{C}[x := e],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h},\sigma[x \mapsto \mathbf{v}],\pi))}$$

Definition 2.67 *(Trace semantics of field assignment statements)* The semantics of statement $x.f := x_1$ requires the trace λ to have full permission to field f of receiver x. If not so, a permission-error is generated. Otherwise, and in case x is not null, the statement appends to λ a configuration which is equal to the current head configuration of λ except to contain a heap where field f of receiver x is updated based on x_1.

$$\frac{(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \sigma(x) = \mathrm{null}}{(\mathrm{C}[x.f := x_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathrm{null}})}$$

$$\frac{\begin{array}{c}(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \\ \sigma(x) = \mathbf{o} \quad \pi^{\mathbf{F}}(\mathbf{o},f) < 1\end{array}}{(\mathrm{C}[x.f := x_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathrm{perm}})}$$

$$\frac{\begin{array}{c}(\mathbf{h},\sigma,\pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \\ \sigma(x) = \mathbf{o} \quad \pi^{\mathbf{F}}(\mathbf{o},f) \geq 1 \quad \sigma(x_1) = \mathbf{v}\end{array}}{(\mathrm{C}[x.f := x_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h}[(\mathbf{o},f) \mapsto \mathbf{v}],\sigma,\pi))}$$

Definition 2.68 *(Trace semantics of method-call statements)* The semantics of a method-call $\overline{x_i'} := m(\overline{e_i})$ is given by an exhale of the precondition of m and an inhale of the postcondition of m in the same context C and the same trace λ. Thus, a method-call in Viper is essentially syntactic sugar.

$$\frac{\overline{a_j} = \mathrm{Pre}(m) \quad \overline{a_j'} = \mathrm{Post}(m) \quad \overline{r_i} = \mathrm{Ret}(m) \quad \overline{x_i} = \mathrm{Param}(m)}{(\mathrm{C}[\overline{x_i'} := m(\overline{e_i})], \lambda) \rightsquigarrow (\mathrm{C}[\overline{\textbf{exhale } a_j[\overline{e_i/x_i}]}; \ \overline{\textbf{inhale } a_j'[\overline{x_i'/r_i}]}], \lambda)}$$

Definition 2.69 *(Trace semantics of label statements)* The semantics of a label statement **label** l is to update the label-mapping function of trace λ to remember the current length of its sequence of configurations. This allows for a subsequent labelled-old expression to be able to retrieve the configuration which is currently the last configuration of λ.

$$\frac{\lambda_1 = \lambda[l \mapsto |\lambda|]}{(\mathrm{C}[\textbf{label } l], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_1)}$$

Definition 2.70 *(Trace semantics of assert statements)* The semantics of an assert statement **assert** a is to generate an error-trace in case a evaluates to False in the current trace λ. If a evaluates to True, the semantics is to continue in the evaluation context C without any further changes.

$$\frac{[\![a, \lambda]\!] = \mathrm{False}}{(\mathrm{C}[\textbf{assert } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{ass}})} \qquad \frac{[\![a, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{assert } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{[\![a, \lambda]\!] = \mathrm{True}}{(\mathrm{C}[\textbf{assert } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda)}$$

Definition 2.71 *(Trace semantics of assume statements)* The semantics of an assume statement **assume** a is similar to the semantics of an assert statement, except that no error-trace is generated in case a evaluates to False in the current trace λ.

$$\frac{[\![a, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{assume } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})} \qquad \frac{[\![a, \lambda]\!] = \mathrm{True}}{(\mathrm{C}[\textbf{assume } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda)}$$

Definition 2.72 *(Trace semantics of inhale statements)* The semantics of an inhale statement **inhale** a is to assume all expressions contained in a and to

extend the current trace λ with all permissions mentioned in accessibility sub-assertions of a. If a contains a predicate p then the current permission mask is extended with permissions to the *instance* of p. This reflects an *iso-recursive* treatment of predicates. Furthermore, the semantics of inhaling a conjunct of assertions a && a_1 is given by the semantics of the sequence of an inhale of a followed by an inhale of a_1. Recall from 2.29 that $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ is the totalised permission collection function. If $\mathbf{b}$ is True, then the permissions are collected iso-recursively.

$$\frac{}{(\mathrm{C}[\textbf{inhale } e], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume } e], \lambda)}$$

$$\frac{\mathbf{Q}^{\mathrm{T}}(\textbf{acc}(x.f,\, q), \lambda, \mathrm{True}) = \mathbf{x}}{(\mathrm{C}[\textbf{inhale } \textbf{acc}(x.f,\, q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \mathbf{Q}^{\mathrm{T}}(\textbf{acc}(x.f,\, q), \lambda, \mathrm{True}) = \pi_1}{(\mathrm{C}[\textbf{inhale } \textbf{acc}(x.f,\, q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma, \pi + \pi_1))}$$

$$\frac{\mathbf{Q}^{\mathrm{T}}(\textbf{acc}(p(\overline{e_i}), q), \lambda, \mathrm{True}) = \mathbf{x}}{(\mathrm{C}[\textbf{inhale } \textbf{acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \mathbf{Q}^{\mathrm{T}}(\textbf{acc}(p(\overline{e_i}), q), \lambda, \mathrm{True}) = \pi_1}{(\mathrm{C}[\textbf{inhale } \textbf{acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma, \pi + \pi_1))}$$

$$\frac{[\![e, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{inhale } e \rightarrow a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{[\![e, \lambda]\!] = \mathrm{False}}{(\mathrm{C}[\textbf{inhale } e \rightarrow a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda)}$$

$$\frac{[\![e, \lambda]\!] = \mathrm{True}}{(\mathrm{C}[\textbf{inhale } e \rightarrow a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{inhale } a], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{inhale } a_1 \,\&\&\, a_2], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{inhale } a_1;\ \textbf{inhale } a_2], \lambda)}$$

$$\frac{\mathbf{Q}^{\mathrm{T}}(a \mathrel{-\!\!*} a_1, \lambda, \mathrm{True}) = \mathbf{x}}{(\mathrm{C}[\textbf{inhale } a \mathrel{-\!\!*} a_1], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \mathbf{Q}^{\mathrm{T}}(a \mathrel{-\!\!*} a_1, \lambda, \mathrm{True}) = \pi_1}{(\mathrm{C}[\textbf{inhale } a \mathrel{-\!\!*} a_1], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma, \pi + \pi_1))}$$

$$\frac{}{(\mathrm{C}[\textbf{inhale forall } x : t :: a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume forall } x : t :: a], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{inhale exists } x : t :: a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume exists } x : t :: a], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{inhale } [a_1, a_2]], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{inhale } a_1], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{inhale forperm}[f]\ x :: a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume forperm}[f]\ x :: a], \lambda)}$$

Definition 2.73 *(Trace semantics of exhale statements)* The semantics of an exhale statement **exhale** a is to assert all expressions contained in a and to remove from a trace all permissions mentioned in accessibility sub-assertions of a. Furthermore, the semantics of exhaling a conjunct of assertions a && a_1 is given by the semantics of the sequence of an exhale of a followed by an exhale of a_1.

$$\frac{}{(\mathrm{C}[\textbf{exhale } e], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assert } e], \lambda)}$$

The semantics of exhaling the permission to a predicate is to *havoc* all heap locations for which the current trace λ has neither direct permission nor has permission folded inside a predicate it currently has access to. The rule below thus makes use of the global permission collection function $\mathbf{G}^{\mathrm{T}}(\_, \_)$. Recall that this function computes a permission mask π_2 that unifies all direct permissions given by $\pi^{\mathbf{F}}$ and all implicit permissions to heap locations given by a full unrolling of predicates to which the current predicate-permission mask $\pi^{\mathbf{P}}$ has access to. Note that this semantics corresponds to an *equi-recursive* treatment of predicates, which is needed for soundness reasons, but renders the semantics not implementable.

$$\frac{\exists\, i.\, [\![e_i, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{exhale acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q < 0}{(\mathrm{C}[\textbf{exhale acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) < \mathbf{Q}_q}{(\mathrm{C}[\textbf{exhale acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq \mathbf{Q}_q \quad \pi_1 = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}[\mathbf{p} \mapsto \pi^{\mathbf{P}}(\mathbf{p}) - \mathbf{Q}_q], \pi^{\mathbf{W}}) \quad \mathbf{G}^{\mathrm{T}}(\pi_1, \lambda) = \mathbf{x}}{(\mathrm{C}[\textbf{exhale acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq \mathbf{Q}_q \quad \pi_1 = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}[\mathbf{p} \mapsto \pi^{\mathbf{P}}(\mathbf{p}) - \mathbf{Q}_q], \pi^{\mathbf{W}}) \quad \mathbf{G}^{\mathrm{T}}(\pi_1, \lambda) = \pi_2 \quad \mathbf{h}_1 \overset{\pi_2}{\equiv} \mathbf{h}}{(\mathrm{C}[\textbf{exhale acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}_1, \sigma, \pi_1))}$$

The semantics of exhaling the permission to a field is analogous to the semantics of exhaling the permission to a predicate and is motivated by a similar reasoning as described above.

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \sigma(x) = \mathrm{null}}{(\mathrm{C}[\textbf{exhale acc}(x.f, q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{null}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q < 0}{(\mathrm{C}[\textbf{exhale acc}(x.f, q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q \geq 0 \quad \pi^{\mathbf{F}}(\mathbf{o}, f) < \mathbf{Q}_q}{(\mathrm{C}[\textbf{exhale acc}(x.f, q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q \geq 0 \quad \pi^{\mathbf{F}}(\mathbf{o},f) \geq \mathbf{Q}_q \quad \pi_1 = (\pi^{\mathbf{F}}[(\mathbf{o},f) \mapsto \pi^{\mathbf{F}}(\mathbf{o},f) - \mathbf{Q}_q], \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \mathbf{G}^{\mathrm{T}}(\pi_1,\lambda) = \mathbf{x}}{(\mathrm{C}[\textbf{exhale acc}(x.f,\, q)],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \quad \sigma(x) = \mathbf{o} \quad \mathbf{Q}_q \geq 0 \quad \pi^{\mathbf{F}}(\mathbf{o},f) \geq \mathbf{Q}_q \quad \pi_1 = (\pi^{\mathbf{F}}[(\mathbf{o},f) \mapsto \pi^{\mathbf{F}}(\mathbf{o},f) - \mathbf{Q}_q], \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \mathbf{G}^{\mathrm{T}}(\pi_1,\lambda) = \pi_2 \quad \mathbf{h}_1 \overset{\pi_2}{\equiv} \mathbf{h}}{(\mathrm{C}[\textbf{exhale acc}(x.f,\, q)],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h}_1,\sigma,\pi_1))}$$

$$\frac{[\![e,\lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{exhale } e \rightarrow a],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathbf{x}})}$$

$$\frac{[\![e,\lambda]\!] = \text{False}}{(\mathrm{C}[\textbf{exhale } e \rightarrow a],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda)}$$

$$\frac{[\![e,\lambda]\!] = \text{True}}{(\mathrm{C}[\textbf{exhale } e \rightarrow a],\lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a],\lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{exhale } a_1 \,\&\&\, a_2],\lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a_1;\ \textbf{exhale } a_2],\lambda)}$$

$$\frac{\mathbf{W}_n(a \mathrel{-\!\!*} a_1,\lambda) = \mathbf{x}}{(\mathrm{C}[\textbf{exhale } a \mathrel{-\!\!*} a_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \quad \mathbf{W}_n(a \mathrel{-\!\!*} a_1,\lambda) = \mathbf{w} \quad \pi^{\mathbf{W}}(\mathbf{w}) < 1}{(\mathrm{C}[\textbf{exhale } a \mathrel{-\!\!*} a_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda_{\text{perm}})}$$

$$\frac{(\mathbf{h},\sigma,\pi) = \text{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi^{\mathbf{W}}) \quad \mathbf{W}_n(a \mathrel{-\!\!*} a_1,\lambda) = \mathbf{w} \quad \pi^{\mathbf{W}}(\mathbf{w}) \geq 1 \quad \pi_1^{\mathbf{W}} = \pi^{\mathbf{W}}[\mathbf{w} \mapsto \pi^{\mathbf{W}}(\mathbf{w}) - 1]}{(\mathrm{C}[\textbf{exhale } a \mathrel{-\!\!*} a_1],\lambda) \rightsquigarrow (\mathrm{C}[\varepsilon],\lambda{:}(\mathbf{h},\sigma,(\pi^{\mathbf{F}},\pi^{\mathbf{P}},\pi_1^{\mathbf{W}})))}$$

$$\frac{}{(\mathrm{C}[\textbf{exhale forall } x : t :: a],\lambda) \rightsquigarrow (\mathrm{C}[\textbf{assert forall } x : t :: a],\lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{exhale exists } x : t :: a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assert exists } x : t :: a], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{exhale } [a_1, a_2]], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a_2], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{exhale forperm}[f]\ x :: a], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assert forperm}[f]\ x :: a], \lambda)}$$

Definition 2.74 *(Trace semantics of fold statements)* The semantics of a predicate-fold statement **fold acc**$(p(\overline{e_i}), q)$ is to remove from a trace λ all permissions implied by a *single* unrolling of p. These permissions are collected by the totalised permission collection function $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$, which was defined in 2.29.

$$\frac{\exists i.\ [\![e_i, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{fold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) = \mathbf{x}}{(\mathrm{C}[\textbf{fold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) = \pi_1 \quad \pi_1 > \pi}{(\mathrm{C}[\textbf{fold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) = \pi_1 \quad \pi_1 \leq \pi \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i})}{(\mathrm{C}[\textbf{fold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma, (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}[\mathbf{p} \mapsto \pi^{\mathbf{P}}(\mathbf{p}) + \mathbf{Q}_q], \pi^{\mathbf{W}}) - \pi_1))}$$

Definition 2.75 *(Trace semantics of unfold statements)* The semantics of a predicate-unfold statement **unfold acc**$(p(\overline{e_i}), q)$ is similar to the semantics of the predicate-fold statement but uses the totalised permission collection function $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ to *add* to a trace all permissions implied by a *single* unrolling of p.

$$\frac{\exists i.\, [\![e_i, \lambda]\!] = \mathbf{x}}{(\mathrm{C}[\textbf{unfold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q < 0}{(\mathrm{C}[\textbf{unfold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) < \mathbf{Q}_q}{(\mathrm{C}[\textbf{unfold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathrm{perm}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq \mathbf{Q}_q \quad \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) = \mathbf{x}}{(\mathrm{C}[\textbf{unfold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_{\mathbf{x}})}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}}) \quad \overline{x_i} = \mathrm{Param}(p) \quad \overline{[\![e_i, \lambda]\!] = \mathbf{v}_i} \quad \mathbf{Q}_q \geq 0 \quad \mathbf{p} = \mathbf{P}_n(p, \overline{\mathbf{v}_i}) \quad \pi^{\mathbf{P}}(\mathbf{p}) \geq \mathbf{Q}_q \quad \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p), \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma[\overline{x_i \mapsto \mathbf{v}_i}], \pi), \mathrm{True}) = \pi_1}{(\mathrm{C}[\textbf{unfold acc}(p(\overline{e_i}), q)], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma, (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}[\mathbf{p} \mapsto \pi^{\mathbf{P}}(\mathbf{p}) - \mathbf{Q}_q], \pi^{\mathbf{W}}) + \pi_1))}$$

Definition 2.76 *(Trace semantics of wand-application statements)* The semantics of a wand-apply statement **apply** $a \mathrel{-\!\!*} a_1$ is given by the semantics of the sequence of an exhale of the wand-instance itself, an exhale of a without randomisation of the heap (thus the usage of **exhale'** which is almost **exhale** but removes permissions without randomisation of the heap) and an inhale of a_1. Not to randomise the heap is important as a_1 might immediately give back permissions which are removed by the exhale of a. Thus, the heap values have to be preserved between exhaling a and inhaling a_1.

$$\frac{}{(\mathrm{C}[\textbf{apply } a \mathrel{-\!\!*} a_1], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a \mathrel{-\!\!*} a_1\textbf{; exhale' } a\textbf{; inhale } a_1], \lambda)}$$

Definition 2.77 *(Trace semantics of if-then-else statements)* The semantics of statement **if** e **then** s_1 **else** s_2 is to allow for two computations in the same context C and on the same trace λ: one computation reflects taking the if-branch, one reflects taking the else-branch. The semantics of taking the if-branch, for example, is given by the sequence of assuming the condition

e followed by the semantics of the statement s_1 making up the body of the if-branch.

$$\frac{}{(\mathrm{C}[\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume } e;\ s_1], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{assume } \neg e;\ s_2], \lambda)}$$

Definition 2.78 *(Trace semantics of while statements)* The semantics of statement **while** e **invariant** a **do** s_1 is to allow for two computations in the same context C and on the same trace λ: the first computation reflects entering the body of the while loop and the second computation reflects the body of the while loop not being entered. Below, the variables to havoc are the variables which are written to inside the body s_1 of the while-statement.

$$\frac{}{(\mathrm{C}[\textbf{while } e \textbf{ invariant } a \textbf{ do } s_1], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a; \overline{\textbf{havoc } x}; \textbf{inhale } a \,\&\&\, e; s_1; \textbf{exhale } a], \lambda)}$$

$$\frac{}{(\mathrm{C}[\textbf{while } e \textbf{ invariant } a \textbf{ do } s_1], \lambda) \rightsquigarrow (\mathrm{C}[\textbf{exhale } a; \overline{\textbf{havoc } x}; \textbf{inhale } a \,\&\&\, \neg e], \lambda)}$$

$$\frac{(\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \quad \mathbf{v} \in \mathbf{D}_x}{(\mathrm{C}[\textbf{havoc } x], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\mathbf{h}, \sigma[x \mapsto \mathbf{v}], \pi))}$$

As has been noted, the semantics of Viper statements on the level of a single partial trace is non-deterministic with non-determinism introduced by the following statements: variable declarations (definition 2.64), new-statements (definition 2.65), exhaling permissions to fields (definition 2.73) and while-statements (definition 2.78).

2.2.4 Verified Viper Program

This section introduces the notion of a verified Viper program. The verification of a Viper program is a judgment about all possible evaluations of the Viper program. Hence, the trace semantics defined in the previous chapter is not directly suitable to capture the notion of a verified program but first requires a lifting to the level of sets of partial traces. As a preliminary to formally defining the notion of a verified program, the trace semantics, defined in the previous chapter, is thus lifted to the level of sets of traces. Moreover, a reflexive and transitive closure is defined over this lifting, with the closure then being the basis for defining the notion of a verified Viper program.

Definition 2.79 *(Lifting of trace semantics to sets of traces)* Assume ψ to be a set of partial traces and complete traces. To allow for the evaluation of ψ, the trace semantics $\rightsquigarrow$, defined in section 2.2.3, is lifted to sets of traces as follows:

$$\psi \rightsquigarrow \{ (s_1, \lambda_1) \mid (s, \lambda) \in \psi. (s, \lambda) \rightsquigarrow (s_1, \lambda_1) \}$$

Lemma 2.80 *(Lifting of trace semantics is total)*
$\forall \psi. \exists \psi_1. \psi \rightsquigarrow \psi_1$

Proof By induction on the size of ψ, a case analysis of the next statement to be evaluated in the last partial trace added to ψ, using totality of the totalised assertion evaluation $[\![a, \lambda]\!]$ (lemma 2.59), using totality of the totalised local permission collection $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ (lemma 2.30) and using totality of the totalised global permission collection $\mathbf{G}^{\mathrm{T}}(\pi, \lambda)$ (lemma 2.34). □

Lemma 2.81 *(Lifting of trace semantics is deterministic)*
$\forall \psi. \psi \rightsquigarrow \psi_1 \ \wedge \ \psi \rightsquigarrow \psi_2 \ \Rightarrow \ \psi_1 = \psi_2$

Proof By induction on the size of ψ, a case analysis of the next Viper statement to be evaluated in the last partial trace added to ψ, using determinism of the totalised assertion evaluation $[\![a, \lambda]\!]$ (lemma 2.61), using determinism of the totalised local permission collection $\mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathbf{b})$ (lemma 2.32) and using determinism of the totalised global permission collection $\mathbf{G}^{\mathrm{T}}(\pi, \lambda)$ (lemma 2.36). □

Definition 2.82 *(Reflexive and transitive closure of the lifting)* Assume ψ to be a set of partial traces and complete traces. The reflexive and transitive closure of the lifted trace semantics $\rightsquigarrow$ on ψ is denoted by $\mathrm{V}[\psi]$ and defined as follows:

$$\mathrm{V}[\psi] = \begin{cases} \psi & \text{if } \psi = \varnothing \\ \psi \ \cup \ \mathrm{V}[\psi_1] & \text{if } \psi \neq \varnothing \text{ and } \psi \rightsquigarrow \psi_1 \end{cases}$$

Lemma 2.83 *(Closure is total)* Assume ψ to be a set of partial traces and complete traces where every partial trace is associated with a finite evaluation context C and a Viper statement s of finite size. Then: $\exists \psi_1. \mathrm{V}[\psi] = \psi_1$.

Proof Lemma 2.80 shows the lifted trace semantics $\rightsquigarrow$ to be total. Moreover, the (non-lifted) trace semantics has no rule for complete traces. Thus, $\mathrm{V}[\psi]$ terminates if the assumptions, as given by the statement of the lemma, hold for ψ. □

Lemma 2.84 *(Closure is deterministic)* Assume ψ to be a set of partial traces and complete traces where every partial trace is associated with a finite evaluation context C and a Viper statement s of finite size. Then: $\mathrm{V}[\psi] = \psi_1 \ \wedge \ \mathrm{V}[\psi] = \psi_2 \ \Rightarrow \psi_1 = \psi_2$.

Proof Follows from lemma 2.81 which shows the lifted trace semantics $\rightsquigarrow$ to be deterministic and from the totality of the closure as shown in lemma 2.83. □

Definition 2.85 *(Trace model of Viper assertion)* The judgment of a Viper trace λ to *model* a Viper assertion a is expressed through the predicate $\lambda \models a$ and defined as follows: $\lambda \models a \Leftrightarrow [\![a, \lambda]\!] = \mathrm{True}$. Most importantly, it follows from 2.37 that error-traces don't model Viper assertions. Also recall $[\![\_, \_]\!]$ to denote the totalised assertion evaluation function defined in chapter 2.58.

Definition 2.86 *(Configuration model of Viper assertion)* The judgment of a triple of a Viper heap $\mathbf{h}$, a Viper store σ and a Viper permission mask π to model a Viper assertion a is expressed through the predicate $(\mathbf{h}, \sigma, \pi) \models a$ and defined as follows: $(\mathbf{h}, \sigma, \pi) \models a \Leftrightarrow \Lambda[(\mathbf{h}, \sigma, \pi)] \models a$. Above, $\Lambda[(\mathbf{h}, \sigma, \pi)]$ denotes the trace consisting of the single configuration $(\mathbf{h}, \sigma, \pi)$.

Definition 2.87 *(Complete trace model of Viper assertion)* The judgment of a complete Viper trace (ε, λ) to *model* a Viper assertion a is expressed through the predicate $(\varepsilon, \lambda) \models a$ and defined as follows: $(\varepsilon, \lambda) \models a \Leftrightarrow \lambda \models a$.

Definition 2.88 *(Set of partial traces and complete traces model of Viper assertion)* Assume ψ to be a set of partial traces and complete traces. The judgment of ψ to *model* a Viper assertion a is expressed through the predicate $\psi \models a$ and defined as follows: $\psi \models a \Leftrightarrow \forall (\varepsilon, \lambda) \in \psi. (\varepsilon, \lambda) \models a$.

Definition 2.89 *(Verified Viper program)* The judgment of a Viper program *prog* to be *verified* is expressed through the predicate $VP(prog)$ and defined as follows: $VP(prog) \Leftrightarrow \forall m \in prog, \forall \mathbf{h}, \sigma, \pi. (\mathbf{h}, \sigma, \pi) \models \mathrm{Pre}(m) \Rightarrow$
$\mathrm{V}[\{ (\mathrm{Body}(m), \Lambda[(\mathbf{h}, \sigma, \pi)]) \}] \models \mathrm{Post}(m)$.

Importantly, the deterministic nature of the closure $\mathrm{V}[\psi]$, as proven in lemma 2.84, makes the judgment of a Viper program to be verified a deterministic one.

Chapter 3

Chalice

Contents

3.1 Syntax

This chapter presents an abstract syntax of Chalice [2]. The concrete syntax of Chalice programs is given in appendix D. As in the presentation of the abstract syntax of Viper programs in chapter 2.1, the abstract syntax of Chalice will also be given as a syntactic domain, presented in an EBNF-like notation.

Definition 3.1 *(Syntactic domain of expressions)* The syntactic domain of *expressions* is denoted by E and has metavariable e associated. The domain of *expressions* is defined as follows.

$$E \coloneqq null \mid b \mid n \mid x \mid x.f \mid \text{-}\, e \mid e + e \mid e \,/\, e \mid \neg\, e \mid e = e \mid x.fun(\bar{e}) \mid \mathbf{old}(e)$$

Above, b is the metavariable ranging over the syntactic domain of booleans B and n ranges over numerals N. Furthermore, x identifies a *variable* and is the metavariable associated with the syntactic domain of variable identifiers Var. Similarly, f, fun and p are associated with the domains F, Fun and P and identify *fields*, *functions* and *predicates*.

The full list of unary operators and binary operators available in Chalice can be found in appendix D. However, the operators listed above suffice to subsequently demonstrate the semantics of Chalice.

Definition 3.2 *(Syntactic domain of assertions)* The syntactic domain of *assertions* is denoted by A and has metavariable a associated. The domain of *assertions* is defined as follows.

$$A \coloneqq e \mid \mathbf{acc}(x.joinable) \mid \mathbf{acc}(x.f,\, n) \mid \mathbf{acc}(x.p) \mid e \rightarrow a \mid a \;\&\&\; a$$

Definition 3.3 *(Syntactic domain of statements)* The syntactic domain of *statements* is denoted by S and has metavariable s associated. The domain of *statements* is defined as follows.

$$S \coloneqq \mathbf{var}\ x : t \mid x := e \mid x.f := x \mid \mathbf{return}\ x \mid x := \mathbf{new}\ c \mid \mathbf{fold}\ \mathbf{acc}(x.p) \mid \mathbf{unfold}\ \mathbf{acc}(x.p) \mid \mathbf{fork}\ x := x.m(\bar{x}) \mid \mathbf{join}\ x := x \mid \mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s \mid s;\ s$$

Above, c identifies a *class* and is the metavariable associated with the syntactic domain of class identifiers C. Similarly, m identifies a *method* and is associated with the domain M.

Definition 3.4 *(Syntactic domain of fields)* The syntactic domain of *fields* is denoted by $Field$ and has metavariable *field* associated. The domain of *fields* is defined as follows.

$$Field \coloneqq \mathbf{var}\ f : t$$

Definition 3.5 *(Syntactic domain of invariants)* The syntactic domain of *invariants* is denoted by *Inv* and has metavariable *inv* associated. The domain of *invariants* is defined as follows.

$Inv \coloneqq \textbf{invariant}\ a$

Definition 3.6 *(Syntactic domain of functions)* The syntactic domain of *functions* is denoted by *Funct* and has metavariable *funct* associated. The domain of *functions* is defined as follows.

$Funct \coloneqq \textbf{function}\ fun(\overline{x : t}) : t\ \overline{\textbf{requires}\ a}\ \overline{\textbf{ensures}\ a}\ \{\ e\ \}$

Definition 3.7 *(Syntactic domain of predicates)* The syntactic domain of *predicates* is denoted by *Pred* and has metavariable *pred* associated. The domain of *predicates* is defined as follows.

$Pred \coloneqq \textbf{predicate}\ p\ \{\ a\ \}$

Definition 3.8 *(Syntactic domain of methods)* The syntactic domain of *methods* is denoted by *Meth* and has metavariable *meth* associated. The domain of *methods* is defined as follows.

$Meth \coloneqq \textbf{method}\ m(\overline{x : t})\ \textbf{returns}\ \overline{x : t}\ \overline{\textbf{requires}\ a}\ \overline{\textbf{ensures}\ a}\ \{\ s\ \}$

Definition 3.9 *(Syntactic domain of declarations)* The syntactic domain of *declarations* is denoted by *Decl* and has metavariable *decl* associated. The domain of *declarations* is defined as follows.

$Decl \coloneqq field \mid inv \mid funct \mid pred \mid meth$

Definition 3.10 *(Syntactic domain of classes)* The syntactic domain of *classes* is denoted by *Class* and has metavariable *class* associated. The domain of *classes* is defined as follows.

$Class \coloneqq \textbf{class}\ c\ \{\ \overline{decl}\ \}$

Definition 3.11 *(Syntactic domain of Chalice programs)* The syntactic domain of *Chalice programs* is denoted by *Prog* and has metavariable *prog* associated. The domain of *Chalice programs* is defined as follows.

$Prog \coloneqq \overline{class}$

3.2 Semantics

The Chalice semantics presented subsequently is an extension of the semantics presented by Summers and Drossopoulou in [9].

3.2.1 Preliminary

Unless otherwise specified, all functions defined in this preliminary are assumed to be total.

Definition 3.12 *(Semantic domain of Booleans, Integer, objects and errors)* The semantic domains of *Booleans* (denoted by **B** and with metavariable **b**), *Integers* (denoted by **Z** and with metavariable **z**), *objects* (denoted by **O** and with metavariable **o**) and *errors* (denoted by **X** and with metavariable **x**) are defined as in the preliminary of the Viper semantics in chapter 2.2.1.

Definition 3.13 *(Semantic domain of thread identifiers)* An infinite semantic domain of *thread identifiers* is assumed and denoted by Γ. The metavariable associated with the domain Γ is τ.

Definition 3.14 *(Semantic domain of method identifiers)* An infinite semantic domain of *method identifiers* is assumed and denoted by **M**. The metavariable associated with the domain of **M** is **m**.

Definition 3.15 *(Semantic domain of values)* The semantic domain of *values* is denoted by **V**, has metavariable **v** associated and is the disjoint union of the domains of *Booleans* **B**, *Integers* **Z**, *objects* **O**, *thread identifiers* Γ, *method identifiers* **M** and additionally contains the special value null, i.e. $\mathbf{V} = \mathbf{B} \cup \mathbf{Z} \cup \mathbf{O} \cup \Gamma \cup \mathbf{M} \cup \{\,\text{null}\,\}$. Furthermore, the function $\mathbf{D} : T \mapsto \{\,\mathbf{B}, \mathbf{Z}, \mathbf{O} \cup \{\text{null}\}\,\}$ is assumed and maps syntactic types t to the corresponding semantic value sub-domain. The value sub-domain to which t is mapped under **D** is denoted by $\mathbf{D}_t$. Important example: $\mathbf{D}_c = \mathbf{O} \cup \{\text{null}\}$, for any class identifier c.

Definition 3.16 *(Semantic domain of heaps)* The semantic domain of *heaps* is denoted by H, has metavariable h associated and is a domain of functions mapping either tuples of semantic objects **o** and syntactic field identifiers f or tuples of semantic thread identifiers τ and syntactic field identifiers f to semantic values. Thus, $\mathrm{h} : ((\mathbf{O} \times F) \mapsto \mathbf{V}) \cup ((\Gamma \times F) \mapsto \mathbf{V})$.

Definition 3.17 *(Semantic domain of runtime heaps)* The semantic domain of *runtime heaps* is denoted by **H**, has metavariable **h** associated and is a domain of functions mapping tuples of semantic objects **o** and syntactic field identifiers f to semantic values **v**. Thus, $\mathbf{h} : (\mathbf{O} \times F) \mapsto \mathbf{V}$.

Definition 3.18 *(Semantic domain of stores)* The semantic domain of *stores* is denoted by Σ, has metavariable σ associated and is a domain of functions mapping syntactic variable identifiers x to semantic values **v**. Hence, $\sigma : \mathit{Var} \mapsto \mathbf{V}$.

Definition 3.19 *(Semantic domain of field-permission masks)* The semantic domain of *field-permission masks* is denoted by $\Pi_{\mathbf{F}}$, has metavariable $\pi^{\mathbf{F}}$ associated and is a domain of functions mapping either tuples of semantic objects

o and syntactic field identifiers f or tuples of semantic thread identifiers τ and syntactic field identifiers f to non-negative values in $\mathbf{Z}$. Hence, $\pi^{\mathbf{F}} : ((\mathbf{O} \times F) \mapsto \mathbf{Z}) \cup ((\Gamma \times F) \mapsto \mathbf{Z})$. Furthermore, the field-permission mask mapping all pairs of objects and fields to 0 and all pairs of thread identifiers and fields to 0 is denoted by $\varnothing^{\mathbf{F}}$. Addition of field-permission masks is defined as follows: $(\pi^{\mathbf{F}} + \pi_1^{\mathbf{F}})(\mathbf{o}, f) = \pi^{\mathbf{F}}(\mathbf{o}, f) + \pi_1^{\mathbf{F}}(\mathbf{o}, f)$ and $(\pi^{\mathbf{F}} + \pi_1^{\mathbf{F}})(\tau, f) = \pi^{\mathbf{F}}(\tau, f) + \pi_1^{\mathbf{F}}(\tau, f)$. Equality of field-permission masks is defined as follows: $\pi^{\mathbf{F}} = \pi_1^{\mathbf{F}} \Leftrightarrow \forall(\mathbf{o},f).\, \pi^{\mathbf{F}}(\mathbf{o},f) = \pi_1^{\mathbf{F}}(\mathbf{o},f) \wedge \forall(\tau,f).\, \pi^{\mathbf{F}}(\tau,f) = \pi_1^{\mathbf{F}}(\tau,f)$.

Definition 3.20 *(Semantic domain of predicate-permission masks)* The semantic domain of *predicate-permission masks* is denoted by $\Pi_{\mathbf{P}}$, has metavariable $\pi^{\mathbf{P}}$ associated and is a domain of functions mapping tuples of semantic objects **o** and syntactic predicate identifiers p to non-negative values in $\mathbf{Z}$. Hence, $\pi^{\mathbf{P}} : (\mathbf{O} \times P) \mapsto \mathbf{Z}$. Furthermore, the predicate-permission mask mapping all pairs of objects and predicate-identifiers to 0 is denoted by $\varnothing^{\mathbf{P}}$. Addition of predicate-permission masks is defined as follows: $(\pi^{\mathbf{P}} + \pi_1^{\mathbf{P}})(\mathbf{o}, p) = \pi^{\mathbf{P}}(\mathbf{o}, p) + \pi_1^{\mathbf{P}}(\mathbf{o}, p)$. Equality of predicate-permission masks is defined as follows: $\pi^{\mathbf{P}} = \pi_1^{\mathbf{P}} \Leftrightarrow \forall(\mathbf{o},p).\, \pi^{\mathbf{P}}(\mathbf{o},p) = \pi_1^{\mathbf{P}}(\mathbf{o},p)$

Definition 3.21 *(Semantic domain of permission masks)* The semantic domain of *permission masks* is denoted by Π, has metavariable π associated and is a domain of tuples over the domains of field-permission masks $\Pi_{\mathbf{F}}$ and predicate-permission masks $\Pi_{\mathbf{P}}$. Hence, $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$. Furthermore, the permission mask consisting of $\varnothing^{\mathbf{F}}$ and $\varnothing^{\mathbf{P}}$ is denoted by $\pi_{\varnothing}$, i.e. $\pi_{\varnothing} = (\varnothing^{\mathbf{F}}, \varnothing^{\mathbf{P}})$. Addition of permission masks is defined as follows: $(\pi^{\mathbf{F}}, \pi^{\mathbf{P}}) + (\pi_1^{\mathbf{F}}, \pi_1^{\mathbf{P}}) = (\pi^{\mathbf{F}} + \pi_1^{\mathbf{F}}, \pi^{\mathbf{P}} + \pi_1^{\mathbf{P}})$. Equality of permission masks is defined as follows: $(\pi^{\mathbf{F}}, \pi^{\mathbf{P}}) = (\pi_1^{\mathbf{F}}, \pi_1^{\mathbf{P}}) \Leftrightarrow \pi^{\mathbf{F}} = \pi_1^{\mathbf{F}} \wedge \pi^{\mathbf{P}} = \pi_1^{\mathbf{P}}$.

Definition 3.22 *(Permission Collection Function)* To collect all permissions explicitly requested by assertion a, the function $\mathbf{Z} : (A \times \mathrm{H} \times \mathrm{H} \times \Sigma) \mapsto \mathbf{Z} \cup \mathbf{X}$ is assumed. $\mathbf{Z}$ collects the permissions inside assertion *a iso-recursively* [9]: predicates inside assertion a are not unrolled but are treated as an entity to which permissions can be either obtained or lost. The full definition of $\mathbf{Z}$ can be found as algorithm 3 in appendix A. Note that the definition of $\mathbf{Z}$ makes use of the expression-evaluation function $[\![e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma]\!]$, which is defined in 3.36.

Definition 3.23 *(Semantic domain of thread configurations)* The semantic domain of *thread configurations* is denoted by C, has metavariable c associated and contains triples of semantic heaps h, semantic stores σ and syntactic statements s as well as the special element idle: $\mathrm{C} = (\mathrm{H} \times \Sigma \times S) \cup \{\, \mathrm{idle}, \mathrm{trash} \,\}$.

Definition 3.24 *(Semantic domain of thread entities)* The semantic domain of *thread entities* is denoted by $\mathbf{E}_\Gamma$ and is a domain of tuples of thread configurations c and thread identifiers τ: $\mathbf{E}_\Gamma = \mathrm{C} \times \Gamma$. The metavariable associated

with the domain $\mathbf{E}_\Gamma$ is $\mathbf{e}_\tau$ and represents the tuple (c, τ). $\mathbf{e}_\tau$ is called *active* if it represents a tuple (c, τ) with $\mathrm{c} = (\mathrm{h}, \sigma, s)$. If $\mathbf{e}_\tau$ has thread configuration $\mathrm{c} = \text{idle}$ then $\mathbf{e}_\tau$ is called *idle* and if $\mathbf{e}_\tau$ has thread configuration $\mathrm{c} = \text{trash}$ then $\mathbf{e}_\tau$ is called *trashed*.

Definition 3.25 *(Semantic domain of object entities)* The semantic domain of *object entities* is denoted by $\mathbf{E_O}$ and is a domain of tuples of an element out of $\{\,\text{alloc}, \text{free}\,\}$ and semantic objects $\mathbf{o}$: $\mathbf{E_O} = \{\,\text{alloc}, \text{free}\,\} \times \mathbf{O}$. The metavariable associated with the domain $\mathbf{E_O}$ is $\mathbf{e_o}$ and represents either the tuple $(\text{alloc}, \mathbf{o})$ or the tuple $(\text{free}, \mathbf{o})$.

Definition 3.26 *(Semantic domain of entities)* The semantic domain of *entities* is denoted by $\mathbf{E}$ and is the union of the domain of *thread entities* and the domain of *object entities*: $\mathbf{E} = \mathbf{E}_\Gamma \cup \mathbf{E_O}$. The metavariable associated with the domain $\mathbf{E}$ is $\mathbf{e}_l$. If $\mathbf{e}_l$ represents a thread entity then $l = \tau$ for some thread identifier τ. If $\mathbf{e}_l$ represents an object entity then $l = \mathbf{o}$ for some semantic object $\mathbf{o}$.

Definition 3.27 *(Semantic domain of runtime entity collections)* The semantic domain of *runtime entity collections* is denoted by $\mathbf{R}$ and is a domain of tuples of sets of thread entities $\mathbf{e}_\tau$ and sets of object entities $\mathbf{e_o}$. The metavariable associated with the domain $\mathbf{R}$ is $\mathbf{r}$ and hence $\mathbf{r} = (\overline{\mathbf{e}_\tau}, \overline{\mathbf{e_o}})$.

Definition 3.28 *(Semantic domain of runtime configurations)* The semantic domain of *runtime configurations* is denoted by $\mathbf{C}$, has metavariable $\mathbf{c}$ associated and is a domain of tuples over the domain of runtime heaps $\mathbf{H}$ and the domain of runtime entity collections $\mathbf{R}$. Thus, $\mathbf{c} = (\mathbf{h}, \mathbf{r})$.

Definition 3.29 *(Mapping between heaps and runtime heaps)* Recall runtime heaps $\mathbf{h}$ to only accept objects as receivers while heaps h accept both, objects as well as thread identifiers as receivers. In order for threads having access to information about other threads currently running, the function $\lceil \_, \_ \rfloor : \mathbf{H} \times \mathbf{R} \mapsto \mathrm{H}$, defined below, creates a heap h which contains all information in a runtime heap $\mathbf{h}$ and all information regarding currently active threads of a runtime entity collection $\mathbf{r}$.

$$\lceil \mathbf{h}, \mathbf{r} \rfloor (\mathbf{o}, f) = \mathbf{h}(\mathbf{o}, f)$$

$$\lceil \mathbf{h}, \mathbf{r} \rfloor (\tau, f) = \begin{cases} \sigma(\mathit{this}) & \text{if } f = \mathit{recv} \wedge ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r} \\ \sigma(x_i) & \text{if } f = \mathit{param}_i \wedge ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r} \wedge \overline{x_i} = \mathrm{Param}(\sigma(\mathit{meth})) \\ \sigma(\mathit{meth}) & \text{if } f = \mathit{meth} \wedge ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r} \end{cases}$$

The inverse, mapping heaps h to runtime heaps **h**, is given by restricting the domain of the heap h to only have objects as receivers. For the sake of type-correctness in the following formulas, the function $\lfloor \_ \rceil : \mathrm{H} \mapsto \mathbf{H}$ is now defined which computes this transformation.

$$\lfloor \mathrm{h} \rceil (\mathbf{o}, f) = \mathrm{h}(\mathbf{o}, f)$$

3.2.2 Evaluation of Expressions

The evaluation of an expression e takes place with respect to a triple consisting of a heap $\mathrm{h}^{\mathrm{old}}$, a heap h and a store σ. Heap $\mathrm{h}^{\mathrm{old}}$, as explained in more detail in chapter 3.2.3, is a snapshot of the Chalice state as it was just before a thread started the execution of its method. Similarly, h is a snapshot of the Chalice state as it was just before a thread executed its next statement. Expression e then evaluates with $\mathrm{h}^{\mathrm{old}}$, h and σ to $\mathbf{r} \in \mathbf{V} \cup \mathbf{X}$ if the subsequently inductively defined semantic function $\Downarrow$ maps e, given the triple $(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)$, to $\mathbf{r}$. If $\Downarrow$ maps e, given the triple $(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)$, to $\mathbf{r}$, the notation $e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{r}$ will be used.

Definition 3.30 *(Evaluation of null, booleans, numerals and variable identifiers)*

$$\frac{}{null \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathrm{null}} \qquad \frac{}{b \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{B}_b} \qquad \frac{}{n \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{Z}_n} \qquad \frac{}{x \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \sigma(x)}$$

Definition 3.31 *(Evaluation of field-lookups)*

$$\frac{\sigma(x) = \mathrm{null}}{x.f \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{x}_{\mathrm{null}}} \qquad \frac{\sigma(x) = \mathbf{o}}{x.f \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathrm{h}(\mathbf{o}, f)}$$

Definition 3.32 *(Evaluation of unary operations)* Evaluation of unary operations is shown for the case of a unary negative operation. The remaining cases evaluate analogous.

$$\frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{x}}{\text{- } e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{x}} \qquad \frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{v}}{\text{- } e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \text{- } \mathbf{v}}$$

Definition 3.33 *(Evaluation of binary operations)* Evaluation of binary operation expressions is shown for the case of a division operation. This case is special as it potentially generates a division-by-zero error. Evaluation of the remaining cases is analogous with the exception of not having to capture the division-by-zero error.

$$\frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{e \;/\; e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}} \qquad \frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{e \;/\; e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}} \qquad \frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} 0}{e \;/\; e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}_{\mathrm{div}}}$$

$$\frac{e \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_1}{e \;/\; e_1 \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}/\mathbf{v}_1}$$

Definition 3.34 *(Evaluation of function calls)* The semantics of a function-call renders the expression evaluation function $\Downarrow$ partial if the function to evaluate recurses infinitely.

$$\frac{\sigma(x) = \mathrm{null}}{x.fun(\overline{e_i}) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}_{\mathrm{null}}} \qquad \frac{\sigma(x) = \mathbf{o} \quad \exists\, i : e_i \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{x.fun(\overline{e_i}) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}$$

$$\frac{\sigma(x) = \mathbf{o} \quad \overline{e_i \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad \mathrm{Body}(fun) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma[this \mapsto \mathbf{o}][\overline{x_i \mapsto \mathbf{v}_i}])} \mathbf{x}}{x.fun(\overline{e_i}) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}$$

$$\frac{\sigma(x) = \mathbf{o} \quad \overline{e_i \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(fun) \quad \mathrm{Body}(fun) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma[this \mapsto \mathbf{o}][\overline{x_i \mapsto \mathbf{v}_i}])} \mathbf{v}}{x.fun(\overline{e_i}) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}}$$

Definition 3.35 *(Evaluation of old expressions)* In essence, **old** is pushed down to the leafs of the expression-tree making up expression e. Field-lookups then take place via $\mathrm{h}^{\mathrm{old}}$ which reflects the Chalice state as it was just before a thread started executing its method.

$$\frac{}{\mathbf{old}(null) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathrm{null}} \qquad \frac{}{\mathbf{old}(b) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{B}_b}$$

$$\frac{}{\mathbf{old}(n) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{Z}_n} \qquad \frac{}{\mathbf{old}(x) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \sigma(x)}$$

$$\frac{\sigma(x) = \mathrm{null}}{\mathbf{old}(x.f) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}_{\mathrm{null}}} \qquad \frac{\sigma(x) = \mathbf{o}}{\mathbf{old}(x.f) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathrm{h}^{\mathrm{old}}(\mathbf{o},f)}$$

$$\frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{\mathbf{old}(\text{- } e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}} \qquad \frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}}{\mathbf{old}(\text{- } e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \text{- } \mathbf{v}}$$

$$\frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{\mathbf{old}(e \ / \ e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}} \qquad \frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad \mathbf{old}(e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{\mathbf{old}(e \ / \ e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}$$

$$\frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad \mathbf{old}(e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} 0}{\mathbf{old}(e \ / \ e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}_{\mathrm{div}}}$$

$$\frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v} \quad \mathbf{old}(e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_1}{\mathbf{old}(e \ / \ e_1) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}/\mathbf{v}_1}$$

$$\frac{\sigma(x) = \mathrm{null}}{\mathbf{old}(x.\mathit{fun}(\overline{e_i})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}_{\mathrm{null}}} \qquad \frac{\sigma(x) = \mathbf{o} \quad \exists\, i : \mathbf{old}(e_i) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{\mathbf{old}(x.\mathit{fun}(\overline{e_i})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}$$

$$\frac{\sigma(x) = \mathbf{o} \quad \overline{\mathbf{old}(e_i) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(\mathit{fun}) \quad \mathbf{old}(\mathrm{Body}(\mathit{fun})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma[\mathit{this}\mapsto\mathbf{o}]\overline{[x_i\mapsto\mathbf{v}_i]})} \mathbf{x}}{\mathbf{old}(x.\mathit{fun}(\overline{e_i})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}$$

$$\frac{\sigma(x) = \mathbf{o} \quad \overline{\mathbf{old}(e_i) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}_i} \quad \overline{x_i} = \mathrm{Param}(\mathit{fun}) \quad \mathbf{old}(\mathrm{Body}(\mathit{fun})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma[\mathit{this}\mapsto\mathbf{o}]\overline{[x_i\mapsto\mathbf{v}_i]})} \mathbf{v}}{\mathbf{old}(x.\mathit{fun}(\overline{e_i})) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}}$$

$$\frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}}{\mathbf{old}(\mathbf{old}(e)) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{x}} \qquad \frac{\mathbf{old}(e) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}}{\mathbf{old}(\mathbf{old}(e)) \Downarrow_{(\mathrm{h}^{\mathrm{old}},\mathrm{h},\sigma)} \mathbf{v}}$$

Definition 3.36 *(Value of expressions)* The expression evaluation function $\Downarrow$ defined above is *partial* as the evaluation of a function-call with an infinite recursion won't terminate. In order to have a semantics of total functions, the function $[\![\_,\_,\_,\_]\!] : E \times \mathrm{H} \times \mathrm{H} \times \Sigma \mapsto \mathbf{V} \cup \mathbf{X}$ is now defined and computes, even in the presence of infinitely recursing functions, either the value to which an expression evaluates to or, in case of a non-terminating evaluation, an error.

$$\llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \begin{cases} \mathbf{v} & \text{if } e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{v} \\ \mathbf{x} & \text{if } e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{x} \\ \mathbf{x}_{\mathrm{inf}} & \text{if } \nexists \mathbf{v}.\, e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{v} \ \wedge\ \nexists \mathbf{x}.\, e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{x} \end{cases}$$

The following lemmas demonstrate that the totalised expression evaluation function $\llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket$ is deterministic.

Lemma 3.37 *(Totalised expression evaluation is total)*
$\forall e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma. \exists \mathbf{u} \in \mathbf{V} \cup \mathbf{X}. \llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathbf{u}$

Proof Follows from totalising conditions in definition of $\llbracket \_, \_, \_, \_ \rrbracket$. □

Lemma 3.38 *(Assertion evaluation is deterministic)*
$\forall e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \forall \mathbf{u}_1, \mathbf{u}_2 \in \mathbf{V} \cup \mathbf{X}. e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{u}_1 \ \wedge\ e \Downarrow_{(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma)} \mathbf{u}_2 \ \Rightarrow\ \mathbf{u}_1 = \mathbf{u}_2$

Proof By structural induction on e. □

Corollary 3.39 *(Totalised expression evaluation is deterministic)*
$\forall e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma. \llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathbf{u}_1 \ \wedge\ \llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathbf{u}_2 \ \Rightarrow \mathbf{u}_1 = \mathbf{u}_2$

Proof Follows from lemmas 3.37 and 3.38. □

Based on the definitions above, the judgment of a model of a Chalice assertion can now be defined.

Definition 3.40 *(Model of Chalice assertion)* The judgment of a 4-tuple of a Chalice heap $\mathrm{h}^{\mathrm{old}}$, a Chalice heap h, a Chalice store σ and a Chalice permission mask π to *model* a Chalice assertion a is expressed through the predicate $(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi) \models a$ and defined as follows:

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi) \models e \Leftrightarrow \llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathrm{True}$$

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})) \models \mathbf{acc}(x.\mathit{joinable}) \Leftrightarrow \sigma(x) = \tau \wedge \pi^{\mathbf{F}}(\tau, \mathit{joinable}) \geq 100$$

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})) \models \mathbf{acc}(x.f, n) \Leftrightarrow \sigma(x) = \mathbf{o} \ \wedge\ \mathbf{Z}_n \geq 0 \ \wedge \pi^{\mathbf{F}}(\mathbf{o}, f) \geq \mathbf{Z}_n$$

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})) \models \mathbf{acc}(x.p) \Leftrightarrow \sigma(x) = \mathbf{o} \wedge \pi^{\mathbf{P}}(\mathbf{o}, p) \geq 100$$

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi) \models e \ \rightarrow\ a \Leftrightarrow \llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathrm{True} \Rightarrow (\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi) \models a$$

$$(\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi) \models a_1 \ \&\&\ a_2 \Leftrightarrow \exists \pi_1, \pi_2. \pi = \pi_1 + \pi_2 \wedge (\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi_1) \models a_1 \wedge (\mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma, \pi_2) \models a_2$$

3.2.3 Local Operational Semantics

This section defines an operational semantics for statements which affect single thread entities only. Assume such a statement to be denoted by s. Furthermore, assume a heap h and a thread entity $\mathbf{e}_\tau$ with a configuration $(\mathrm{h}^{\mathrm{old}}, \sigma, (s; s_1))$, i.e. $\mathbf{e}_\tau = ((\mathrm{h}^{\mathrm{old}}, \sigma, (s; s_1)), \tau)$. Heap $\mathrm{h}^{\mathrm{old}}$ in thread entity $\mathbf{e}_\tau$ reflects the method pre-state and is used during the evaluation of old-expressions. Statement s then leads the heap h and the thread entity $\mathbf{e}_\tau$ to a new heap h_1 and a new thread entity $\mathbf{e}'_\tau$ if the subsequently inductively defined semantic function $\rightharpoonup$ maps the tuple $(\mathrm{h}, \mathbf{e}_\tau)$ to $(\mathrm{h}_1, \mathbf{e}'_\tau)$. If $\rightharpoonup$ maps the tuple $(\mathrm{h}, \mathbf{e}_\tau)$ to $(\mathrm{h}_1, \mathbf{e}'_\tau)$, the notation $\mathrm{h}, \mathbf{e}_\tau \rightharpoonup \mathrm{h}_1, \mathbf{e}'_\tau$ will be used.

Definition 3.41 *(Semantics of variable declaration statement)* The semantics of a variable declaration statement **var** $x : t$ allows Chalice to continue with an arbitrary value chosen from the semantic domain corresponding to type t. Thus, this semantics introduces non-determinism into the local operational semantics of Chalice.

$$\frac{\mathbf{v} \in \mathbf{D}_t}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\mathbf{var}\ x : t; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)}$$

Definition 3.42 *(Semantics of variable assignment statement)* Recall $[\![\_, \_, \_, \_]\!]$ to denote the totalised expression evaluation function defined in 3.36.

$$\frac{[\![e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma]\!] = \mathbf{v}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (x := e; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)}$$

Definition 3.43 *(Semantics of field assignment statement)*

$$\frac{\sigma(x) = \mathbf{o} \quad \mathrm{h}_1 = \mathrm{h}[(\mathbf{o}, f) \mapsto \sigma(y)]}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (x.f := y; s)), \tau) \rightharpoonup \mathrm{h}_1, ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)}$$

Definition 3.44 *(Semantics of fold statement)* As the operational semantics of Chalice has no explicit permission-logic incorporated, the semantics of a fold statement, and subsequently also the semantics of an unfold statement, is simply to continue to the next statement.

$$\frac{\sigma(x) = \mathbf{o}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\mathbf{fold}\ \mathbf{acc}(x.p); s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)}$$

Definition 3.45 *(Semantics of unfold statement)*

$$\frac{\sigma(x) = \mathbf{o}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{unfold acc}(x.p); s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)}$$

Definition 3.46 *(Semantics of if-then-else statement)* Recall $[\![\_, \_, \_, \_]\!]$ to denote the totalised expression evaluation function defined in 3.36.

$$\frac{[\![e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma]\!] = \mathrm{True}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (s_1; s)), \tau)}$$

$$\frac{[\![e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma]\!] = \mathrm{False}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (s_2; s)), \tau)}$$

3.2.4 Paired Operational Semantics

This section defines an operational semantics for statements which affect a pair of entities. Assume such a statement to be denoted by s. Furthermore, assume a heap h, a thread entity $\mathbf{e}_\tau$ with configuration $(\mathrm{h}^{\mathrm{old}}, \sigma, (s; s_1))$, i.e. $\mathbf{e}_\tau = ((\mathrm{h}^{\mathrm{old}}, \sigma, (s; s_1)), \tau)$, as well as an entity $\mathbf{e}_l$. Statement s then leads the heap h, the thread entity $\mathbf{e}_\tau$ and the entity $\mathbf{e}_l$ to a new heap h_1, a new thread entity $\mathbf{e}'_\tau$ and a new entity $\mathbf{e}'_l$ if the subsequently inductively defined semantic function $\rightarrow$ maps the tuple $(\mathrm{h}, (\mathbf{e}_\tau, \mathbf{e}_l))$ to $(\mathrm{h}_1, (\mathbf{e}'_\tau, \mathbf{e}'_l))$. If $\rightarrow$ maps the tuple $(\mathrm{h}, (\mathbf{e}_\tau, \mathbf{e}_l))$ to $(\mathrm{h}_1, (\mathbf{e}'_\tau, \mathbf{e}'_l))$, the notation $\mathrm{h}, \mathbf{e}_\tau | \mathbf{e}_l \rightarrow \mathrm{h}_1, \mathbf{e}'_\tau | \mathbf{e}'_l$ will be used.

Definition 3.47 *(Semantics of fork statement)* The semantics of a fork statement for method m includes taking an arbitrary idle thread τ_1 which subsequently works off m. Moreover, thread τ_1 gets a method pre-state, reflected in $\mathrm{h}_1^{\mathrm{old}}$, which is the encoding of the current runtime configuration, reflected in h. Note that this rule introduces non-determinism into the operational semantics of Chalice as it allows any idle thread to be chosen to work off the forked method.

$$\frac{\mathrm{h}_1^{\mathrm{old}} = \mathrm{h} \quad \sigma(y) = \mathbf{o} \quad \sigma_1 = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \mathbf{o}]\overline{[x_i \mapsto \sigma(z_i)]} \quad s_1 = \mathrm{Body}(m)}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{fork } x := y.m(\overline{z_i}); s)), \tau) | (\mathrm{idle}, \tau_1) \rightarrow \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau) | ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)}$$

Definition 3.48 *(Semantics of join statement)*

$$\frac{\sigma(x) = \tau_1}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{join } y := x; s)), \tau) | ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, \textbf{return } z), \tau_1) \rightarrow \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[y \mapsto \sigma_1(z)], s), \tau) | (\mathrm{trash}, \tau_1)}$$

Definition 3.49 *(Semantics of new statement)* The semantics of a new statement includes taking an arbitrary free object **o** and then to remove **o** from the set of free objects. Note that this rule introduces non-determinism into the semantics of Chalice as it allows for any free object to be chosen.

$$\frac{cls(\mathbf{o}) = c \quad \overline{f_i} = fields(c) \quad \mathrm{h}_1 = \mathrm{h}[\overline{(\mathbf{o}, f_i) \mapsto zero(f_i)}]}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (x := \textbf{new}\ c; s)), \tau) | (\mathrm{free}, \mathbf{o}) \rightarrow \mathrm{h}_1, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{o}], s), \tau) | (\mathrm{alloc}, \mathbf{o})}$$

3.2.5 Operational Semantics

This section defines an operational semantics on the level of runtime configurations $\mathbf{c} = (\mathbf{h}, \mathbf{r})$. A runtime configuration $(\mathbf{h}, \mathbf{r})$ can transition to a runtime configuration $(\mathbf{h}_1, \mathbf{r}_1)$ if $(\mathbf{h}_1, \mathbf{r}_1)$ is in the reflexive and transitive closure of the subsequently inductively defined semantic function $\rightsquigarrow$. If $\rightsquigarrow$ maps the tuple $(\mathbf{h}, \mathbf{r})$ to $(\mathbf{h}_1, \mathbf{r}_1)$, the notation $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$ will be used. $\rightsquigarrow$ is defined by the following two rules.

$$\frac{\mathbf{r}[\tau] = \mathbf{e}_\tau \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \mathrm{h}, \mathbf{e}_\tau \rightharpoonup \mathrm{h}_1, \mathbf{e}'_\tau \quad \mathbf{h}_1 = \lfloor \mathrm{h}_1 \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto \mathbf{e}'_\tau]}$$

$$\frac{\mathbf{r}[\tau] = \mathbf{e}_\tau \quad \mathbf{r}[l] = \mathbf{e}_l \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \mathrm{h}, \mathbf{e}_\tau | \mathbf{e}_l \rightarrow \mathrm{h}_1, \mathbf{e}'_\tau | \mathbf{e}'_l \quad \mathbf{h}_1 = \lfloor \mathrm{h}_1 \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto \mathbf{e}'_\tau][l \mapsto \mathbf{e}'_l]}$$

Chapter 4

Encoding Chalice into Viper

This chapter defines an encoding of Chalice into Viper. For each syntactic Chalice domain introduced in chapter 3.1, a syntactic function is specified which maps a Chalice language construct to a Viper language construct. All encodings are such that the Chalice methodology given in chapter 4 of [10] is preserved. Following the encoding of syntactic domains, an encoding of the semantic domains will be defined.

The following assumptions are made about the Chalice program to be encoded:

- All field identifiers, all function identifiers, all predicate identifiers and all method identifiers of all classes in the Chalice program are unique with respect to the whole program. This could be established, for example, by prepending the class name to each of the identifiers in the input program.
- A field access is always via *this*: *this.f* instead of just *f*.

Contents

4.1 Syntactic Domains

Definition 4.1 *(Encoding of expressions)* The translation of Chalice expressions into Viper is given by the syntactic function $[\![\_]\!]$. The inductive definition of $[\![\_]\!]$ can be found in table 4.1. (Note that the symbol used for the encoding function will subsequently be overloaded by the encoding definitions of the remaining syntactic Chalice domains. However, the actual encoding function at hand can be recognized by considering the type of the input.)

$$
\begin{array}{lcl}
[\![null]\!] & = & null \\
[\![b]\!] & = & b \\
[\![n]\!] & = & n \\
[\![x]\!] & = & x \\
[\![x.f]\!] & = & [\![x]\!].f \\
[\![\text{- } e]\!] & = & \text{- } [\![e]\!] \\
[\![e_1 \text{ + } e_2]\!] & = & [\![e_1]\!] \text{ + } [\![e_2]\!] \\
[\![e_1 \text{ / } e_2]\!] & = & [\![e_1]\!] \text{ / } [\![e_2]\!] \\
[\![\neg\, e]\!] & = & \neg\, [\![e]\!] \\
[\![e_1 = e_2]\!] & = & [\![e_1]\!] = [\![e_2]\!] \\
[\![x.fun(\overline{e_i})]\!] & = & fun([\![x]\!], \overline{[\![e_i]\!]}) \\
[\![\mathbf{old}(e)]\!] & = & \mathbf{old}([\![e]\!])
\end{array}
$$

Table 4.1: Encoding Chalice expressions into Viper

This encoding warrants discussion as to why the Chalice methodology of [10] is preserved. In Chalice, a field lookup $x.f$, for example, requires to assert to have enough permission to lookup field f of the object behind x. A corresponding assertion is not directly represented in the encoding specified in table 4.1. However, the semantics of Viper defined in chapter 2.2 is such that the field lookup first checks whether non-zero permission is available for the field. If this check fails, a permission-error is generated. This error then propagates to the level of traces where an error-trace is generated. This error-trace then leads the verification of the encoded program to fail. Thus, the methodology of [10] is preserved. Furthermore note that a function call in Chalice is bound to an object as Chalice functions are properties of Chalice classes. As Viper has no notion of a class, a Chalice function call (bound to an object) is modelled as a Viper function call (not bound to an object but) expecting a reference-type variable as its first argument which represents *this* from Chalice. Thus, function calls in Viper are then relative to a reference-type variable which previously was created by the translation to correspond to a class object in Chalice. The case where Chalice evaluates a function call to $\mathbf{x}_{\text{null}}$, which happens if the receiver of the function call is null, is matched in Viper as the encoding of statements, given in 4.3, looks

ahead into expressions and creates a non-null assertion for every function-call expression. An approach analogous to function calls will subsequently be used in the translation of Chalice predicate accesses as well as Chalice method calls, both of which are bound to objects in Chalice.

Definition 4.2 *(Encoding of assertions)* The translation of Chalice assertions into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.2.

$[\![e]\!]$	=	$[\![e]\!]$
$[\![\mathbf{acc}(x.joinable)]\!]$	=	$\mathbf{acc}([\![x]\!].recv, 1/1)$ $\overline{\&\& \mathbf{acc}([\![x]\!].arg_i, 1/1)}$
$[\![\mathbf{acc}(x.f, n)]\!]$	=	$\mathbf{acc}([\![x.f]\!], [\![n]\!]/100)$
$[\![\mathbf{acc}(x.p)]\!]$	=	$\mathbf{acc}(p([\![x]\!]), 1/1)$
$[\![e \rightarrow a]\!]$	=	$[\![e]\!] \rightarrow [\![a]\!]$
$[\![a_1 \,\&\&\, a_2]\!]$	=	$[\![a_1]\!] \,\&\&\, [\![a_2]\!]$

Table 4.2: Encoding Chalice assertions into Viper

The methodology of Chalice presented in [10] dictates that a conjunct of assertions is operationally equivalent to a sequence of the assertions. The semantics of Viper is such that inhaling and exhaling conjuncts of assertions is reduced to a sequence of inhales and exhales of assertions. As all validations of assertions in Chalice are reduced to either exhaling or inhaling the assertion in Viper, the methodology of Chalice is preserved. The only translation that warrants discussion is the translation of joinable-access assertions. A joinable-access assertion occurs in the context of fork and join statements and is expanded into a conjunct of access assertions to all fields which are used to model the Chalice methodology of forks and joins. The meaning of each field is explained in the next section which deals with the encoding of Chalice statements. Expanding a joinable-access assertion in this manner guarantees a subsequent join to have all the permissions required to gather the information needed for an inhale of the forked methods post-condition. More details regarding forks and joins follow in the discussion of the encoding of statements.

Definition 4.3 *(Encoding of statements)* The translation of Chalice statements into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.3.

According to Chalice, statement x := **new** c has to lead to a heap containing for each field of class c an initial default value and to a permission-mask with full access to each field of class c. The translation into Viper first creates a reference-type variable mentioning each field of class c. The semantics of Viper is such that a new statement leads to full permissions for each of the fields mentioned in the statement. As the encoding furthermore leads to a

$[\![\textbf{var } x : t]\!]$	=	$\textbf{var } [\![x]\!] : [\![t]\!]$
$[\![x := e]\!]$	=	$\overline{\textbf{assert } [\![x_i]\!] \mathrel{!}= [\![null]\!]};$ $[\![x]\!] := [\![e]\!]$
$[\![x.f := y]\!]$	=	$[\![x.f]\!] := [\![y]\!]$
$[\![\textbf{return } x]\!]$	=	ε
$[\![x := \textbf{new } c]\!]$	=	$[\![x]\!] := \textbf{new}(\overline{f_i});$ $\overline{[\![x]\!].f_i := zero(f_i)}$
$[\![\textbf{fold acc}(x.p)]\!]$	=	$\textbf{assert } [\![x]\!] \mathrel{!}= [\![null]\!];$ $\textbf{fold acc}(p([\![x]\!]), 1/1)$
$[\![\textbf{unfold acc}(x.p)]\!]$	=	$\textbf{assert } [\![x]\!] \mathrel{!}= [\![null]\!];$ $\textbf{unfold acc}(p([\![x]\!]), 1/1)$
$[\![\textbf{fork } x := y.m(\overline{z_i})]\!]$	=	$\textbf{exhale } [\![\text{Pre}(m)]\!][[\![y]\!]/[\![this]\!]]\overline{[[\![z_i]\!]/[\![x_i]\!]]};$ $\textbf{var } [\![x]\!] : Ref;$ $[\![x]\!] := \textbf{new}(recv, \overline{arg_i});$ $\overline{[\![x]\!].recv := [\![y]\!];}$ $\overline{[\![x]\!].arg_i := [\![z_i]\!];}$
$[\![\textbf{join } y := x]\!]$	=	$\textbf{assert acc}([\![x]\!].recv, 1/1)\ \overline{\&\& \textbf{acc}([\![x]\!].arg_i, 1/1)};$ $\textbf{var } [\![y]\!] : [\![\text{Type}(m)]\!];$ $\textbf{inhale } [\![\text{Post}(m)]\!][[\![x]\!].recv/[\![this]\!]]\overline{[[\![x]\!].arg_i/[\![x_i]\!]]}$ $\overline{[[\![y]\!]/[\![z]\!]];}$ $\overline{\textbf{exhale acc}([\![x]\!].recv, 1/1)\ \&\& \textbf{acc}([\![x]\!].arg_i, 1/1)};$
$[\![\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2]\!]$	=	$\overline{\textbf{assert } [\![x_i]\!] \mathrel{!}= [\![null]\!];}$ $\textbf{if } [\![e]\!] \textbf{ then } [\![s_1]\!] \textbf{ else } [\![s_2]\!]$
$[\![s_1;\ s_2]\!]$	=	$[\![s_1]\!];\ [\![s_2]\!]$

Table 4.3: Encoding Chalice statements into Viper

sequence of field-assignments assigning a default zero value of the appropriate type (with respect to the field to be assigned), the translation respects the methodology of Chalice.

As can be seen in table 4.3, variable assignment statements as well as if-then-else statements are preceded by a sequence of asserts asserting a variable not to be null. This sequence has to be computed by the encoding and results from looking ahead into expressions and searching for function-calls. Each receiver of such a function call has to be asserted for non-null in Viper in order to match a Chalice error resulting from a null-receiver.

Fork and join statements, as dictated by Chalice, are translated into an exhale of the forked methods m's pre-condition and into an inhale of m's post-condition. The encoding given in table 4.3 works as follows. A fork statement constructs a record with fields *recv* and $\overline{arg_i}$. Note that the fields $\overline{arg_i}$ have to be of appropriate types with respect to m and the arguments m expects. The encoding thus has to create for each Chalice type an *arg* field

and pick the type-correct ones during the translation of a fork. Recall that a post-condition is a function of *this*, the formal arguments $\overline{x_i}$ and the return variable z. As such it is sufficient to record the receiver of the method-call as well as the arguments to the method-call to later properly inhale m's post-condition. At the join site, *this* is then replaced by the receiver *recv*, which was recorded at the fork site, and the formal argument variables $\overline{x_i}$ are replaced by the argument variables $\overline{arg_i}$, which also were recorded at the fork site. Moreover, the result variable y substitutes for the formal return variable z. After the inhale of m's post-condition, the permission to the receiver field *recv* and to the argument fields $\overline{arg_i}$ of the record are exhaled. This prevents a second join on the same object. Note that at the join-site, m is available to the translation as Chalice tokens carry along the identifier of the method that was forked.

Definition 4.4 *(Encoding of fields)* The translation of Chalice class fields into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.4.

$$[\![\textbf{var}\ f : t]\!] \quad = \quad \textbf{field}\ f : [\![t]\!]$$

Table 4.4: Encoding Chalice class fields into Viper

This encoding works, as an identifier of a Chalice class field is assumed to be unique with respect to the whole Chalice program under translation.

Definition 4.5 *(Encoding of functions)* The translation of Chalice functions into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.5.

$$\begin{array}{l}[\![\textbf{function}\ fun(\overline{x : t}) : t\ \ \overline{\textbf{requires}\ a}\ \ \overline{\textbf{ensures}\ a}\ \ \{\ e\ \}]\!] \ = \\ \textbf{function}\ fun([\![this]\!] : Ref,\ \overline{[\![x]\!] : [\![t]\!]}) : [\![t]\!]\ \ \overline{\textbf{requires}\ [\![a]\!]}\ \ \overline{\textbf{ensures}\ [\![a]\!]}\ \{\ [\![e]\!]\ \}\end{array}$$

Table 4.5: Encoding Chalice functions into Viper

Note that this translation works, as a function name is assumed to be unique with respect to the whole Chalice program. Furthermore it is worth to note that the resulting Viper function has at least the formal parameter *this* of type *Ref*. This is due to Chalice function calls being bound to objects. At the call site of a function, the Viper reference-type variable corresponding to the Chalice object is then passed as the first argument to the function, as can be seen in table 4.1.

Definition 4.6 *(Encoding of predicates)* The translation of Chalice predicates into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.6.

$$[\![\textbf{predicate } p \{ a \}]\!] \quad = \quad \textbf{predicate } p([\![this]\!] : Ref) \{ [\![a]\!] \}$$

Table 4.6: Encoding Chalice predicates into Viper

Note that also this translation assumes Chalice predicate names to be unique with respect to the whole Chalice program under translation. Furthermore note that the resulting Viper predicate always has exactly *this* of type *Ref* as a formal argument. This is due to Chalice predicates being defined with respect to a class and the predicate usage hence being bound to an object. As can be seen in the translation of assertions in table 4.2, the use of a Chalice predicate is translated into a call of the predicate, with the Viper reference-type variable corresponding to the Chalice object, on which the predicate was called, passed as the one argument.

Definition 4.7 *(Encoding of methods)* The translation of Chalice methods into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.7.

$$\begin{aligned}&[\![\textbf{method } m(\overline{x : t}) \textbf{ returns } \overline{x : t} \ \overline{\textbf{requires } a} \ \overline{\textbf{ensures } a} \ \{ s \}]\!] \quad = \\ &\quad \textbf{method } m([\![this]\!] : Ref, \overline{[\![x]\!] : [\![t]\!]}) : \textbf{returns } \overline{[\![x]\!] : [\![t]\!]} \\ &\quad \overline{\textbf{requires } [\![a]\!]} \ \overline{\textbf{ensures } [\![a]\!]} \ \{ [\![s]\!] \}\end{aligned}$$

Table 4.7: Encoding Chalice methods into Viper

Also here it is assumed that Chalice method names are unique with respect to the whole Chalice program. Furthermore, as in the case of functions and predicates, the resulting method has at least the formal argument *this* of type *Ref*, as Chalice method calls are bound to objects. As explicit method calls are not supported by the semantics presented in this work, this translation is ahead of time. A future semantics will be able to model a method call by passing the Viper reference-type variable corresponding to the receiver of the Chalice method call as the first argument to the method.

Definition 4.8 *(Encoding of classes)* The translation of Chalice classes into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.8.

$$\begin{aligned}&[\![\textbf{class } c \{ \overline{field}; inv; \overline{funct}; \overline{pred}; \overline{meth} \}]\!] \quad = \\ &\quad \overline{[\![field]\!]}; \overline{[\![funct]\!]}; \overline{[\![pred]\!]}; \overline{[\![meth]\!]}\end{aligned}$$

Table 4.8: Encoding Chalice classes into Viper

Note that the translation of a class excludes the translation of the class invariant. This is due to the semantics presented in this work not supporting any use case of class invariants.

Definition 4.9 *(Encoding of Chalice programs)* The translation of a Chalice program into Viper is given by the inductive syntactic function $[\![\_]\!]$ defined in table 4.9.

$$[\![\overline{class}]\!] \quad = \quad \overline{[\![class]\!]}$$

Table 4.9: Encoding a Chalice program into Viper

4.2 Semantic Domains

As can be seen from the definition of the Chalice statement encoding given in table 4.3, the allocation of a new object in Chalice is matched with the allocation of a new object in Viper and the fork of a method in Chalice is matched with an allocation of an object in Viper. The encoding of semantic domains, presented subsequently, requires a means of relating the object allocated in Chalice when executing a new-statement with the object allocated in Viper and relating the thread identifier chosen by Chalice during the executing of a fork-statement with the object allocated in Viper. By definition given in 2.13, the set of objects $\mathbf{O}$ is infinite. Thus, the means of relating Chalice objects with Viper objects and Chalice thread-identifiers with Viper objects can be given by an assumed bijection from $\mathbf{O} \cup \Gamma$ to $\mathbf{O}$.

Definition 4.10 *(Bijection translating between Chalice and Viper)* The translation of Chalice objects into Viper objects and the translation of Chalice thread-identifiers into Viper objects is given by the assumed bijective function $\gamma : \mathbf{O} \cup \Gamma \mapsto \mathbf{O}$. The inverse function of γ will be denoted as γ^{-1}.

Definition 4.11 *(Encoding of values)* Chalice values $\mathbf{v}$ are encoded into Viper values according to the semantic function $[\![\_]\!]$ defined below. (Note that also the encoding functions of semantic domains are notationally overloaded. As in the case of the overloaded syntactic functions from the previous chapter, the semantic functions too can be recognized by considering the type of the input.)

$$[\![\mathbf{v}]\!] = \begin{cases} \gamma(\mathbf{v}) & \text{if } \mathbf{v} \in \mathbf{O} \cup \Gamma \\ \mathbf{v} & \text{else} \end{cases}$$

Note that the value encoding function $[\![\_]\!]$ is bijective.

Definition 4.12 *(Encoding of stores)* Chalice stores σ are encoded into Viper stores according to the semantic function $[\![\_]\!]$ defined below.

$$[\![\sigma]\!]([\![x]\!]) = [\![\sigma(x)]\!]$$

Definition 4.13 *(Encoding of permission masks)* Chalice permission masks $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$ are encoded into Viper permission masks according to the semantic function $[\![\_]\!]$ defined below.

$$[\![\pi]\!](\mathbf{o}, f) = \begin{cases} [\![\frac{\pi^{\mathbf{F}}(\gamma^{-1}(\mathbf{o}), \mathit{joinable})}{100}]\!] & \text{if } f \in \{\, \mathit{recv}, \overline{\mathit{arg}_i} \,\} \\ [\![\frac{\pi^{\mathbf{F}}(\gamma^{-1}(\mathbf{o}), f)}{100}]\!] & \text{else} \end{cases}$$

$$[\![\pi]\!](\mathbf{p}) = [\![\frac{\pi^{\mathbf{P}}(\gamma^{-1}(\mathbf{o}), p)}{100}]\!] \text{ for } (\mathbf{o}, p) \in \mathbf{O} \times P \text{ s.t. } \mathbf{p} = \mathbf{P}_2(p, \mathbf{o})$$

Definition 4.14 *(Encoding of heaps)* Recall that a Chalice heap h is always the encoding of a Chalice runtime heap $\mathbf{h}$ and a runtime entity collection $\mathbf{r}$, i.e. $\mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor$. The encoding of Chalice heap $\mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor$ into a Viper heap is thus based on $\mathbf{h}$ and $\mathbf{r}$ and given by the following function $[\![\_]\!]$.

$$\begin{aligned} [\![\mathrm{h}]\!](\mathbf{o}, f) &= [\![\lceil \mathbf{h}, \mathbf{r} \rfloor]\!](\mathbf{o}, f) \\ &= \begin{cases} [\![\sigma(\mathit{this})]\!] & \text{if } f = \mathit{recv} \ \wedge\ ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r} \text{ s.t. } \tau = \gamma^{-1}(\mathbf{o}) \\ [\![\sigma(x_i)]\!] & \text{if } f = \mathit{arg}_i \ \wedge\ ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r} \text{ s.t. } \tau = \gamma^{-1}(\mathbf{o}) \\ [\![\mathbf{h}(\gamma^{-1}(\mathbf{o}), f)]\!] & \text{else} \end{cases} \end{aligned}$$

Chapter 5

Soundness of Encoding

Informally, the encoding of Chalice into Viper presented in chapter 4 is sound, whenever the verification of a translated Chalice program within Viper implies the verification of the Chalice program with respect to the semantics of Chalice. To formalise this idea, this chapter first defines the judgment of a Chalice program to be verified and a Chalice runtime configuration to be valid. Based on these judgments, soundness of the encoding can then formally be stated.

Contents

5.1 Verified Chalice Programs

The notion of a Chalice program *prog* to be *verified* is based on the semantics of Viper and essentially expresses the encoding of *prog* to be a verified Viper program.

Definition 5.1 *(Verified Chalice Program)* The judgment of a Chalice program *prog* to be *verified* is expressed through the predicate $VP(prog)$ and defined as follows: $VP(prog) \Leftrightarrow \forall m \in prog, \forall \mathbf{h}, \sigma, \pi.\ (\mathbf{h}, \sigma, \pi) \models [\![\text{Pre}(m)]\!]$ $\Rightarrow \text{V}[\{\ ([\![\text{Body}(m)]\!], \Lambda[(\mathbf{h}, \sigma, \pi)])\ \}] \models [\![\text{Post}(m)]\!]$.

5.2 Valid Runtime Configurations

As a preliminary to the formal statement of the encoding of Chalice into Viper to be sound, the notion of a valid Chalice runtime is introduced. This notion reflects (in strongly simplified parlance) that each step taken by Chalice during the execution of a program can be matched within Viper by one or possibly multiple steps during the evaluation of the encoded program. The notion of a valid Chalice runtime requires the judgment of a sequence of Chalice permission masks to be valid with respect to a Chalice runtime configuration. Thus, before defining valid runtime configurations, the validity of a sequence of Chalice permission masks is introduced.

Definition 5.2 *(Valid Permission Mask)* The judgment of a Chalice permissions mask $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$ to be *valid* is expressed through predicate $\models \pi$ and defined as follows: $\models \pi \Leftrightarrow \forall(\mathbf{o},f).\ \pi^{\mathbf{F}}(\mathbf{o},f) \leq 100\ \wedge\ \forall(\tau,f).\ \pi^{\mathbf{F}}(\tau,f) \leq 100$.

Definition 5.3 *(Valid Permission Masks)* The judgment of a sequence of Chalice permissions masks $\overline{\pi_\tau}$ to be *valid* with respect to a Chalice runtime entity collection $\mathbf{r}$ is expressed through predicate $\models_{\mathbf{r}} \overline{\pi_\tau}$ and defined as follows: $\models_{\mathbf{r}} \overline{\pi_\tau} \ \Leftrightarrow\ \models\ (\sum_{\mathbf{e}_\tau \in \mathbf{r}} \pi_\tau) + (\sum_{\mathbf{e}_\mathbf{o} \in free(\mathbf{r})} \sum_{f \in fields(\mathbf{e}_\mathbf{o})} (\varnothing^{\mathbf{F}}[(\mathbf{o},f)\ \mapsto\ 100], \varnothing^{\mathbf{P}})) +$ $(\sum_{\mathbf{e}_\tau \in idle(\mathbf{r})} (\varnothing^{\mathbf{F}}[(\tau, joinable) \mapsto 100], \varnothing^{\mathbf{P}}))$.

Definition 5.4 *(Read-only equivalence of stores)* The judgment of Chalice stores σ and σ_1 to be *read-only equivalent* reflects σ and σ_1 to be equal on variables which are read only in a Chalice method. The judgment is expressed through predicate $\sigma \overset{\text{ro}}{\equiv} \sigma_1$ and defined as follows: $\sigma \overset{\text{ro}}{\equiv} \sigma_1 \ \Leftrightarrow\ \sigma(this) = \sigma_1(this)\ \wedge\ \sigma(meth) = \sigma_1(meth)\ \wedge\ \forall x \in \text{Param}(\sigma(meth)).\, \sigma(x) = \sigma_1(x)$

Definition 5.5 *(Heap equivalence on permission masks)* The judgment of a Chalice runtime heap $\mathbf{h}$ and a Chalice runtime heap $\mathbf{h}_1$ to be equal on permission mask $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$ is expressed through predicate $\mathbf{h} \overset{\pi}{\equiv} \mathbf{h}_1$ and defined as follows: $\mathbf{h} \overset{\pi}{\equiv} \mathbf{h}_1\ \Leftrightarrow\ \forall(\mathbf{o},f).\ \pi^{\mathbf{F}}(\mathbf{o},f) > 0 \Rightarrow \mathbf{h}(\mathbf{o},f) = \mathbf{h}_1(\mathbf{o},f)$.

Definition 5.6 *(Valid Runtime Configuration)* The judgment of a Chalice runtime configuration $(\mathbf{h}, \mathbf{r})$ to be *valid* with respect to a Chalice program *prog* is expressed through the predicate $VR(prog, \mathbf{h}, \mathbf{r})$ and defined as follows:

$$
\begin{aligned}
VR(prog, \mathbf{h}, \mathbf{r}) \;\Leftrightarrow\; & \exists \overline{(\pi_\tau^{\text{old}}, \pi_\tau)}. && (5.1)\\
& \models_{\mathbf{r}} \overline{\pi_\tau} \wedge && (5.2)\\
& \forall \mathbf{e}_\tau \in idle(\mathbf{r}).\ \pi_\tau = \pi_\varnothing \wedge && (5.3)\\
& \forall((\mathrm{h}^{\text{old}}, \sigma, s), \tau) \in \mathbf{r}. && (5.4)\\
& \quad (\llbracket \mathrm{h}^{\text{old}} \rrbracket, \llbracket \sigma^{\text{old}} \rrbracket, \llbracket \pi_\tau^{\text{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge && (5.5)\\
& \quad \exists(\llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\text{old}} \rrbracket, \llbracket \sigma^{\text{old}} \rrbracket, \llbracket \pi_\tau^{\text{old}} \rrbracket)])\ \}], && (5.6)\\
& \quad \exists \mathbf{h}_2 \overset{\pi_\tau}{\equiv} \mathbf{h}.\ (\llbracket \lceil \mathbf{h}_2, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda) \wedge && (5.7)\\
& \qquad\qquad \mathrm{V}[\{\ (\llbracket s \rrbracket, \lambda)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket && (5.8)\\
& \quad \text{where}\ \ m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),\\
& \qquad\qquad \sigma^{\text{old}} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}
\end{aligned}
$$

In (5.1), the judgment asks for each thread $\tau \in \mathbf{r}$ for the existence of a tuple of permission-masks $(\pi_\tau^{\text{old}}, \pi_\tau)$ where π_τ^{old} is the permission-mask associated with the method pre-state and π_τ is the mask associated with the current state of τ while working off a method m (if any). (5.2) imposes the distribution of the current permissions to the threads, reflected in $\overline{\pi_\tau}$, to be valid and (5.3) requires idle threads not to use any permissions. (5.4) to (5.8) imposes for all active threads τ, working off method m and with s the statement to be executed next, the following requirement: (5.5) the encoded method pre-state associated with thread τ has to satisfy the encoded precondition of method m and (5.6) there has to be a partial trace in the transitive closure of the Viper set of traces semantics starting with a singleton set of partial traces containing only the partial trace made up of the encoded method pre-state of thread τ and an evaluation context where the encoded body of method m has to be worked off s.t. (5.7) the last configuration of this partial trace is equivalent to the current state of thread τ (modulo fields where thread τ currently has no access to) and (5.8) this partial trace has to go on to complete traces satisfying the encoded postcondition of method m.

5.3 Soundness of Encoding

With all the notions introduced in the previous chapters, soundness of the encoding of Chalice into Viper can now formally be stated as follows.

Corollary 5.7 *(Soundness of the encoding of Chalice into Viper)*

$$
\begin{aligned}
&\forall \mathit{prog}, m, \mathbf{h}, \mathbf{r}, \mathbf{h}_1, \mathbf{r}_1.\\
&\quad \mathit{VP}(\mathit{prog}) \wedge m \in \mathit{prog} \wedge && (5.9)\\
&\quad \overline{\mathbf{r}[\tau_1 \neq \tau] = (\mathit{idle}, \tau_1)} \wedge \mathbf{r}[\tau] = ((\lceil \mathbf{h}, \mathbf{r} \rfloor, \sigma[\mathit{meth} \mapsto m], \mathrm{Body}(m)), \tau) \wedge && (5.10)\\
&\quad \mathit{VR}(\mathit{prog}, \mathbf{h}, \mathbf{r}) \wedge \mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1 \wedge && (5.11)\\
&\quad \mathbf{r}_1[\tau] = ((\lceil \mathbf{h}, \mathbf{r} \rfloor, \sigma_1, \mathbf{return}\, x), \tau) && (5.12)\\
&\quad \Rightarrow\\
&\quad \exists \pi. (\lceil \mathbf{h}, \mathbf{r} \rfloor, \lceil \mathbf{h}_1, \mathbf{r}_1 \rfloor, \sigma_1, \pi) \models \mathrm{Post}(\sigma_1(\mathit{meth})) && (5.13)
\end{aligned}
$$

Proof Follows as a corollary from lemma 5.8 and theorem 5.9, both stated subsequently. □

Corollary 5.7 expresses the following: whenever it holds that (5.9) a Chalice program *prog* is a valid program and m is a method of *prog* and (5.10) the runtime entity collection $\mathbf{r}$ has all threads τ_1 idle except for thread τ setup with method pre-state $\lceil \mathbf{h}, \mathbf{r} \rfloor$ and working off method m and (5.11) $\mathbf{h}$ and $\mathbf{r}$ form a valid runtime with respect to *prog* and the Chalice operational semantics transitions from $\mathbf{h}$ and $\mathbf{r}$ to $\mathbf{h}_1$ and $\mathbf{r}_1$ and (5.12) in $\mathbf{r}_1$ thread τ has worked off method m then (5.13) there is a permission mask π for thread τ with which the current state of thread τ can satisfy the postcondition of m.

Lemma 5.8, used to prove the soundness-of-encoding corollary 5.7 and stated next, demonstrates that modelling of assertions is sound.

Lemma 5.8 *(Soundness of modelling assertions)*

$$
\begin{aligned}
&\forall \mathit{prog}, m.\ \mathit{VP}(\mathit{prog}) \wedge m \in \mathit{prog} \wedge \sigma^{\mathrm{old}} = \sigma_0[\mathit{meth} \mapsto m] \wedge && (5.14)\\
&\quad (\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi^{\mathrm{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge && (5.15)\\
&\quad (\llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi^{\mathrm{old}} \rrbracket)])\ \}] \ \wedge\ \llbracket a \rrbracket \in \llbracket s \rrbracket \wedge && (5.16)\\
&\quad (\llbracket \mathrm{h} \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi \rrbracket) = \mathrm{LAST}(\lambda) \ \wedge\ \sigma \overset{\mathrm{ro}}{\equiv} \sigma^{\mathrm{old}} \ \wedge\ \mathrm{h}_1^{\mathrm{old}} \overset{\pi^{\mathrm{old}}}{\equiv} \mathrm{h}^{\mathrm{old}} \ \wedge\ \mathrm{h}_1 \overset{\pi}{\equiv} \mathrm{h} \wedge && (5.17)\\
&\quad \lambda \models \llbracket a \rrbracket && (5.18)\\
&\quad \Rightarrow\\
&\quad (\mathrm{h}_1^{\mathrm{old}}, \mathrm{h}_1, \sigma, \pi) \models a && (5.19)
\end{aligned}
$$

Proof By structural induction on the Chalice assertion a. For full proof see B.9. □

Theorem 5.9, used to prove the soundness-of-encoding corollary 5.7 and stated next, demonstrates that validity of runtime configurations is preserved by the operational semantics of Chalice.

Theorem 5.9 *(Preservation of runtime configuration validity by Chalice)*
$\forall \mathit{prog}, \mathbf{h}, \mathbf{r}, \mathbf{h}_1, \mathbf{r}_1.\ \mathit{VP}(\mathit{prog}) \wedge \mathit{VR}(\mathit{prog}, \mathbf{h}, \mathbf{r}) \wedge \mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1 \ \Rightarrow\ \mathit{VR}(\mathit{prog}, \mathbf{h}_1, \mathbf{r}_1)$

Proof By induction on the length of the execution leading from $(\mathbf{h}, \mathbf{r})$ to $(\mathbf{h}_1, \mathbf{r}_1)$ as given by the rules of $\rightsquigarrow$. For full proof see B.11. □

Chapter 6

Conclusion and Future Work

Contents

6.1 Conclusion

This thesis presented a first formal semantics for a substantial subset of the Viper language. The design of the semantics was strongly influenced by the presence of **old**-expressions whose evaluation, in essence, requires a semantics to keep a history of states. For this, the semantics of Viper defined the notion of a *trace* (2.28) as a sequence of states indexable via state-labels. Such a trace then allows for an expression being evaluated in any state the trace met during the verification of a method and thus also allows for the evaluation of **old**-expressions (2.47) (2.48).

The semantics presented also supports language constructs dealing with *predicates*. As predicates are recursive definitions of assertions, they can either be treated as entities to which a state can have access to (the *iso-recursive* treatment) or can be identified with the full unrolling of their definition (the *equi-recursive* treatment). The semantics of Viper is a *hybrid*: predicates are treated iso-recursively until *completeness* requires an unrolling. This combination is, for example, reflected in the semantics of exhaling permissions to a predicate (2.73). As such, the semantics of Viper is theoretically complete but non-implementable in case predicates with an infinite unrolling are present.

Moreover, the semantics of Viper is in parts *non-deterministic* as it allows, for example, a declared variable to be initialised with an arbitrary value chosen from the domain of its type (2.64). However, this non-determinism has no effect on whether a program verifies: the verification of a program was defined to be a judgment about all possible evaluations of the program (2.89). As this includes all of the non-determinism possibly generated by the semantics of a statement, overall verification was proven to be deterministic (2.84).

This thesis also presented an encoding of Chalice into Viper and proved this encoding to be sound (5.7): whenever the encoding of a Chalice program verifies with respect to the semantics of Viper then the Chalice program verifies with respect to the semantics of Chalice. Proving soundness required to match within Viper every step previously taken by Chalice. A challenge arouse from both Chalice as well as Viper to allocate new objects non-deterministically. Moreover, Chalice also selects threads non-deterministically which, by encoding, leads to Viper allocating a new object - non-deterministically. To obtain a sound encoding required the encoding to uniquely translate Chalice objects into Viper objects and to uniquely translate Chalice threads into Viper objects. Realising the sets of objects to be countably infinite then allowed to construct this translation from an assumed bijective function mapping from the union of objects and threads in Chalice to objects in Viper. Soundness could then be proven by first encoding the step of Chalice into Viper, observing a potentially non-deterministic

set of evaluations in Viper and then by finally pulling out the one string of evaluation that matched, under the assumed translation, the step taken in Chalice.

6.2 Future Work

The Viper semantics presented does currently not support the *packing of wands*. Packing a wand involves the computation of a *footprint* which satisfies the semantics of a wand [4]. Sound algorithms computing such a footprint exist [11] but incorporating an algorithm into the semantics of Viper seems arbitrary until it is proven that such an algorithm computes a footprint which, in a certain sense, is *optimal*. However, to this day, the nature of a wand-footprint is not well understood and questions like *uniqueness* or *minimality* of a footprint have not formally been answered yet. As the existence of an optimal footprint is not guaranteed by theory and the incorporation of an arbitrary footprint computation into Viper is out of question, the development of a semantics for packing wands is work for a future project.

Moreover, the Viper semantics presented in this thesis does not support custom domains, triggers for quantifiers and goto-statements. The extension of the semantics to include these language constructs is all potential work for a future project.

Similarly, the subset of Chalice that was encoded into Viper and whose encoding was proven sound within this thesis can be extended until all of Chalice is supported. Major elements of Chalice currently missing support are *object-monitors* as well as *abstract read permissions* [12].

At last, the conceptual complexity as well as the notational complexity of the Viper semantics, the Chalice semantics and mostly the notion of soundness and the proof thereof call for a mechanisation within a suitable proof-assistant like Coq [13] or Isabelle/HOL [14]. A first pay-off of such a mechanisation would be the guarantee that soundness is indeed proven which, due to the level of details involved, is almost impossible to obtain from a pen-and-paper proof. A second interesting pay-off would result from the proof-assistants being able to generate executable code which is a provably correct representation of the theory specified within the proof-assistant. Thus, if the semantics of Viper would be mechanised, if the semantics of Chalice would be mechanised and if the encoding of Chalice into Viper would be mechanised then a proof-assistant would be able to generate a provably correct executable which encodes Chalice into Viper.

Appendix A

Auxiliary Functions

Algorithm 1 Local Permission Collection Function $\mathbf{Q}$

1: **function** $\mathbf{Q}(a : A,\ \lambda : \Lambda,\ isIso : \mathbf{B}) : \Pi \cup \mathbf{X}$
2: // Assume: $\lambda = \lambda_1{:}(\mathbf{h},\ \sigma,\ \pi)$
3: **if** $a \equiv \mathbf{acc}(x.f,\ q)$ **then**
4: **if** $\sigma(x) = \text{null}$ **then**
5: **return** $\mathbf{x}_{\text{null}}$
6: **else if** $\mathbf{Q}_q < 0$ **then**
7: **return** $\mathbf{x}_{\text{perm}}$
8: **else**
9: **return** $(\varnothing^{\mathbf{F}}[(\sigma(x), f) \mapsto \mathbf{Q}_q], \varnothing^{\mathbf{P}}, \varnothing^{\mathbf{W}})$
10: **end if**
11: **end if**
12: **if** $a \equiv \mathbf{acc}(p(e_1,\ ...,\ e_n), q)$ **then**
13: **if** $\exists i.\ \llbracket e_i, \lambda \rrbracket = \mathbf{x}$ **then**
14: **return** $\mathbf{x}$
15: **else if** $\mathbf{Q}_q < 0$ **then**
16: **return** $\mathbf{x}_{\text{perm}}$
17: **else**
18: // Assume: $\overline{\llbracket e_i, \lambda \rrbracket = \mathbf{v}_i},\ \mathbf{p} = \mathbf{P}_{|p|}(p, \overline{\mathbf{v}_i}),\ \overline{x_i} = \text{FV}(p)$
19: $\pi_1 := (\varnothing^{\mathbf{F}}, \varnothing^{\mathbf{P}}[\mathbf{p} \mapsto \mathbf{Q}_q], \varnothing^{\mathbf{W}})$
20: **if** $isIso$ **then**
21: **return** π_1
22: **else if** $\mathbf{Q}(\text{Body}(p)\overline{[y_i/x_i]}, \lambda_1{:}(\mathbf{h},\ \sigma\overline{[y_i \mapsto \mathbf{v}_i]},\ \pi), isIso) = \mathbf{x}$ **then**
23: **return** $\mathbf{x}$
24: **else**
25: **return** $\pi_1 + \mathbf{Q}(\text{Body}(p)\overline{[y_i/x_i]}, \lambda_1{:}(\mathbf{h},\ \sigma\overline{[y_i \mapsto \mathbf{v}_i]},\ \pi), isIso)$
26: **end if**
27: **end if**
28: **end if**
29: **if** $a \equiv e\ \rightarrow\ a_1$ **then**
30: **if** $\llbracket e, \lambda \rrbracket = \mathbf{x}$ **then**
31: **return** $\mathbf{x}$
32: **else if** $\llbracket e, \lambda \rrbracket = \text{True}$ **then**
33: **return** $\mathbf{Q}(a_1, \lambda, isIso)$
34: **end if**
35: **end if**
36: **if** $a \equiv a_1\ \&\&\ a_2$ **then**
37: **if** $\mathbf{Q}(a_1, \lambda, isIso) = \mathbf{x}$ **then**
38: **return** $\mathbf{x}$
39: **else if** $\mathbf{Q}(a_2, \lambda, isIso) = \mathbf{x}$ **then**
40: **return** $\mathbf{x}$
41: **else**
42: **return** $\mathbf{Q}(a_1, \lambda, isIso) + \mathbf{Q}(a_2, \lambda, isIso)$
43: **end if**
44: **end if**
45: **if** $a \equiv a_1\ -\!\!*\ a_2$ **then**
46: **if** $\mathbf{W}_n(a_1\ -\!\!*\ a_2, \lambda) = \mathbf{x}$ **then**
47: **return** $\mathbf{x}$
48: **else**
49: **return** $(\varnothing^{\mathbf{F}}, \varnothing^{\mathbf{P}}, \varnothing^{\mathbf{W}}[\mathbf{W}_n(a_1\ -\!\!*\ a_2, \lambda) \mapsto 1])$
50: **end if**
51: **end if**
52: **return** $\pi_\varnothing$
53: **end function**

Algorithm 2 Global Permission Collection Function **G**

1: **function** $\mathbf{G}(\pi : \Pi, \lambda : \Lambda) : \Pi \cup \mathbf{X}$
2: // Assume: $\lambda = \lambda_1{:}(\mathbf{h}, \sigma, \pi)$ and $\pi = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}}, \pi^{\mathbf{W}})$
3: **do**
4: $card := |\{ (\mathbf{o}, f) \mid \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \}|$
5: **for each** $\mathbf{p} \in \mathbf{P}$ s.t. $\pi^{\mathbf{P}}(\mathbf{p}) > 0$ **do**
6: // Assume: $p, \overline{\mathbf{v}_i}$ s.t. $\mathbf{p} = \mathbf{P}_{|p|}(p, \overline{\mathbf{v}_i})$ and $\overline{x_i} = \mathrm{FV}(p)$
7: **if** $\mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p)\overline{[y_i / x_i]}, \lambda_1{:}(\mathbf{h}, \sigma\overline{[y_i \mapsto \mathbf{v}_i]}, \pi), \mathrm{False}) = \mathbf{x}$ **then**
8: **return x**
9: **else**
10: $\pi := \pi + \mathbf{Q}^{\mathrm{T}}(\mathrm{Body}(p)\overline{[y_i / x_i]}, \lambda_1{:}(\mathbf{h}, \sigma\overline{[y_i \mapsto \mathbf{v}_i]}, \pi), \mathrm{False})$
11: **end if**
12: **end for each**
13: **while** $card < |\{ (\mathbf{o}, f) \mid \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \}|$
14: **return** π
15: **end function**

Algorithm 3 Permission Collection Function **Z**

```
1: function Z(a : A, h^old : H, h : H, σ : Σ) : Π ∪ X
2:     if a ≡ acc(x.joinable) then
3:         if σ(x) = null then
4:             return x_null
5:         else
6:             return (∅^F[(σ(x), joinable) ↦ 100], ∅^P)
7:         end if
8:     end if
9:     if a ≡ acc(x.f, n) then
10:        if σ(x) = null then
11:            return x_null
12:        else if Z_n < 0 then
13:            return x_perm
14:        else
15:            return (∅^F[(σ(x), f) ↦ Z_n], ∅^P)
16:        end if
17:    end if
18:    if a ≡ acc(x.p) then
19:        if σ(x) = null then
20:            return x_null
21:        else
22:            return (∅^F, ∅^P[(σ(x), p) ↦ 100])
23:        end if
24:    end if
25:    if a ≡ e → a_1 then
26:        if ⟦e, h^old, h, σ⟧ = x then
27:            return x
28:        else if ⟦e, h^old, h, σ⟧ = True then
29:            return Z(a_1, h^old, h, σ)
30:        end if
31:    end if
32:    if a ≡ a_1 && a_2 then
33:        if Z(a_1, h^old, h, σ) = x then
34:            return x
35:        else if Z(a_2, h^old, h, σ) = x then
36:            return x
37:        else
38:            return Z(a_1, h^old, h, σ) + Z(a_2, h^old, h, σ)
39:        end if
40:    end if
41:    return π_∅
42: end function
```

Appendix B

Proofs

Contents

B.1 Soundness of Encoding

Lemma B.1 *(Soundness of old-expression evaluation)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![\textbf{old}(e)]\!] \in [\![s]\!] \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \lceil [\![\textbf{old}(e)]\!], \lambda \rfloor = [\![\textbf{v}]\!] \Rightarrow \lceil \textbf{old}(e), \text{h}_1^{\text{old}}, \text{h}, \sigma \rfloor = \textbf{v}$

Proof By structural induction on the Chalice expression e. □

Lemma B.2 *(Soundness of expression evaluation)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![e]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lceil [\![e]\!], \lambda \rfloor = [\![\textbf{v}]\!] \Rightarrow \lceil e, \text{h}_1^{\text{old}}, \text{h}_1, \sigma \rfloor = \textbf{v}$

Proof By structural induction on the Chalice expression e and using lemma B.1. □

Lemma B.3 *(Soundness of modelling expressions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![e]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![e]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models e$

Proof As a corollary from lemma B.2, using the definition of $\lambda \models a$ given in 2.85, using $[\![\text{True}]\!] = \text{True}$ and using the definition of $(\text{h}^{\text{old}}, \text{h}, \sigma, \pi) \models a$ given in 3.40. □

Lemma B.4 *(Soundness of modelling joinable assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![\textbf{acc}(x.joinable)]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![\textbf{acc}(x.joinable)]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models \textbf{acc}(x.joinable)$

Proof By unfolding all definitions. □

Lemma B.5 *(Soundness of modelling field-access assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![\textbf{acc}(x.f, n)]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![\textbf{acc}(x.f, n)]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models \textbf{acc}(x.f, n)$

Proof By unfolding all definitions. □

Lemma B.6 *(Soundness of modelling predicate-access assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{ ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \}] \wedge [\![\textbf{acc}(x.p)]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![\textbf{acc}(x.p)]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models \textbf{acc}(x.p)$

Proof By unfolding all definitions. □

Lemma B.7 *(Soundness of modelling conditional assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{\, ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \,\}] \wedge [\![e \rightarrow a]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![e \rightarrow a]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models e \rightarrow a$

Proof Unfolding all definitions, using lemma B.2 and induction when reaching $[\![a]\!]$. □

Lemma B.8 *(Soundness of modelling conjuncts of assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{\, ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \,\}] \wedge [\![a_1 \;\&\&\; a_2]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![a_1 \;\&\&\; a_2]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models a_1 \;\&\&\; a_2$

Proof Unfolding all definitions and induction when reaching $[\![a_1]\!]$ and $[\![a_2]\!]$. □

Lemma B.9 *(Soundness of modelling assertions)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{\, ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \,\}] \wedge [\![a]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge \lambda \models [\![a]\!] \Rightarrow (\text{h}_1^{\text{old}}, \text{h}_1, \sigma, \pi) \models a$

Proof As a corollary from lemmas B.3, B.4, B.5, B.6, B.7 and B.8. □

Lemma B.10 *(Completeness of expression evaluation)*
$\forall prog, m.\, VP(prog) \wedge m \in prog \wedge \sigma^{\text{old}} = \sigma_0[meth \mapsto m] \wedge ([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!]) \models [\![\text{Pre}(m)]\!] \wedge ([\![s]\!], \lambda) \in \text{V}[\{\, ([\![\text{Body}(m)]\!], \Lambda[([\![\text{h}^{\text{old}}]\!], [\![\sigma^{\text{old}}]\!], [\![\pi^{\text{old}}]\!])]) \,\}] \wedge [\![e]\!] \in [\![s]\!] \wedge ([\![\text{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \text{LAST}(\lambda) \wedge \sigma \overset{\text{ro}}{\equiv} \sigma^{\text{old}} \wedge \text{h}_1^{\text{old}} \overset{\pi^{\text{old}}}{\equiv} \text{h}^{\text{old}} \wedge \text{h}_1 \overset{\pi}{\equiv} \text{h} \wedge [\![e, \text{h}_1^{\text{old}}, \text{h}_1, \sigma]\!] = \mathbf{v} \Rightarrow [\![[\![e]\!], \lambda]\!] = [\![\mathbf{v}]\!]$

Proof Analogous to B.2. □

Theorem B.11 *(Preservation of runtime configuration validity by Chalice)*
$\forall prog, \mathbf{h}, \mathbf{r}, \mathbf{h}_1, \mathbf{r}_1.\ VP(prog) \wedge VR(prog, \mathbf{h}, \mathbf{r}) \wedge \mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1 \Rightarrow VR(prog, \mathbf{h}_1, \mathbf{r}_1)$

Proof The proof is by induction on the length of the execution leading from $(\mathbf{h}, \mathbf{r})$ to $(\mathbf{h}_1, \mathbf{r}_1)$ as given by the rules of $\rightsquigarrow$. The base case, where no execution took place, trivially holds. An arbitrary execution of length $n+1$ is now assumed where by induction hypothesis, the sub-execution of length n preserves the runtime configuration validity. Thus, left to prove is that the last step in the execution preserves the validity of the runtime configuration. The transition from $(\mathbf{h}, \mathbf{r})$ to $(\mathbf{h}_1, \mathbf{r}_1)$ is now assumed to be the result of a single step of Chalice. A case analysis of the Chalice statements that possibly lead to this last step will now prove that $(\mathbf{h}_1, \mathbf{r}_1)$ is still a valid runtime. Assumptions 1 and 2 are the same for each case analysis in the induction proof and are restated here for later referencing.

$$VP(prog) \Leftrightarrow \forall m \in prog, \forall \mathbf{h}, \sigma, \pi.\ (\mathbf{h}, \sigma, \pi) \models [\![\mathrm{Pre}(m)]\!] \Rightarrow \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[(\mathbf{h}, \sigma, \pi)])\ \}] \models [\![\mathrm{Post}(m)]\!] \tag{A1}$$

$$VR(prog, \mathbf{h}, \mathbf{r}) \ \Leftrightarrow\ \exists \overline{(\pi^{\mathrm{old}}_{\tau}, \pi_{\tau})}. \tag{A2.0}$$
$$\models_{\mathbf{r}} \overline{\pi_{\tau}} \wedge \tag{A2.1}$$
$$\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}).\ \pi_{\tau_2} = \pi_{\varnothing} \wedge \tag{A2.2}$$
$$\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_2, s_3), \tau_2) \in \mathbf{r}. \tag{A2.3}$$
$$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi^{\mathrm{old}}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{A2.4}$$
$$\exists ([\![s_3]\!], \lambda) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi^{\mathrm{old}}_{\tau_2}]\!])])\ \}], \tag{A2.5}$$
$$\exists \mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h}.\ ([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma_2]\!], [\![\pi_{\tau_2}]\!]) = \mathrm{LAST}(\lambda) \wedge \tag{A2.6}$$
$$\mathrm{V}[\{\ ([\![s_3]\!], \lambda)\ \}] \models [\![\mathrm{Post}(m)]\!] \tag{A2.7}$$
$$\text{where } \mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_2(meth),\ \overline{x_i} = \mathrm{Param}(m),\ \sigma^{\mathrm{old}} = [meth \mapsto m][this \mapsto \sigma_2(this)]\overline{[x_i \mapsto \sigma_2(x_i)]}$$

The proof now proceeds to a case analysis of the Chalice statement executed that lead to transition $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$.

Case var $x : t$: In this case, assumption 3, $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$, (A3) has the following derivation tree:

$$\frac{\mathbf{r}[\tau] = ((\mathrm{h}^{\mathrm{old}}, \sigma, (\mathbf{var}\ x : t; s)), \tau) \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \dfrac{\mathbf{v} \in \mathbf{D}_t}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\mathbf{var}\ x : t; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)} \quad \mathbf{h}_1 = \lfloor \mathrm{h} \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]}$$

With $\mathbf{h}_1 = \lfloor \mathrm{h} \rceil = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ and $\mathbf{r}_1 = \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$, the conclusion to prove is as follows.

$$VR(prog, \mathbf{h}_1, \mathbf{r}_1) \Leftrightarrow VR(prog, \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]) \Leftrightarrow \exists \overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)}. \tag{B.1}$$

$$\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \overline{\pi'_\tau} \wedge \tag{B.2}$$

$$\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]).\ \pi'_{\tau_2} = \pi_\varnothing \wedge \tag{B.3}$$

$$\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]. \tag{B.4}$$

$$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{B.5}$$

$$\exists ([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])])\ \}], \tag{B.6}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.7}$$

$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.8}$$

$$\mathrm{V}[\{\ ([\![s_4]\!], \lambda_1)\ \}] \models [\![\mathrm{Post}(m)]\!] \tag{B.9}$$

where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth),\ \overline{x_i} = \mathrm{Param}(m),$

$$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$$

The case is proven by taking the same sequence of permission masks as given by assumption (A2.0):

$$\overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)} = \overline{(\pi_\tau^{\mathrm{old}}, \pi_\tau)} \tag{B.10}$$

It remains to be shown that the choice (B.10) satisfies conclusions (B.2), (B.3) and (B.4)-(B.9).

Claim (B.2)**:** $\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \overline{\pi'_\tau}$

Proof From (B.10): $\overline{\pi'_\tau} = \overline{\pi_\tau}$. Thus:

$$\text{(B.2)} \Leftrightarrow \models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \overline{\pi_\tau} \tag{B.11}$$

By expanding the definition of $\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]}$ given in 5.3:

$$\begin{aligned} \text{(B.11)} \Leftrightarrow \models (& \sum_{\mathbf{e}_\tau \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \pi_\tau) + \\ & (\sum_{\mathbf{e_o} \in free(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)])} \sum_{f \in fields(\mathbf{e_o})} (\varnothing^{\mathbf{F}}[(\mathbf{o}, f) \mapsto 100], \varnothing^{\mathbf{P}})) + \\ & (\sum_{\mathbf{e}_\tau \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)])} (\varnothing^{\mathbf{F}}[(\tau, joinable) \mapsto 100], \varnothing^{\mathbf{P}})) \end{aligned} \tag{B.12}$$

According to (A3), the set of free objects is the same as before the transition:

$$\mathit{free}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]) = \mathit{free}(\mathbf{r}) \tag{B.13}$$

Thus, for term two in (B.12) it holds:

$$(\sum_{\mathbf{e_o} \in \mathit{free}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)])} \sum_{f \in \mathit{fields}(\mathbf{e_o})} (\varnothing^{\mathbf{F}}[(\mathbf{o}, f) \mapsto 100], \varnothing^{\mathbf{P}})) = (\sum_{\mathbf{e_o} \in \mathit{free}(\mathbf{r})} \sum_{f \in \mathit{fields}(\mathbf{e_o})} (\varnothing^{\mathbf{F}}[(\mathbf{o}, f) \mapsto 100], \varnothing^{\mathbf{P}})) \tag{B.14}$$

Similarly, the set of idle threads is the same as before the transition. Thus:

$$(\sum_{\mathbf{e}_\tau \in \mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)])} (\varnothing^{\mathbf{F}}[(\tau, \mathit{joinable}) \mapsto 100], \varnothing^{\mathbf{P}})) = (\sum_{\mathbf{e}_\tau \in \mathit{idle}(\mathbf{r})} (\varnothing^{\mathbf{F}}[(\tau, \mathit{joinable}) \mapsto 100], \varnothing^{\mathbf{P}}))$$

Hence:

$$\begin{aligned} \text{(B.11)} \Leftrightarrow \models & (\sum_{\mathbf{e}_\tau \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \pi_\tau) + \\ & (\sum_{\mathbf{e_o} \in \mathit{free}(\mathbf{r})} \sum_{f \in \mathit{fields}(\mathbf{e_o})} (\varnothing^{\mathbf{F}}[(\mathbf{o}, f) \mapsto 100], \varnothing^{\mathbf{P}})) + \\ & (\sum_{\mathbf{e}_\tau \in \mathit{idle}(\mathbf{r})} (\varnothing^{\mathbf{F}}[(\tau, \mathit{joinable}) \mapsto 100], \varnothing^{\mathbf{P}})) \end{aligned} \tag{B.15}$$

By the operational semantics of Chalice, and in particular by (A3), threads are neither removed nor added to a runtime entity collection. Thus:

$$(\sum_{\mathbf{e}_\tau \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]} \pi_\tau) = (\sum_{\mathbf{e}_\tau \in \mathbf{r}} \pi_\tau) \tag{B.16}$$

From (B.16) it thus follows:

$$\begin{aligned} \text{(B.15)} \Leftrightarrow \models & (\sum_{\mathbf{e}_\tau \in \mathbf{r}} \pi_\tau) + \\ & (\sum_{\mathbf{e_o} \in \mathit{free}(\mathbf{r})} \sum_{f \in \mathit{fields}(\mathbf{e_o})} (\varnothing^{\mathbf{F}}[(\mathbf{o}, f) \mapsto 100], \varnothing^{\mathbf{P}})) + \\ & (\sum_{\mathbf{e}_\tau \in \mathit{idle}(\mathbf{r})} (\varnothing^{\mathbf{F}}[(\tau, \mathit{joinable}) \mapsto 100], \varnothing^{\mathbf{P}})) \end{aligned} \tag{B.17}$$

By definition of $\models_{\mathbf{r}}$ given in 5.3:

$$\text{(B.17)} \Leftrightarrow \models_{\mathbf{r}} \overline{\pi_\tau} \tag{B.18}$$

Summarising (B.11), (B.12), (B.15), (B.17) and (B.18):

$$\text{(B.2)} \Leftrightarrow \models_{\mathbf{r}} \overline{\pi_\tau} \tag{B.19}$$

$\models_{\mathbf{r}} \overline{\pi_\tau}$ holds by assumption (A2.1), which proves claim (B.2). □

Claim (B.3)**:** $\forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]).\ \pi'_{\tau_2} = \pi_{\varnothing}$

Proof By choice of permission masks (B.10):

$$(\text{B.3}) \ \Leftrightarrow \ \forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]).\ \pi_{\tau_2} = \pi_{\varnothing} \tag{B.20}$$

According to (A3), the set of idle threads remained unchanged during the transition:

$$\mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]) = \mathit{idle}(\mathbf{r}) \tag{B.21}$$

Thus:

$$(\text{B.20}) \ \Leftrightarrow \ \forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}).\ \pi_{\tau_2} = \pi_{\varnothing} \tag{B.22}$$

Summarising (B.20) and (B.22):

$$(\text{B.3}) \ \Leftrightarrow \ \forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}).\ \pi_{\tau_2} = \pi_{\varnothing} \tag{B.23}$$

$\forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}).\ \pi_{\tau_2} = \pi_{\varnothing}$ holds by assumption (A2.2), which proves claim (B.3). □

Claim (B.4)**-**(B.9)**:** $\forall((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$.
$(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \ \wedge$
$\exists(\llbracket s_4 \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket)])\ \}], \exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$.
$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket, \llbracket \sigma_3 \rrbracket, \llbracket \pi'_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda_1) \ \wedge \ \mathrm{V}[\{\ (\llbracket s_4 \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket$
where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m), \sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma_3(\mathit{this})]\overline{[x_i \mapsto \sigma_3(x_i)]}$

Proof Claim (B.4)-(B.9) quantifies over the set of active threads in runtime-entity collection $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$. According to (A3), this set of active threads can be partitioned as follows: a) thread τ which executed statement **var** $x : t$ and b) all other active threads $\tau_2 \neq \tau$. Claim (B.4)-(B.9) is proven if it holds for thread τ as well as for an arbitrary thread chosen from partition b).

Arbitrary active but non-executing thread $\tau_2 \neq \tau$:

Proof By chosing $((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$ with $\tau_2 \neq \tau$, claim (B.4)-(B.9) reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \ \wedge \tag{B.24}$$

$$\exists(\llbracket s_4 \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket)])\ \}], \tag{B.25}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.26}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket, \llbracket \sigma_3 \rrbracket, \llbracket \pi'_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda_1) \ \wedge \tag{B.27}$$

$$\mathrm{V}[\{\ (\llbracket s_4 \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.28}$$

where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m),$
$\sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma_3(\mathit{this})]\overline{[x_i \mapsto \sigma_3(x_i)]}$

By choice of permissions (B.10): $\pi'_{\tau_2} = \pi_{\tau_2}$ and $\pi^{\text{old}'}_{\tau_2} = \pi^{\text{old}}_{\tau_2}$. Furthermore, as τ_2 is active but non-executing: $\sigma_3 = \sigma_2$, $s_4 = s_3$, and $\sigma^{\text{old}'} = \sigma^{\text{old}}$, all given by assumption (A2.3)-(A2.7). Claim (B.4)-(B.9) thus reduces to:

$$(\llbracket \text{h}^{\text{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\text{old}} \rrbracket, \llbracket \pi^{\text{old}}_{\tau_2} \rrbracket) \models \llbracket \text{Pre}(m) \rrbracket \wedge \tag{B.29}$$

$$\exists (\llbracket s_3 \rrbracket, \lambda_1) \in \text{V}[\{\, (\llbracket \text{Body}(m) \rrbracket, \Lambda[(\llbracket \text{h}^{\text{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\text{old}} \rrbracket, \llbracket \pi^{\text{old}}_{\tau_2} \rrbracket)]) \,\}], \tag{B.30}$$

$$\exists \mathbf{h}_4 \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.31}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\text{h}^{\text{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket, \llbracket \sigma_2 \rrbracket, \llbracket \pi_{\tau_2} \rrbracket) = \text{LAST}(\lambda_1) \wedge \tag{B.32}$$

$$\text{V}[\{\, (\llbracket s_3 \rrbracket, \lambda_1) \,\}] \models \llbracket \text{Post}(m) \rrbracket \tag{B.33}$$

$$\text{where } \text{h}^{\text{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_2(\mathit{meth}),\ \overline{x_i} = \text{Param}(m),$$
$$\sigma^{\text{old}} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma_2(\mathit{this})]\overline{[x_i \mapsto \sigma_2(x_i)]}$$

(B.29) holds by assumption (A2.4). To prove (B.30)-(B.33), the partial trace $(\llbracket s_3 \rrbracket, \lambda)$ and the runtime heap $\mathbf{h}_3$, promised by assumption (A2.5), are chosen. Thus, assume:

$$(\llbracket s_3 \rrbracket, \lambda) \in \text{V}[\{\, (\llbracket \text{Body}(m) \rrbracket, \Lambda[(\llbracket \text{h}^{\text{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\text{old}} \rrbracket, \llbracket \pi^{\text{old}}_{\tau_2} \rrbracket)]) \,\}] \tag{B.34}$$

$$\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h} \tag{B.35}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma_2 \rrbracket, \llbracket \pi_{\tau_2} \rrbracket) = \text{LAST}(\lambda) \tag{B.36}$$

$$\text{V}[\{\, (\llbracket s_3 \rrbracket, \lambda) \,\}] \models \llbracket \text{Post}(m) \rrbracket \tag{B.37}$$

The existence of $(\llbracket s_3 \rrbracket, \lambda)$ and $\mathbf{h}_3$ then prove (B.30)-(B.33) if additionally $\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ and $\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket = \llbracket \lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\text{h}^{\text{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket$ holds.

Claim: $\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$

Proof By assumption (B.35):

$$\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h} \tag{B.38}$$

Assume $\pi_{\tau_2} = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$. Then, by definition of $\overset{\pi_{\tau_2}}{\equiv}$ given in 5.5:

$$\text{(B.38)} \Leftrightarrow \forall (\mathbf{o}, f).\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}, f) = \mathbf{h}(\mathbf{o}, f) \tag{B.39}$$

Using the definition of $\lceil \_, \_ \rfloor$ given in 3.29:

$$\text{(B.39)} \Leftrightarrow \forall (\mathbf{o}, f).\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}, f) = \lceil \mathbf{h}, \mathbf{r} \rfloor (\mathbf{o}, f) \tag{B.40}$$

Using the definition of $\lfloor \_ \rceil$ given in 3.29:

$$\text{(B.40)} \Leftrightarrow \forall (\mathbf{o}, f).\, \pi^{\mathbf{F}}(\mathbf{o}, f) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}, f) = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil (\mathbf{o}, f) \tag{B.41}$$

Summarising (B.38)-(B.41) and applying the definition of $\overset{\pi_{\tau_2}}{\equiv}$ given in 5.5 proves the claim:

$$\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h} \Leftrightarrow \mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil \tag{B.42}$$

□

Claim: $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!]$

Proof By definition of function equality:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!] \Leftrightarrow \\ \forall(\mathbf{o}, f).[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}, f) = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!](\mathbf{o}, f) \tag{B.43}$$

The claim is proven by a case analysis on f.

Case $f \notin \{ \mathit{recv}, \overline{\mathit{arg}_i} \}$

Then by definition of heap encoding $[\![\_]\!]$ given in 4.14:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}, f) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}), f)]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!](\mathbf{o}, f) \tag{B.44}$$

Case $f = \mathit{recv}$

Then by definition of heap encoding $[\![\_]\!]$ given in 4.14:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}, f) = [\![\sigma_4(\mathit{this})]\!] \\ \text{where } ((\mathrm{h}_2^{\mathrm{old}}, \sigma_4, s_5), \tau_3) \in \mathbf{r} \text{ s.t. } \tau_3 = \gamma^{-1}(\mathbf{o}) \tag{B.45}$$

In Chalice, variable *this* is read-only. Furthermore, by assumption (A3), no active thread was reset to idle. Thus, thread τ_3 still exists in $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$ with the same value for *this* in σ_4:

$$[\![\sigma_4(\mathit{this})]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!](\mathbf{o}, f) \tag{B.46}$$

Case $f = \mathit{arg}_i$

Analogous to the case $f = \mathit{recv}$.

Thus, the claim holds:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!] \tag{B.47}$$

□

Thus, conclusion (B.4)-(B.9) holds for an arbitrary but non-executing thread. □

Active and executing thread τ:

Proof For the active and executing thread $((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)]$ claim (B.4)-(B.9) reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.48}$$

$$\exists (\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket)])\ \}], \tag{B.49}$$

$$\exists \mathbf{h}_4 \overset{\pi'_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.50}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket, \llbracket \sigma[x \mapsto \mathbf{v}] \rrbracket, \llbracket \pi'_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.51}$$

$$\mathrm{V}[\{\ (\llbracket s \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.52}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma[x \mapsto \mathbf{v}](\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma[x \mapsto \mathbf{v}](\mathit{this})]\overline{[x_i \mapsto \sigma[x \mapsto \mathbf{v}](x_i)]}$$

By choice of permissions (B.10): $\pi'_\tau = \pi_\tau$ and $\pi_\tau^{\mathrm{old}'} = \pi_\tau^{\mathrm{old}}$. Furthermore, variable *meth* is ghost, variable *this* is read-only as are variables $\overline{x_i}$. Thus: $\sigma[x \mapsto \mathbf{v}](\mathit{meth}) = \sigma(\mathit{meth})$, $\sigma[x \mapsto \mathbf{v}](\mathit{this}) = \sigma(\mathit{this})$ and $\overline{\sigma[x \mapsto \mathbf{v}](x_i) = \sigma(x_i)}$. Thus, $\sigma^{\mathrm{old}'} = \sigma^{\mathrm{old}}$, given by assumption (A2.3)-(A2.7). Conclusion (B.4)-(B.9) thus reduces for τ to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.53}$$

$$\exists (\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}], \tag{B.54}$$

$$\exists \mathbf{h}_4 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.55}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor \rrbracket, \llbracket \sigma[x \mapsto \mathbf{v}] \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.56}$$

$$\mathrm{V}[\{\ (\llbracket s \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.57}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma(\mathit{this})]\overline{[x_i \mapsto \sigma(x_i)]}$$

(B.53) holds by assumption (A2.4). Assumptions (A2.5)-(A2.7) promise the existence of a partial trace $(\llbracket \mathbf{var}\ x : t; s \rrbracket, \lambda)$ and a runtime heap $\mathbf{h}_3$ for which it holds:

$$(\llbracket \mathbf{var}\ x : t; s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.58}$$

$$\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h} \tag{B.59}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda) \tag{B.60}$$

$$\mathrm{V}[\{\ (\llbracket \mathbf{var}\ x : t; s \rrbracket, \lambda)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.61}$$

By definition of statement encoding $\llbracket \_ \rrbracket$ given in 4.3:

$$(\llbracket \mathbf{var}\ x : t; s \rrbracket, \lambda) = (\mathbf{var}\ \llbracket x \rrbracket : \llbracket t \rrbracket; \llbracket s \rrbracket, \lambda) \tag{B.62}$$

From the definition 2.64 of the Viper trace semantics of variable declarations and (B.60) it follows:

$$(\mathrm{C}[\mathbf{var}\ \llbracket x \rrbracket : \llbracket t \rrbracket], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket[\llbracket x \rrbracket \mapsto \mathbf{v}_1], \llbracket \pi_\tau \rrbracket)) \tag{B.63}$$
$$\text{where } \mathrm{C} = \bullet; \llbracket s \rrbracket \text{ and } \mathbf{v}_1 \in \mathbf{D}_{\llbracket t \rrbracket}$$

From the definition 2.26 of plugging a statement into an evaluation context it follows:

$$(\mathrm{C}[\varepsilon], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) = ([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) \tag{B.64}$$

From the definition 2.82 of the reflexive and transitive closure $\mathrm{V}[\psi]$, (B.58) and (B.62)-(B.64) it follows:

$$([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi_\tau^{\mathrm{old}}]\!])]) \,\}] \tag{B.65}$$

Moreover, the definition 2.82 of the reflexive and transitive closure $\mathrm{V}[\psi]$ also allows to conclude:

$$\forall \mathbf{v}_1 \in \mathbf{D}_{[\![t]\!]}.([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi_\tau^{\mathrm{old}}]\!])]) \,\}] \tag{B.66}$$

The assumed bijection γ defined in 4.10 and the definition 4.11 of value encodings $[\![\_]\!]$ allows to conclude:

$$\exists([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi_\tau^{\mathrm{old}}]\!])]) \,\}] \tag{B.67}$$
where $\mathbf{v}_1 = [\![\mathbf{v}]\!]$ with $\mathbf{v}$ given by assumption (A3)

The partial trace $([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!]))$ promised by (B.67) is now taken. Thus assume:

$$([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!])) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi_\tau^{\mathrm{old}}]\!])]) \,\}] \tag{B.68}$$

$$\mathbf{v}_1 = [\![\mathbf{v}]\!] \tag{B.69}$$

Recall in order for claim (B.4)-(B.9) to hold, (B.54)-(B.57) have to hold. This is proven to hold with the partial trace $([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1], [\![\pi_\tau]\!]))$ given by (B.68) and $\mathbf{h}_3$ given by (B.59). In order for claim (B.4)-(B.9) to hold with this choice, it is left to prove that $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$, $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!]$ and $[\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1] = [\![\sigma[x \mapsto \mathbf{v}]]\!]$. $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ holds with an argument analogous to (B.42). $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \mathbf{v}], s), \tau)] \rfloor]\!]$ holds with an argument analogous to (B.47). Left to prove: $[\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1] = [\![\sigma[x \mapsto \mathbf{v}]]\!]$.

Claim: $[\![\sigma]\!][[\![x]\!] \mapsto \mathbf{v}_1] = [\![\sigma[x \mapsto \mathbf{v}]]\!]$

Proof First recall (B.69): $\mathbf{v}_1 = [\![\mathbf{v}]\!]$. Thus, left to prove: $[\![\sigma]\!][[\![x]\!] \mapsto [\![\mathbf{v}]\!]] = [\![\sigma[x \mapsto \mathbf{v}]]\!]$. By definition of store equality:

$$\begin{aligned} &[\![\sigma]\!][[\![x]\!] \mapsto [\![\mathbf{v}]\!]] = [\![\sigma[x \mapsto \mathbf{v}]]\!] \;\Leftrightarrow \\ &\forall [\![y]\!].[\![\sigma]\!][[\![x]\!] \mapsto [\![\mathbf{v}]\!]]([\![y]\!]) = [\![\sigma[x \mapsto \mathbf{v}]]\!]([\![y]\!]) \end{aligned} \tag{B.70}$$

The claim is then proven by a case analysis on y with either $y = x$ or $y \neq x$ and a straight forward unfolding of the definition of store encodings $[\![]\!]$ given in 4.12. □

Thus, conclusion (B.4)-(B.9) holds for the active and executing thread. □

Thus, conclusion (B.4)-(B.9) holds for an arbitrary active thread. □

Case if e **then** s_1 **else** s_2: In this case, assumption 3, $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$, (A3) has the following derivation tree:

$$\frac{\mathbf{r}[\tau] = ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s)), \tau) \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \dfrac{[\![e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma]\!] = \mathrm{True}}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s)), \tau) \rightharpoonup \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)} \quad \mathbf{h}_1 = \lfloor \mathrm{h} \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]}$$

With $\mathbf{h}_1 = \lfloor \mathrm{h} \rceil = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ and $\mathbf{r}_1 = \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]$, the conclusion to prove is as follows.

$$VR(prog, \mathbf{h}_1, \mathbf{r}_1) \;\Leftrightarrow\; VR(prog, \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]) \;\Leftrightarrow\; \exists \overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)}. \tag{B.71}$$

$$\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]} \overline{\pi'_\tau} \wedge \tag{B.72}$$

$$\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]).\ \pi'_{\tau_2} = \pi_\varnothing \wedge \tag{B.73}$$

$$\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]. \tag{B.74}$$

$$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{B.75}$$

$$\exists ([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])]\ )\ \}], \tag{B.76}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.77}$$

$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.78}$$

$$\mathrm{V}[\{\ ([\![s_4]\!], \lambda_1)\ \}] \models [\![\mathrm{Post}(m)]\!] \tag{B.79}$$

where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth),\ \overline{x_i} = \mathrm{Param}(m),$
$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$

The case is proven by taking the same sequence of permission masks as given by assumption (A2.0):

$$\overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)} = \overline{(\pi_\tau^{\mathrm{old}}, \pi_\tau)} \tag{B.80}$$

It remains to be shown that the choice (B.80) satisfies conclusions (B.72), (B.73) and (B.74)-(B.79).

Claim (B.72)**:** $\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]} \overline{\pi'_\tau}$

Proof Analogous to proof of claim (B.2). □

Claim (B.73)**:** $\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]).\ \pi'_{\tau_2} = \pi_\varnothing$

Proof Analogous to proof of claim (B.3). □

Claim (B.74)**-**(B.79)**:** $\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)].$
$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge$
$\exists ([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])]\ )\ \}], \exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil.$
$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \mathrm{V}[\{\ ([\![s_4]\!], \lambda_1)\ \}] \models [\![\mathrm{Post}(m)]\!]$
where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth),\ \overline{x_i} = \mathrm{Param}(m), \sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$

Proof Claim (B.74)-(B.79) quantifies over the set of active threads in runtime-entity collection $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]$. According to (A3), this set of active threads can be partitioned as follows: a) thread τ which executed statement **if** e **then** s_1 **else** s_2 and b) all other active threads $\tau_2 \neq \tau$. Claim (B.74)-(B.79) is proven if it holds for thread τ as well as for an arbitrary thread chosen from partition b).

Arbitrary active but non-executing thread $\tau_2 \neq \tau$:

Proof Analogous to proof of claim (B.4)-(B.9) and the subproof considering the arbitrary active but non-executing thread. □

Active and executing thread τ:

Proof For the active and executing thread $((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)]$ claim (B.74)-(B.79) reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.81}$$

$$\exists (\llbracket s_1; s \rrbracket, \lambda_1) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket)]) \,\}], \tag{B.82}$$

$$\exists \mathbf{h}_4 \overset{\pi'_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.83}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi'_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.84}$$

$$\mathrm{V}[\{\, (\llbracket s_1; s \rrbracket, \lambda_1) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.85}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}$$

By choice of permissions (B.80): $\pi'_\tau = \pi_\tau$ and $\pi_\tau^{\mathrm{old}'} = \pi_\tau^{\mathrm{old}}$. Moreover $\sigma^{\mathrm{old}'} = \sigma^{\mathrm{old}}$, given by assumption (A2.3)-(A2.7). Claim (B.74)-(B.79) thus reduces for τ to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.86}$$

$$\exists (\llbracket s_1; s \rrbracket, \lambda_1) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)]) \,\}], \tag{B.87}$$

$$\exists \mathbf{h}_4 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.88}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.89}$$

$$\mathrm{V}[\{\, (\llbracket s_1; s \rrbracket, \lambda_1) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.90}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}$$

(B.86) holds by assumption (A2.4). Assumptions (A2.5)-(A2.7) promise the existence of a partial

trace $(\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)$ and a runtime heap $\mathbf{h}_3$ for which it holds:

$$(\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.91}$$

$$\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h} \tag{B.92}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda) \tag{B.93}$$

$$\mathrm{V}[\{\ (\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.94}$$

By definition of statement encoding $\llbracket \_ \rrbracket$ given in 4.3:

$$(\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda) = (\overline{\textbf{assert } \llbracket x_i \rrbracket \text{ != } \llbracket null \rrbracket}; \textbf{if } \llbracket e \rrbracket \textbf{ then } \llbracket s_1 \rrbracket \textbf{ else } \llbracket s_2 \rrbracket; \llbracket s \rrbracket, \lambda) \tag{B.95}$$

Recall the semantics of Viper **assert** statements given in 2.70: a failing **assert** results in an error-trace. Moreover, recall the rule of error-trace propagation given in 2.63: error-traces unconditionally propagate to the end of the evaluation. Also recall A2.1 according to which the current program is a valid program. From this and (B.86) and (B.91) it follows that all **assert** statements succeed. According to the semantics of Viper **assert** statements, trace λ continues unaffected through succeeding assert statements. From this, (B.91) and (B.95) it then follows:

$$(\textbf{if } \llbracket e \rrbracket \textbf{ then } \llbracket s_1 \rrbracket \textbf{ else } \llbracket s_2 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.96}$$

Additionally, by definition 2.82 of the closure $\mathrm{V}[]$ of the lifted trace semantics, it also holds:

$$(\textbf{if } \llbracket e \rrbracket \textbf{ then } \llbracket s_1 \rrbracket \textbf{ else } \llbracket s_2 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)\ \}] \tag{B.97}$$

Now recall the semantics of Viper if-else statements given in 2.77. Moreover recall definition 2.82 of the closure $\mathrm{V}[]$ of the lifted trace semantics. With this, the following holds:

$$(\textbf{assume } \llbracket e \rrbracket; \llbracket s_1 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.98}$$

$$(\textbf{assume } \llbracket e \rrbracket; \llbracket s_1 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)\ \}] \tag{B.99}$$

Recall from assumption (A3): $\llbracket e, \mathrm{h}^{\mathrm{old}}, \mathrm{h}, \sigma \rrbracket = \mathrm{True}$. From this and lemma B.10 it follows: $\llbracket \llbracket e \rrbracket, \lambda \rrbracket = \llbracket \mathrm{True} \rrbracket \overset{4.11}{=} \mathrm{True}$. From this, (B.98) and the semantics of Viper **assume** statements given in 2.71 the following follows:

$$(\llbracket s_1 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.100}$$

$$(\llbracket s_1 \rrbracket; \llbracket s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)\ \}] \tag{B.101}$$

From this and the definition of statement encodings given in 4.3 the following follows:

$$(\llbracket s_1; s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.102}$$

$$(\llbracket s_1; s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s \rrbracket, \lambda)\ \}] \tag{B.103}$$

From (B.94) and (B.103) and the definition of the closure $\mathrm{V}[]$ given in 2.82 the following follows:

$$\mathrm{V}[\{\ (\llbracket s_1; s \rrbracket, \lambda)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.104}$$

Recall in order for claim (B.74)-(B.79) to hold, (B.87)-(B.90) has to hold. This can be proven for $(\llbracket s_1; s \rrbracket, \lambda)$, as given by (B.102), and $\mathbf{h}_3$ as given by (B.92). With this choice and (B.104), (B.90) holds. Left to prove with this choice is: $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ and $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor]\!]$. $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ holds with an argument analogous to (B.42). $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s_1; s), \tau)] \rfloor]\!]$ holds with an argument analogous to (B.47). Thus, claim (B.74)-(B.79) holds for the active and executing thread τ. □

Thus, conclusion (B.74)-(B.79) holds for an arbitrary active thread. □

Case fork $x := y.m(\overline{z_i})$: In this case, assumption 3, $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$, (A3) has the following derivation tree:

$$\frac{\mathrm{h}_1^{\mathrm{old}} = \mathrm{h} \quad \sigma(y) = \mathbf{o} \quad \sigma_1 = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \mathbf{o}]\overline{[x_i \mapsto \sigma(z_i)]} \quad s_1 = \mathrm{Body}(m)}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{fork } x := y.m(\overline{z_i}); s)), \tau) | (\mathrm{idle}, \tau_1) \rightarrow \mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau) | ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)}$$

$$\frac{\mathbf{r}[\tau] = ((\mathrm{h}^{\mathrm{old}}, \sigma, (\textbf{fork } x := y.m(\overline{z_i}); s)), \tau) \quad \mathbf{r}[\tau_1] = (\mathrm{idle}, \tau_1) \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \mathbf{h}_1 = \lfloor \mathrm{h} \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]}$$

With $\mathbf{h}_1 = \lfloor \mathrm{h} \rceil = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil$ and $\mathbf{r}_1 = \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]$, the conclusion to prove is as follows.

$$\begin{aligned}
&\mathit{VR}(\mathit{prog}, \mathbf{h}_1, \mathbf{r}_1) \Leftrightarrow \mathit{VR}(\mathit{prog}, \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]) \Leftrightarrow \exists \overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)}. && \text{(B.105)}\\
&\quad \models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]} \overline{\pi'_\tau} \wedge && \text{(B.106)}\\
&\quad \forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]). \ \pi'_{\tau_2} = \pi_\varnothing \wedge && \text{(B.107)}\\
&\quad \forall ((\mathrm{h}_{\tau_2}^{\mathrm{old}}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]. && \text{(B.108)}\\
&\qquad (\llbracket \mathrm{h}_{\tau_2}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_{\tau_2}^{\mathrm{old}'} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge && \text{(B.109)}\\
&\qquad \exists (\llbracket s_4 \rrbracket, \lambda_1) \in \mathrm{V}[\{ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}_{\tau_2}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_{\tau_2}^{\mathrm{old}'} \rrbracket)]) \}], && \text{(B.110)}\\
&\qquad \exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{=} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. && \text{(B.111)}\\
&\qquad\quad (\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor \rrbracket, \llbracket \sigma_3 \rrbracket, \llbracket \pi'_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge && \text{(B.112)}\\
&\qquad\quad \mathrm{V}[\{ (\llbracket s_4 \rrbracket, \lambda_1) \}] \models \llbracket \mathrm{Post}(m) \rrbracket && \text{(B.113)}\\
&\qquad \text{where } \mathrm{h}_{\tau_2}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(\mathit{meth}), \ \overline{x_i} = \mathrm{Param}(m),\\
&\qquad\qquad \sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma_3(\mathit{this})]\overline{[x_i \mapsto \sigma_3(x_i)]}
\end{aligned}$$

The case is proven by taking the following permission masks:

$$\pi_\tau^{\mathrm{old}'} = \pi_\tau^{\mathrm{old}} \tag{B.114}$$

$$\pi'_\tau = \pi_\tau + (\varnothing^{\mathbf{F}}[(\tau_1, \mathit{joinable}) \mapsto 100], \varnothing^{\mathbf{P}}) - \mathbf{Z}(\mathrm{Pre}(m)[y/\mathit{this}]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma) \tag{B.115}$$

$$\pi_{\tau_1}^{\mathrm{old}'} = \mathbf{Z}(\mathrm{Pre}(m)[y/\mathit{this}]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma) \tag{B.116}$$

$$\pi'_{\tau_1} = \mathbf{Z}(\mathrm{Pre}(m)[y/\mathit{this}]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma) \tag{B.117}$$

$$\forall \tau_2 \notin \{ \tau, \tau_1 \}. (\pi_{\tau_2}^{\mathrm{old}'}, \pi'_{\tau_2}) = (\pi_{\tau_2}^{\mathrm{old}}, \pi_{\tau_2}) \tag{B.118}$$

where m is the method forked by τ and now executed by τ_1

It remains to be shown that choices (B.114)-(B.118) satisfy conclusions (B.106), (B.107) and (B.108)-(B.113).

Claim (B.106)**:** $\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]} \overline{\pi'_\tau}$

Proof Analogous to proof of claim (B.2) and noting that τ_1 was removed from the set of idle threads and the permissions π'_{τ_1} given to τ_1 are the permissions taken away from τ. □

Claim (B.107)**:** $\forall \mathbf{e}_{\tau_2} \in \mathit{idle}(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]).\ \pi'_{\tau_2} = \pi_{\varnothing}$

Proof Analogous to proof of claim (B.3) and noting that with (B.118) all threads that are still idle have the same permissions as before. □

Claim (B.108)**-**(B.113)**:** $\forall((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)].$
$(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \ \wedge$
$\exists(\llbracket s_4 \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau_2} \rrbracket)])\ \}], \exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil.$
$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor \rrbracket, \llbracket \sigma_3 \rrbracket, \llbracket \pi'_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda_1) \ \wedge$
$\mathrm{V}[\{\ (\llbracket s_4 \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket$
where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m), \sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma_3(\mathit{this})]\overline{[x_i \mapsto \sigma_3(x_i)]}$

Proof Claim (B.108)-(B.113) quantifies over the set of active threads in runtime-entity collection $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]$. According to (A3), this set of active threads can be partitioned as follows: a) thread τ which executed statement **fork** $x := y.m(\overline{z_i})$ b) the newly forked thread τ_1 and c) all other active threads $\tau_2 \notin \{\ \tau, \tau_1\ \}$. Claim (B.108)-(B.113) is proven if it holds for thread τ, thread τ_1 as well as for an arbitrary thread τ_2 chosen from partition c).

Active and forking thread τ:

Proof For the active and executing thread $((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]$ claim (B.108)-(B.113) reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \ \wedge \tag{B.119}$$

$$\exists(\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi^{\mathrm{old}'}_{\tau} \rrbracket)])\ \}], \tag{B.120}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.121}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor \rrbracket, \llbracket \sigma[x \mapsto \tau_1] \rrbracket, \llbracket \pi'_{\tau} \rrbracket) = \mathrm{LAST}(\lambda_1) \ \wedge \tag{B.122}$$

$$\mathrm{V}[\{\ (\llbracket s \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.123}$$

where $\mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma[x \mapsto \tau_1](\mathit{meth}),\ \overline{x_i} = \mathrm{Param}(m),$
$\sigma^{\mathrm{old}'} = [\mathit{meth} \mapsto m][\mathit{this} \mapsto \sigma[x \mapsto \tau_1](\mathit{this})]\overline{[x_i \mapsto \sigma[x \mapsto \tau_1](x_i)]}$

By choice of permissions (B.114): $\pi^{\mathrm{old}'}_{\tau} = \pi^{\mathrm{old}}_{\tau}$. Furthermore, variable *meth* is ghost, variable *this* is read-only as are variables $\overline{x_i}$. Thus: $\sigma[x \mapsto \tau_1](\mathit{meth}) = \sigma(\mathit{meth})$, $\sigma[x \mapsto \tau_1](\mathit{this}) = \sigma(\mathit{this})$ and $\overline{\sigma[x \mapsto \tau_1](x_i) = \sigma(x_i)}$. Thus, $\sigma^{\mathrm{old}'} = \sigma^{\mathrm{old}}$, given by assumption (A2.3)-(A2.7). Claim (B.108)-

(B.113) thus reduces for τ to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.124}$$

$$\exists (\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}], \tag{B.125}$$

$$\exists \mathbf{h}_4 \overset{\pi'_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.126}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor \rrbracket, \llbracket \sigma[x \mapsto \tau_1] \rrbracket, \llbracket \pi'_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.127}$$

$$\mathrm{V}[\{\ (\llbracket s \rrbracket, \lambda_1)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.128}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}$$

(B.124) holds by assumption (A2.4). Thus left to prove in order for claim (B.108)-(B.113) to hold are (B.125)-(B.128). Assumptions (A2.5)-(A2.7) promise the existence of a partial trace $(\llbracket \textbf{fork}\ x := y.m(\overline{z_i}); s \rrbracket, \lambda)$ and a runtime heap $\mathbf{h}_3$ for which it holds:

$$(\llbracket \textbf{fork}\ x := y.m(\overline{z_i}); s \rrbracket, \lambda) \in \mathrm{V}[\{\ (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)])\ \}] \tag{B.129}$$

$$\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h} \tag{B.130}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda) \tag{B.131}$$

$$\mathrm{V}[\{\ (\llbracket \textbf{fork}\ x := y.m(\overline{z_i}); s \rrbracket, \lambda)\ \}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.132}$$

By definition of statement encoding $\llbracket \_ \rrbracket$ given in 4.3:

$$(\llbracket \textbf{fork}\ x := y.m(\overline{z_i}); s \rrbracket, \lambda) = (\textbf{exhale}\ \llbracket \mathrm{Pre}(m) \rrbracket[\llbracket y \rrbracket / \llbracket this \rrbracket]\overline{[\llbracket z_i \rrbracket / \llbracket x_i \rrbracket]}; \tag{B.133}$$

$$\textbf{var}\ \llbracket x \rrbracket : Ref; \tag{B.134}$$

$$\llbracket x \rrbracket := \textbf{new}(recv, \overline{arg_i}); \tag{B.135}$$

$$\llbracket x \rrbracket.recv := \llbracket y \rrbracket; \tag{B.136}$$

$$\overline{\llbracket x \rrbracket.arg_i := \llbracket z_i \rrbracket}; \tag{B.137}$$

$$\llbracket s \rrbracket, \lambda) \tag{B.138}$$

From (B.132), the Viper rule of unconditional error-trace propagation (2.63), the definition of the Viper closure V[] given in 2.82 and the definition of $\models$ given in (2.88) it follows that the **exhale** in (B.133) must succeed. Thus:

$$(\mathrm{C}[\textbf{exhale}\ \llbracket \mathrm{Pre}(m) \rrbracket[\llbracket y \rrbracket / \llbracket this \rrbracket]\overline{[\llbracket z_i \rrbracket / \llbracket x_i \rrbracket]}, \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_2) \wedge \lambda_1 \notin \Lambda_{\mathrm{Err}} \tag{B.139}$$

$$\text{with } \mathrm{C} = \bullet; \textbf{var}\ \llbracket x \rrbracket : Ref; \llbracket x \rrbracket := \textbf{new}(recv, \overline{arg_i}); \llbracket x \rrbracket.recv := \llbracket y \rrbracket; \overline{\llbracket x \rrbracket.arg_i := \llbracket z_i \rrbracket}; \llbracket s \rrbracket$$

From (B.139) and (B.131) and lemmas B.13, B.15 and B.14 it follows:

$$(\mathbf{h}^\star, \llbracket \sigma \rrbracket, \pi^\star) = \mathrm{LAST}(\lambda_2) \tag{B.140}$$

$$\text{where } \pi^\star = \llbracket \pi_\tau \rrbracket - \mathbf{Q}^{\mathrm{T}}(\llbracket \mathrm{Pre}(m) \rrbracket[\llbracket y \rrbracket / \llbracket this \rrbracket]\overline{[\llbracket z_i \rrbracket / \llbracket x_i \rrbracket]}, \lambda, \mathrm{True}),\ \mathbf{h}^\star \overset{\pi^\star}{\equiv} \llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket$$

From (B.139) and the definition of the semantics of Viper variable declaration statements given in (2.64) it follows:

$$\begin{aligned}&\forall \mathbf{o} \in \mathbf{O}.\, \exists \lambda_3.\, (\mathrm{C}_1[\mathbf{var}\ [\![x]\!] : \mathit{Ref}], \lambda_2) \rightsquigarrow (\mathrm{C}_1[\varepsilon], \lambda_3) \\ &\text{with } \mathrm{C}_1 = \bullet; [\![x]\!] := \mathbf{new}(\mathit{recv}, \overline{\mathit{arg}_i}); [\![x]\!].\mathit{recv} := [\![y]\!]; \overline{[\![x]\!].\mathit{arg}_i := [\![z_i]\!]}; [\![s]\!] \\ &\text{and } (\mathbf{h}^\star, [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{o}], \pi^\star) = \mathrm{LAST}(\lambda_3)\end{aligned} \tag{B.141}$$

From (B.141) and the definition of the semantics of Viper allocation statements given in (2.65) it follows:

$$\begin{aligned}&\forall \mathbf{o}^\star \notin \mathbf{O}(\lambda_3).\, \exists \lambda_4.\, (\mathrm{C}_2[[\![x]\!] := \mathbf{new}(\mathit{recv}, \overline{\mathit{arg}_i})], \lambda_3) \rightsquigarrow (\mathrm{C}_2[\varepsilon], \lambda_4) \\ &\text{with } \mathrm{C}_2 = \bullet; [\![x]\!].\mathit{recv} := [\![y]\!]; \overline{[\![x]\!].\mathit{arg}_i := [\![z_i]\!]}; [\![s]\!] \\ &\text{and } (\mathbf{h}^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto \mathbf{v}]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto \mathbf{v}_i]}, [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{o}][[\![x]\!] \mapsto \mathbf{o}^\star], \pi^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto 1]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto 1]}) = \mathrm{LAST}(\lambda_4)\end{aligned} \tag{B.142}$$

The proof now continues by analysing $(\mathrm{C}_2[\varepsilon], \lambda_4)$ for which it holds:

$$\mathrm{C}_2 = \bullet; [\![x]\!].\mathit{recv} := [\![y]\!]; \overline{[\![x]\!].\mathit{arg}_i := [\![z_i]\!]}; [\![s]\!] \tag{B.143}$$

$$(\mathbf{h}^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto \mathbf{v}]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto \mathbf{v}_i]}, [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{o}][[\![x]\!] \mapsto \mathbf{o}^\star], \pi^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto 1]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto 1]}) = \mathrm{LAST}(\lambda_4) \tag{B.144}$$

$$\mathbf{o}^\star = \gamma(\tau_1) \tag{B.145}$$

From (B.143), (B.144) and (B.145) as well as the Viper semantics of field assignments given in 2.67 it follows (simplified):

$$(\mathrm{C}_2[\varepsilon], \lambda_4) \rightsquigarrow \ldots \rightsquigarrow ([\![s]\!], \lambda_5) \tag{B.146}$$

$$\begin{aligned}&(\mathbf{h}^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto [\![\sigma]\!]([\![y]\!])]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto [\![\sigma]\!]([\![z_i]\!])]}, \\ &\qquad [\![\sigma]\!][[\![x]\!] \mapsto \mathbf{o}^\star], \\ &\qquad \pi^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto 1]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto 1]}) = \mathrm{LAST}(\lambda_5)\end{aligned} \tag{B.147}$$

$$\mathbf{o}^\star = \gamma(\tau_1) \tag{B.148}$$

Moreover, from (B.129), (B.132), (B.139), (B.141), (B.142), (B.146) and the definition of the closure $\mathrm{V}[]$ given in 2.82 it follows:

$$([\![s]\!], \lambda_5) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}}]\!], [\![\pi_\tau^{\mathrm{old}}]\!])]) \,\}] \tag{B.149}$$

$$\mathrm{V}[\{\, ([\![s]\!], \lambda_5) \,\}] \models [\![\mathrm{Post}(m)]\!] \tag{B.150}$$

Recall that in order for claim (B.108)-(B.113) to hold, (B.125)-(B.128) have to hold. This is now proven for $([\![s]\!], \lambda_5)$ and $\mathbf{h}_3$, which is given by (B.130). Left to prove with this choice:

$$\mathbf{h}_3 \overset{\pi'_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil \tag{B.151}$$

$$\begin{aligned}&\mathbf{h}^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto [\![\sigma]\!]([\![y]\!])]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto [\![\sigma]\!]([\![z_i]\!])]} = \\ &\qquad [\![\lceil \mathbf{h}_3, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor]\!]\end{aligned} \tag{B.152}$$

$$[\![\sigma]\!][[\![x]\!] \mapsto \mathbf{o}^\star] = [\![\sigma[x \mapsto \tau_1]]\!] \tag{B.153}$$

$$\pi^\star[(\mathbf{o}^\star, \mathit{recv}) \mapsto 1]\overline{[(\mathbf{o}^\star, \mathit{arg}_i) \mapsto 1]} = [\![\pi'_\tau]\!] \tag{B.154}$$

$$\text{where } \pi^\star = [\![\pi_\tau]\!] - \mathbf{Q}^{\mathrm{T}}([\![\mathrm{Pre}(m)]\!][[\![y]\!]/[\![\mathit{this}]\!]]\overline{[[\![z_i]\!]/[\![x_i]\!]]}, \lambda, \mathrm{True}),\ \mathbf{h}^\star \overset{\pi^\star}{\equiv} [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], \mathbf{o}^\star = \gamma(\tau_1)$$

(B.151) follows from a proof analogous to (B.42) and noting that $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h}$, as given by (B.130), and $\forall(\mathbf{o},f).\pi'_\tau(\mathbf{o},f) \Rightarrow \pi_\tau(\mathbf{o},f)$. (B.152) follows from a proof analogous to (B.47) and noting $\gamma^{-1}(\mathbf{o}^\star) = \tau_1$. (B.153) follows from a proof analogous to (B.70) and noting $\mathbf{o}^\star = \gamma(\tau_1)$. (B.154) holds if $\mathbf{Q}^{\mathrm{T}}([\![\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}]\!], \lambda, \mathrm{True}) = \mathbf{Z}(\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma)$. This follows from lemma B.19.

Thus, claim (B.108)-(B.113) holds for the active and forking thread τ. □

Newly forked thread τ_1:

Proof For the newly forked thread $((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]$ claim (B.108)-(B.113) reduces to:

$$([\![\mathrm{h}_1^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi_{\tau_1}^{\mathrm{old}'}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{B.155}$$

$$\exists([\![s_1]\!], \lambda_1) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}_1^{\mathrm{old}}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi_{\tau_1}^{\mathrm{old}'}]\!])]) \,\}], \tag{B.156}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_1}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.157}$$

$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor]\!], [\![\sigma_1]\!], [\![\pi'_{\tau_1}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.158}$$

$$\mathrm{V}[\{\, ([\![s_1]\!], \lambda_1) \,\}] \models [\![\mathrm{Post}(m)]\!] \tag{B.159}$$

$$\text{where } \mathrm{h}_1^{\mathrm{old}} = \lceil \mathbf{h}, \mathbf{r} \rfloor, m = \sigma_1(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_1(this)]\overline{[x_i \mapsto \sigma_1(x_i)]}$$

From (A3) it follows: $\sigma_1(this) = \mathbf{o} = \sigma(y)$ and $\overline{\sigma_1(x_i) = \sigma(z_i)}$. Thus, $\sigma^{\mathrm{old}'} = \sigma_1$. Also, from (A3): $s_1 = \mathrm{Body}(m)$. Moreover, by choice of permissions (B.116): $\pi_{\tau_1}^{\mathrm{old}'} = \mathbf{Z}(\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma)$ With this, claim (B.108)-(B.113) reduces for τ_1 to:

$$([\![\mathrm{h}_1^{\mathrm{old}}]\!], [\![\sigma_1]\!], [\![\mathbf{Z}(\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma)]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{B.160}$$

$$\exists([\![\mathrm{Body}(m)]\!], \lambda_1) \in \mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}_1^{\mathrm{old}}]\!], [\![\sigma_1]\!], [\![\mathbf{Z}(\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma)]\!])]) \,\}], \tag{B.161}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_1}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor \rceil. \tag{B.162}$$

$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)] \rfloor]\!], [\![\sigma_1]\!], [\![\pi'_{\tau_1}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.163}$$

$$\mathrm{V}[\{\, ([\![\mathrm{Body}(m)]\!], \lambda_1) \,\}] \models [\![\mathrm{Post}(m)]\!] \tag{B.164}$$

$$\text{where } \mathrm{h}_1^{\mathrm{old}} = \lceil \mathbf{h}, \mathbf{r} \rfloor, m = \sigma_1(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma_1 = [meth \mapsto m][this \mapsto \sigma_1(this)]\overline{[x_i \mapsto \sigma_1(x_i)]}$$

To prove (B.160), recall (B.139). For the partial trace given in (B.139), it holds with lemma B.12:

$$\lambda \models [\![\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}]\!] \tag{B.165}$$
$$\text{where } ([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!], [\![\sigma]\!], [\![\pi_\tau]\!]) \overset{(B.131)}{=} \mathrm{LAST}(\lambda)$$

From (B.165), and lemma B.16 it follows:

$$\Lambda[(\llbracket\lceil\mathbf{h}_3, \mathbf{r}\rfloor\rrbracket, \llbracket\sigma\rrbracket, \llbracket\pi_\tau\rrbracket)] \models \llbracket \mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}\rrbracket \overset{2,86}{\Leftrightarrow}$$
$$(\llbracket\lceil\mathbf{h}_3, \mathbf{r}\rfloor\rrbracket, \llbracket\sigma\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}\rrbracket \tag{B.166}$$

From (B.166) and lemma B.17 and lemma B.18:

$$(\llbracket\lceil\mathbf{h}_3, \mathbf{r}\rfloor\rrbracket, \llbracket[this \mapsto \sigma(y)]\overline{[x_i \mapsto \sigma(z_i)]}\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.167}$$

Moreover, as field $meth \notin \mathrm{FV}(\mathrm{Pre}(m))$:

$$(\llbracket\lceil\mathbf{h}_3, \mathbf{r}\rfloor\rrbracket, \llbracket[meth \mapsto m][this \mapsto \sigma(y)]\overline{[x_i \mapsto \sigma(z_i)]}\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.168}$$

From (B.168) and (A3):

$$(\llbracket\lceil\mathbf{h}_3, \mathbf{r}\rfloor\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.169}$$

From (B.169), (B.130) and lemma B.20 it follows:

$$(\llbracket\lceil\mathbf{h}, \mathbf{r}\rfloor\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.170}$$

From (B.170) and (A3) it follows:

$$(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_\tau\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.171}$$

From (B.171) and lemma B.22 it follows:

$$(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\mathbf{Z}(\mathrm{Pre}(m), \mathrm{h}_1^{\mathrm{old}}, \mathrm{h}_1^{\mathrm{old}}, \sigma_1)\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.172}$$

From (B.172) and (A3):

$$(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\mathbf{Z}(\mathrm{Pre}(m), \mathrm{h}, \mathrm{h}, \sigma_1)\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.173}$$

From (B.173), (A3) and lemma B.21:

$$(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\mathbf{Z}(\mathrm{Pre}(m)[y/this]\overline{[z_i/x_i]}, \mathrm{h}, \mathrm{h}, \sigma)\rrbracket) \models \llbracket \mathrm{Pre}(m)\rrbracket \tag{B.174}$$

Thus, claim (B.160) holds. Claim (B.161)-(B.164) is proven with $\Lambda[(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_{\tau_1}^{\mathrm{old}'}\rrbracket)]$ and $\mathbf{h}$, all given by (A3). Left to prove with this choice:

$$\mathbf{h} \overset{\pi'_{\tau_1}}{\equiv} \lfloor\lceil\mathbf{h}, \mathbf{r}\rfloor\rceil \tag{B.175}$$
$$\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket = \llbracket\lceil\mathbf{h}, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma[x \mapsto \tau_1], s), \tau)][\tau_1 \mapsto ((\mathrm{h}_1^{\mathrm{old}}, \sigma_1, s_1), \tau_1)]\rfloor\rrbracket \tag{B.176}$$
$$\mathrm{V}[\{\ (\llbracket\mathrm{Body}(m)\rrbracket, \Lambda[(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_{\tau_1}^{\mathrm{old}'}\rrbracket)])\ \}] \models \llbracket\mathrm{Post}(m)\rrbracket \tag{B.177}$$

(B.175) holds with a proof analogous to (B.42). (B.177) follows from assumption (A1) which can be triggered with (B.174). By assumption (A3): $\mathrm{h}_1^{\mathrm{old}} = \lceil\mathbf{h}, \mathbf{r}\rfloor$. (B.176) follows from a proof analogous to (B.47) and by observing that $\Lambda[(\llbracket \mathrm{h}_1^{\mathrm{old}}\rrbracket, \llbracket\sigma_1\rrbracket, \llbracket\pi_{\tau_1}^{\mathrm{old}'}\rrbracket)]$ has no permission to access fields *recv* or arg_i of object $\mathbf{o}^\star = \gamma(\tau_1)$. Thus, any query for such a field in the right-hand side heap of (B.176) will be forwarded to $\mathbf{r}$.

Thus, claim (B.160)-(B.164) holds for the newly forked thread τ_1. □

Arbitrary active thread $\tau_2 \notin \{ \tau, \tau_1 \}$**:**

Proof Analogous to proof of claim (B.4)-(B.9) and the subproof considering the arbitrary active but non-executing thread. □

Thus, claim (B.160)-(B.164) holds for an arbitrary active thread. □

Case $x.f := y$: In this case, assumption 3, $\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}_1$, (A3) has the following derivation tree:

$$\frac{\mathbf{r}[\tau] = ((\mathrm{h}^{\mathrm{old}}, \sigma, (x.f := y; s)), \tau) \quad \mathrm{h} = \lceil \mathbf{h}, \mathbf{r} \rfloor \quad \dfrac{\sigma(x) = \mathbf{o} \quad \mathrm{h}_1 = \mathrm{h}[(\mathbf{o}, f) \mapsto \sigma(y)]}{\mathrm{h}, ((\mathrm{h}^{\mathrm{old}}, \sigma, (x.f := y; s)), \tau) \rightharpoonup \mathrm{h}_1, ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)} \quad \mathbf{h}_1 = \lfloor \mathrm{h}_1 \rceil}{\mathbf{h}, \mathbf{r} \rightsquigarrow \mathbf{h}_1, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]}$$

With $\mathbf{h}_1 = \lfloor \mathrm{h}_1 \rceil = \lfloor \mathrm{h}[(\mathbf{o}, f) \mapsto \sigma(y)] \rceil = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$ and $\mathbf{r}_1 = \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$, the conclusion to prove is as follows.

$$VR(prog, \mathbf{h}_1, \mathbf{r}_1) \;\Leftrightarrow\; VR(prog, \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]) \;\Leftrightarrow\; \exists \overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)}. \tag{B.178}$$

$$\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]} \overline{\pi'_\tau} \wedge \tag{B.179}$$

$$\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]).\ \pi'_{\tau_2} = \pi_\varnothing \wedge \tag{B.180}$$

$$\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]. \tag{B.181}$$

$$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \tag{B.182}$$

$$\exists ([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])])\ \}], \tag{B.183}$$

$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil. \tag{B.184}$$

$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.185}$$

$$\mathrm{V}[\{\ ([\![s_4]\!], \lambda_1)\ \}] \models [\![\mathrm{Post}(m)]\!] \tag{B.186}$$

where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth),\ \overline{x_i} = \mathrm{Param}(m)$,
$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$

The case is proven by taking the same sequence of permission masks as given by assumption (A2.0):

$$\overline{(\pi_\tau^{\mathrm{old}'}, \pi'_\tau)} = \overline{(\pi_\tau^{\mathrm{old}}, \pi_\tau)} \tag{B.187}$$

It remains to be shown that the choice (B.187) satisfies conclusions (B.179), (B.180) and (B.181)-(B.186).

Claim (B.179)**:** $\models_{\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]} \overline{\pi'_\tau}$

Proof Analogous to the proof of (B.2) for statement **var** $x : t$. □

Claim (B.180)**:** $\forall \mathbf{e}_{\tau_2} \in idle(\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]).\ \pi^2_{\tau_2} = \pi_\varnothing$

Proof Analogous to the proof of (B.3) for statement **var** $x : t$. □

Claim (B.181)**-**(B.186)**:** $\forall ((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$.
$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge$
$\exists ([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{\ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])])\ \}], \exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$.
$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \mathrm{V}[\{\ ([\![s_4]\!], \lambda_1)\ \}] \models [\![\mathrm{Post}(m)]\!]$
where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth),\ \overline{x_i} = \mathrm{Param}(m), \sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$

Proof Claim (B.181)-(B.186) quantifies over the set of active threads in runtime-entity collection $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$. According to (A3), this set of active threads can be partitioned as follows: a) thread τ which executed statement $x.f := y$ and b) all other active threads $\tau_2 \neq \tau$. Claim (B.181)-(B.186) is proven if it holds for thread τ as well as for an arbitrary thread chosen from partition b).

Active and executing thread τ:

Proof For the active and executing thread $((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$ claim (B.4)-(B.9) reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.188}$$

$$\exists (\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}'} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}'} \rrbracket)]) \,\}], \tag{B.189}$$

$$\exists \mathbf{h}_4 \overset{\pi'_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil. \tag{B.190}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi'_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.191}$$

$$\mathrm{V}[\{\, (\llbracket s \rrbracket, \lambda_1) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.192}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}$$

By choice of permissions (B.187): $\pi'_\tau = \pi_\tau$ and $\pi_\tau^{\mathrm{old}'} = \pi_\tau^{\mathrm{old}}$. Furthermore, $\sigma^{\mathrm{old}'} = \sigma^{\mathrm{old}}$, given by assumption (A2.3)-(A2.7). Claim (B.181)-(B.186) thus reduces for τ to:

$$(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.193}$$

$$\exists (\llbracket s \rrbracket, \lambda_1) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)]) \,\}], \tag{B.194}$$

$$\exists \mathbf{h}_4 \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil. \tag{B.195}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.196}$$

$$\mathrm{V}[\{\, (\llbracket s \rrbracket, \lambda_1) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.197}$$

$$\text{where } \mathrm{h}^{\mathrm{old}} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}} = [meth \mapsto m][this \mapsto \sigma(this)]\overline{[x_i \mapsto \sigma(x_i)]}$$

(B.193) holds by assumption (A2.4). Assumptions (A2.5)-(A2.7) promise the existence of a partial trace $(\llbracket x.f := y; s \rrbracket, \lambda)$ and a runtime heap $\mathbf{h}_3$ for which it holds:

$$(\llbracket x.f := y; s \rrbracket, \lambda) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi_\tau^{\mathrm{old}} \rrbracket)]) \,\}] \tag{B.198}$$

$$\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h} \tag{B.199}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma \rrbracket, \llbracket \pi_\tau \rrbracket) = \mathrm{LAST}(\lambda) \tag{B.200}$$

$$\mathrm{V}[\{\, (\llbracket x.f := y; s \rrbracket, \lambda) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.201}$$

By definition of statement encoding $[\![\_]\!]$ given in 4.3:

$$([\![x.f := y; s]\!], \lambda) = ([\![x]\!].f := [\![y]\!]; [\![s]\!], \lambda) \tag{B.202}$$

From (B.201) and (B.202), the Viper rule of unconditional error-trace propagation (2.63), the definition of the Viper closure V[] given in 2.82 and the definition of $\models$ given in (2.88) it follows that the field assignment in (B.202) has to succeed:

$$(\mathrm{C}[[\![x]\!].f := [\![y]\!]], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}], [\![\sigma]\!], [\![\pi_\tau]\!])) \tag{B.203}$$

$$\text{where } \mathrm{C} = \bullet; [\![s]\!], \mathbf{v} = [\![\sigma]\!]([\![y]\!]) = [\![\sigma(y)]\!], \mathbf{o}_1 = [\![\sigma]\!]([\![x]\!]) = [\![\sigma(x)]\!] \overset{(A3)}{=} [\![\mathbf{o}]\!] = \gamma(\mathbf{o})$$

Moreover:

$$[\![\pi_\tau]\!](\mathbf{o}_1, f) \geq 1 \tag{B.204}$$

(B.204) will be important when analysing an arbitrary but non-executing thread τ_2 as it implies $[\![\pi_{\tau_2}]\!](\mathbf{o}_1, f) = 0$ for all $\tau_2 \neq \tau$. By definition 2.26 of the Viper evaluation context it follows from (B.203):

$$(\mathrm{C}[\varepsilon], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}], [\![\sigma]\!], [\![\pi_\tau]\!])) = ([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}], [\![\sigma]\!], [\![\pi_\tau]\!])) \tag{B.205}$$

$$\text{where } \mathbf{v} = [\![\sigma]\!]([\![y]\!]) = [\![\sigma(y)]\!], \mathbf{o}_1 = [\![\sigma]\!]([\![x]\!]) = [\![\sigma(x)]\!] \overset{(A3)}{=} [\![\mathbf{o}]\!] = \gamma(\mathbf{o})$$

Recall in order for (B.181)-(B.186) to hold, (B.194)-(B.197) have to hold. This is now proven for $([\![s]\!], \lambda{:}([\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}], [\![\sigma]\!], [\![\pi_\tau]\!]))$ and $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)]$, where $\mathbf{h}_3$ is given by (B.199).In order for (B.194)-(B.197) to hold with this choice it has to hold: $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)] \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$ and $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}] = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!]$.

Claim: $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)] \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$

Proof Recall (B.199): $\mathbf{h}_3 \overset{\pi_\tau}{\equiv} \mathbf{h}$ and assume $\pi_\tau = (\pi^{\mathbf{F}}, \pi^{\mathbf{P}})$. Then, by definition of $\overset{\pi_\tau}{\equiv}$ given in 2.25:

$$\text{(B.199)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}_2, f_2) = \mathbf{h}(\mathbf{o}_2, f_2) \tag{B.206}$$

Using the definition of $\lceil \_, \_ \rfloor$ given in 3.29:

$$\text{(B.206)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}_2, f_2) = \lceil \mathbf{h}, \mathbf{r} \rfloor (\mathbf{o}_2, f_2) \tag{B.207}$$

Furthermore:

$$\text{(B.207)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) = \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) \tag{B.208}$$

Using the definition of $\lfloor \_ \rceil$ given in 3.29:

$$\text{(B.208)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil (\mathbf{o}_2, f_2) \tag{B.209}$$

The definition of $\overset{\pi_\tau}{\equiv}$ given in 2.25 then proves the claim:

$$\text{(B.209)} \Leftrightarrow \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)] \overset{\pi_\tau}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil \tag{B.210}$$

□

Claim: $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}] = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!]$

Proof By definition of function equality:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}] = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!] \Leftrightarrow \tag{B.211}$$
$$\forall (\mathbf{o}_2, f_2).\ [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2)$$

Recall from the semantics of Chalice: $f \notin \{ recv, \overline{arg_i} \}$. The claim is then proven by a case analysis of $(\mathbf{o}_2, f_2)$.

Case $(\mathbf{o}_2, f_2) = (\mathbf{o}_1, f)$
Then:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_1, f) = \mathbf{v} \tag{B.212}$$

Note: $f_2 \notin \{ recv, \overline{arg_i}$. Also recall from (B.205): $\mathbf{o}_1 = \gamma(\mathbf{o})$. Then, by definition of heap encoding given in (4.14):

$$\begin{aligned} &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) = \\ &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_1, f) = \\ &[\![\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\gamma^{-1}(\mathbf{o}_1), f)]\!] = [\![\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\gamma^{-1}(\gamma(\mathbf{o})), f)]\!] = \\ &[\![\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}, f)]\!] = [\![\sigma(y)]\!] \overset{(B.205)}{=} \mathbf{v} \end{aligned} \tag{B.213}$$

Thus, from (B.212) and (B.213):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) \tag{B.214}$$

Case $\mathbf{o}_2 = \mathbf{o}_1 \wedge f_2 \neq f \wedge f_2 \in \{ recv, \overline{arg_i} \}$
Assume $f_2 = recv$. The complementary cases can be shown analogously to the following argument. Then:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_1, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_1, f_2) \tag{B.215}$$

By definition of heap encoding given in (4.14):

$$\begin{aligned} &[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_1, f_2) = [\![\sigma_6(this)]\!] \\ &\text{where } ((\mathrm{h}_3, \sigma_6, s_8), \tau_3) \in \mathbf{r} \text{ s.t. } \tau_3 = \gamma^{-1}(\mathbf{o}_1) \end{aligned} \tag{B.216}$$

In Chalice, variable *this* is read-only. Furthermore, by assumption (A3), no active thread was reset to idle. Thus, thread τ_3 still exists in $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$ with the same value for *this* in σ_6:

$$[\![\sigma_6(this)]\!] = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_1, f_2) \tag{B.217}$$

Thus, from (B.215), (B.216) and (B.217):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) \tag{B.218}$$

Case $\mathbf{o}_2 = \mathbf{o}_1 \wedge f_2 \neq f \wedge f_2 \notin \{ recv, \overline{arg_i} \}$
Then:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_1, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_1, f_2) \tag{B.219}$$

Then, by definition of heap encoding given in (4.14):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_1, f_2) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_1), f_2)]\!] \overset{(B.205)}{=} [\![\mathbf{h}_3(\gamma^{-1}(\gamma(\mathbf{o})), f_2)]\!] = [\![\mathbf{h}_3(\mathbf{o}, f_2)]\!] \tag{B.220}$$

Thus, from (B.219) and (B.220):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\mathbf{h}_3(\mathbf{o}, f_2)]\!] \tag{B.221}$$

On the other hand:

$$\begin{aligned} &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) = \\ &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_1, f_2) \end{aligned} \tag{B.222}$$

Then, by definition of heap encoding given in (4.14):

$$[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_1, f_2) = [\![\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\gamma^{-1}(\mathbf{o}_1), f_2)]\!] \overset{f \neq f_2}{=} \tag{B.223}$$

$$[\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_1), f_2)]\!] \overset{(B.205)}{=} [\![\mathbf{h}_3(\gamma^{-1}(\gamma(\mathbf{o})), f_2)]\!] = [\![\mathbf{h}_3(\mathbf{o}, f_2)]\!]$$

Thus, from (B.222) and (B.223):

$$[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) = [\![\mathbf{h}_3(\mathbf{o}, f_2)]\!] \tag{B.224}$$

Thus, from (B.221) and (B.224):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) \tag{B.225}$$

Case $\mathbf{o}_2 \neq \mathbf{o}_1 \wedge f_2 = f$
Note from the semantics of Chalice: $f_2 \notin \{ recv, \overline{arg_i} \}$. Then:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f_2) = [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1, f) \mapsto \mathbf{v}](\mathbf{o}_2, f) \overset{\mathbf{o}_1 \neq \mathbf{o}_2}{=} [\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_2, f) \tag{B.226}$$

As $f_2 \notin \{ recv, \overline{arg_i} \}$, it follows from the definition of heap encoding given in (4.14):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_2, f) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2), f)]\!] \tag{B.227}$$

On the other hand:

$$\begin{aligned} &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f_2) = \\ &[\![\lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2, f) \end{aligned} \tag{B.228}$$

As $f_2 \notin \{ recv, \overline{arg_i} \}$, it follows from the definition of heap encoding given in (4.14):

$$[\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2,f) = \quad \text{(B.229)}$$
$$[\![\mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)](\gamma^{-1}(\mathbf{o}_2),f)]\!]$$

Recall from (B.205): $\mathbf{o}_1 = \gamma(\mathbf{o})$. Thus: $\mathbf{o} = \gamma^{-1}(\mathbf{o}_1)$. From this and $\mathbf{o}_1 \neq \mathbf{o}_2$: $\mathbf{o} \neq \gamma^{-1}(\mathbf{o}_2)$. From this and (B.229):

$$[\![\mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)](\gamma^{-1}(\mathbf{o}_2),f)]\!] = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2),f)]\!] \quad \text{(B.230)}$$

From (B.228), (B.229) and (B.230):

$$[\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2,f_2) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2),f)]\!] \quad \text{(B.231)}$$

From this and (B.226):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!][(\mathbf{o}_1,f) \mapsto \mathbf{v}](\mathbf{o}_2,f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!](\mathbf{o}_2,f_2) \quad \text{(B.232)}$$

Case $\mathbf{o}_2 \neq \mathbf{o}_1 \ \wedge f_2 \neq f \wedge f_2 \in \{ recv, \overline{arg_i} \}$
Analogously to the previous cases.
Case $\mathbf{o}_2 \neq \mathbf{o}_1 \ \wedge f_2 \neq f \wedge f_2 \notin \{ recv, \overline{arg_i} \}$
Analogously to the previous cases.

Thus, the claim holds. □

Thus, conclusion (B.181)-(B.186) holds for the active and executing thread. □

Arbitrary active but non-executing thread $\tau_2 \neq \tau$:

Proof By chosing $((\mathrm{h}^{\mathrm{old}}_{\tau_2}, \sigma_3, s_4), \tau_2) \in \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$ with $\tau_2 \neq \tau$, claim (B.181)-(B.186) reduces to:

$$([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!]) \models [\![\mathrm{Pre}(m)]\!] \wedge \quad \text{(B.233)}$$
$$\exists([\![s_4]\!], \lambda_1) \in \mathrm{V}[\{ ([\![\mathrm{Body}(m)]\!], \Lambda[([\![\mathrm{h}^{\mathrm{old}}_{\tau_2}]\!], [\![\sigma^{\mathrm{old}'}]\!], [\![\pi^{\mathrm{old}'}_{\tau_2}]\!])]) \}], \quad \text{(B.234)}$$
$$\exists \mathbf{h}_4 \overset{\pi'_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o},f) \mapsto \sigma(y)] \rceil. \quad \text{(B.235)}$$
$$([\![\lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor]\!], [\![\sigma_3]\!], [\![\pi'_{\tau_2}]\!]) = \mathrm{LAST}(\lambda_1) \wedge \quad \text{(B.236)}$$
$$\mathrm{V}[\{ ([\![s_4]\!], \lambda_1) \}] \models [\![\mathrm{Post}(m)]\!] \quad \text{(B.237)}$$
where $\mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_3(meth), \ \overline{x_i} = \mathrm{Param}(m),$
$$\sigma^{\mathrm{old}'} = [meth \mapsto m][this \mapsto \sigma_3(this)]\overline{[x_i \mapsto \sigma_3(x_i)]}$$

By choice of permissions (B.187): $\pi'_{\tau_2} = \pi_{\tau_2}$ and $\pi^{\mathrm{old}'}_{\tau_2} = \pi^{\mathrm{old}}_{\tau_2}$. Furthermore, as τ_2 is active but non-executing: $\sigma_3 = \sigma_2$, $s_4 = s_3$, and $\sigma^{\mathrm{old}'} = \sigma^{\mathrm{old}}$, all given by assumption (A2.3). Claim

(B.181)-(B.186) thus reduces to:

$$(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi^{\mathrm{old}}_{\tau_2} \rrbracket) \models \llbracket \mathrm{Pre}(m) \rrbracket \wedge \tag{B.238}$$

$$\exists (\llbracket s_3 \rrbracket, \lambda_1) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi^{\mathrm{old}}_{\tau_2} \rrbracket)]) \,\}], \tag{B.239}$$

$$\exists \mathbf{h}_4 \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil. \tag{B.240}$$

$$(\llbracket \lceil \mathbf{h}_4, \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor \rrbracket, \llbracket \sigma_2 \rrbracket, \llbracket \pi_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda_1) \wedge \tag{B.241}$$

$$\mathrm{V}[\{\, (\llbracket s_3 \rrbracket, \lambda_1) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.242}$$

$$\text{where } \mathrm{h}^{\mathrm{old}}_{\tau_2} = \lceil \mathbf{h}_2, \mathbf{r}_2 \rfloor, m = \sigma_2(meth),\ \overline{x_i} = \mathrm{Param}(m),$$
$$\sigma^{\mathrm{old}} = [meth \mapsto m][this \mapsto \sigma_2(this)]\overline{[x_i \mapsto \sigma_2(x_i)]}$$

(B.238) holds by assumption (A2.4). To prove (B.239)-(B.242), the partial trace $(\llbracket s_3 \rrbracket, \lambda)$ and the runtime heap $\mathbf{h}_3$, promised by assumption (A2.5), are now taken. Thus, assume:

$$(\llbracket s_3 \rrbracket, \lambda) \in \mathrm{V}[\{\, (\llbracket \mathrm{Body}(m) \rrbracket, \Lambda[(\llbracket \mathrm{h}^{\mathrm{old}}_{\tau_2} \rrbracket, \llbracket \sigma^{\mathrm{old}} \rrbracket, \llbracket \pi^{\mathrm{old}}_{\tau_2} \rrbracket)]) \,\}] \tag{B.243}$$

$$\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h} \tag{B.244}$$

$$(\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket, \llbracket \sigma_2 \rrbracket, \llbracket \pi_{\tau_2} \rrbracket) = \mathrm{LAST}(\lambda) \tag{B.245}$$

$$\mathrm{V}[\{\, (\llbracket s_3 \rrbracket, \lambda) \,\}] \models \llbracket \mathrm{Post}(m) \rrbracket \tag{B.246}$$

(B.239)-(B.242) can now be proven for $(\llbracket s_3 \rrbracket, \lambda)$ and $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)]$. In order for (B.239)-(B.242) being proven, it has to hold: $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)] \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$ and $\llbracket \lceil \mathbf{h}_3, \mathbf{r} \rfloor \rrbracket = \llbracket \lceil \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)] \rfloor \rrbracket$.

Claim: $\mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)] \overset{\pi_{\tau_2}}{\equiv} \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil$

Proof Recall (B.244): $\mathbf{h}_3 \overset{\pi_{\tau_2}}{\equiv} \mathbf{h}$ and assume $\pi_{\tau_2} = (\pi^{\mathbf{F}}_{\tau_2}, \pi^{\mathbf{P}}_{\tau_2})$. Then, by definition of $\overset{\pi_{\tau_2}}{\equiv}$ given in 2.25:

$$\text{(B.244)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}_{\tau_2}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}_2, f_2) = \mathbf{h}(\mathbf{o}_2, f_2) \tag{B.247}$$

By definition of mapping runtime-heaps to heaps $\lceil \_, \_ \rfloor$ given in 3.29:

$$\text{(B.247)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}_{\tau_2}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3(\mathbf{o}_2, f_2) = \lceil \mathbf{h}, \mathbf{r} \rfloor (\mathbf{o}_2, f_2) \tag{B.248}$$

Moreover:

$$\text{(B.248)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}_{\tau_2}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) = \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) \tag{B.249}$$

Then by definition of mapping heaps to runtime-heaps $\lfloor \rceil$ given in 3.29:

$$\text{(B.249)} \Leftrightarrow \forall (\mathbf{o}_2, f_2).\, \pi^{\mathbf{F}}_{\tau_2}(\mathbf{o}_2, f_2) > 0 \Rightarrow \mathbf{h}_3[(\mathbf{o}, f) \mapsto \sigma(y)](\mathbf{o}_2, f_2) = \lfloor \lceil \mathbf{h}, \mathbf{r} \rfloor [(\mathbf{o}, f) \mapsto \sigma(y)] \rceil (\mathbf{o}_2, f_2) \tag{B.250}$$

Thus, by definition of $\overset{\pi_{\tau_2}}{\equiv}$ given in 2.25:

$$\mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)] \overset{\pi_{\tau_2}}{\equiv} \lfloor\lceil \mathbf{h}, \mathbf{r}\rfloor[(\mathbf{o},f) \mapsto \sigma(y)]\rceil \tag{B.251}$$

□

Claim: $[\![\lceil \mathbf{h}_3, \mathbf{r}\rfloor]\!] = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]\rfloor]\!]$

Proof By definition of function equality:

$$\begin{aligned}&[\![\lceil \mathbf{h}_3, \mathbf{r}\rfloor]\!] = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]\rfloor]\!] \Leftrightarrow \\ &\forall(\mathbf{o}_2,f_2).\ [\![\lceil \mathbf{h}_3, \mathbf{r}\rfloor]\!](\mathbf{o}_2,f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]\rfloor]\!](\mathbf{o}_2,f_2)\end{aligned} \tag{B.252}$$

The claim is then proven by a case analysis of f_2.

Case $f_2 = recv$
Then by definition of heap encoding $[\![\_]\!]$ given in 4.14:

$$\begin{aligned}&[\![\lceil \mathbf{h}_3, \mathbf{r}\rfloor]\!](\mathbf{o}_2,f_2) = [\![\sigma_6(this)]\!] \\ &\text{where } ((\mathrm{h}_3, \sigma_6, s_8), \tau_3) \in \mathbf{r} \text{ s.t. } \tau_3 = \gamma^{-1}(\mathbf{o}_2)\end{aligned} \tag{B.253}$$

In Chalice, variable *this* is read-only. Furthermore, by assumption (A3), no active thread was reset to idle. Thus, thread τ_3 still exists in $\mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]$ with the same value for *this* in σ_6:

$$[\![\sigma_6(this)]\!] = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}}, \sigma, s), \tau)]\rfloor]\!](\mathbf{o}_2,f_2) \tag{B.254}$$

Case $f_2 = arg_i$
Similar to the case for *recv*.

Case $f_2 \notin \{ recv, \overline{arg_i} \}$
The semantics of Viper for a field-access, which is the context in which a heap-lookup $[\![\lceil \mathbf{h}_3, \mathbf{r}\rfloor]\!](\mathbf{o}_2,f_2)$ occurs, requires for thread τ_2 to have enough permission to field f_2 of object $\mathbf{o}_2$:

$$[\![\pi_{\tau_2}]\!](\mathbf{o}_2,f_2) \geq 1 \tag{B.255}$$

As $f_2 \notin \{ recv, \overline{arg_i} \}$, it follows from the definition of permission-mask encoding given in 4.13 and the definition of value encoding given in 4.11:

$$\text{(B.255)} \Leftrightarrow [\![\frac{\pi^{\mathbf{F}}_{\tau_2}(\gamma^{-1}(\mathbf{o}_2),f_2)}{100}]\!] \geq 1 \Leftrightarrow \pi^{\mathbf{F}}_{\tau_2}(\gamma^{-1}(\mathbf{o}_2),f_2) \geq 100 \tag{B.256}$$

Recall (B.204): $[\![\pi_\tau]\!](\mathbf{o}_1,f) \geq 1$ with $\pi_\tau = (\pi_\tau^{\mathbf{F}}, \pi_\tau^{\mathbf{P}})$. Furthermore, from (B.203): $\mathbf{o}_1 = \gamma(\mathbf{o})$. Thus, with the definition of permission-mask encoding given in 4.13 and the definition of value encoding given in 4.11:

$$[\![\pi_\tau]\!](\mathbf{o}_1,f) \geq 1 \;\Leftrightarrow\; [\![\frac{\pi_\tau^{\mathbf{F}}(\gamma^{-1}(\mathbf{o}_1),f)}{100}]\!] \geq 1 \Leftrightarrow \frac{\pi_\tau^{\mathbf{F}}(\gamma^{-1}(\gamma(\mathbf{o})),f)}{100} \geq 1 \Leftrightarrow \pi_\tau^{\mathbf{F}}(\mathbf{o},f) \geq 100 \tag{B.257}$$

Now, as $f_2 \notin \{\, recv, \overline{arg_i} \,\}$, it follows from the definition of heap encoding $[\![\_]\!]$ given in 4.14:

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_2,f_2) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2),f_2)]\!] \tag{B.258}$$

Similarly:

$$[\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}},\sigma,s),\tau)] \rfloor]\!](\mathbf{o}_2,f_2) = [\![\mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)](\gamma^{-1}(\mathbf{o}_2),f_2)]\!] \tag{B.259}$$

From (B.256) and (B.257) it follows:

$$(\mathbf{o},f) \neq (\gamma^{-1}(\mathbf{o}_2),f_2) \tag{B.260}$$

Thus:

$$\text{(B.259)} \;\Leftrightarrow\; [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2),f_2)]\!] \tag{B.261}$$

From (B.259) and (B.261):

$$[\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}},\sigma,s),\tau)] \rfloor]\!](\mathbf{o}_2,f_2) = [\![\mathbf{h}_3(\gamma^{-1}(\mathbf{o}_2),f_2)]\!] \tag{B.262}$$

Thus, from (B.258) and (B.262):

$$[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!](\mathbf{o}_2,f_2) = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}},\sigma,s),\tau)] \rfloor]\!](\mathbf{o}_2,f_2) \tag{B.263}$$

□

Thus, the claim $[\![\lceil \mathbf{h}_3, \mathbf{r} \rfloor]\!] = [\![\lceil \mathbf{h}_3[(\mathbf{o},f) \mapsto \sigma(y)], \mathbf{r}[\tau \mapsto ((\mathrm{h}^{\mathrm{old}},\sigma,s),\tau)] \rfloor]\!]$ holds. □

Thus, conclusion (B.181)-(B.186) holds for an arbitrary active but non-executing thread. □

Thus, conclusion (B.181)-(B.186) holds for an arbitrary active thread. □

Auxiliary

Lemma B.12 $\forall a, \lambda, \lambda_1.\, (\mathrm{C}[\mathbf{exhale}\ a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_1) \wedge \lambda_1 \notin \Lambda_{\mathrm{Err}} \Rightarrow \lambda \models a$

Proof By structural induction on the Chalice assertion a. □

Lemma B.13 $\forall a, \lambda, \lambda_1.\, (\mathrm{C}[\mathbf{exhale}\ a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_1) \wedge \lambda_1 \notin \Lambda_{\mathrm{Err}} \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \wedge (\mathbf{h}_1, \sigma_1, \pi_1) = \mathrm{LAST}(\lambda_1) \Rightarrow \sigma_1 = \sigma$

Proof By structural induction on the Chalice assertion a. □

Lemma B.14 $\forall a, \lambda, \lambda_1.\, (\mathrm{C}[\textbf{exhale } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_1) \wedge \lambda_1 \notin \Lambda_{\mathrm{Err}} \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \wedge (\mathbf{h}_1, \sigma_1, \pi_1) = \mathrm{LAST}(\lambda_1) \Rightarrow \pi_1 = \pi - \mathbf{Q}^{\mathrm{T}}(a, \lambda, \mathrm{True})$

Proof By structural induction on the Chalice assertion a. □

Lemma B.15 $\forall a, \lambda, \lambda_1.\, (\mathrm{C}[\textbf{exhale } a], \lambda) \rightsquigarrow (\mathrm{C}[\varepsilon], \lambda_1) \wedge \lambda_1 \notin \Lambda_{\mathrm{Err}} \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \wedge (\mathbf{h}_1, \sigma_1, \pi_1) = \mathrm{LAST}(\lambda_1) \Rightarrow \mathbf{h}_1 \overset{\pi_1}{\equiv} \mathbf{h}$

Proof By structural induction on the Chalice assertion a and the observation that whenever $\mathbf{G}^{\mathrm{T}}(\pi_1, \lambda) = \pi_2$ then $\pi_1 \leq \pi_2$. This argument is needed when the induction proof reaches the exhale of a predicate. □

Lemma B.16 $\forall m, \lambda.\, \lambda \models [\![\mathrm{Pre}(m)]\!] \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \Rightarrow \Lambda[(\mathbf{h}, \sigma, \pi)] \models [\![\mathrm{Pre}(m)]\!]$

Proof By structural induction on the Chalice precondition $\mathrm{Pre}(m)$ and realising that a Chalice precondition does not contain **old** expressions. □

Lemma B.17 $\forall a, \lambda.\, \lambda \models a \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \wedge \overline{x_i} = \mathrm{FV}(a) \Rightarrow \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \overline{[x_i \mapsto \sigma(x_i)]}, \pi) \models a$

Proof By structural induction on assertion a. □

Lemma B.18 $\forall a, \lambda.\, \lambda \models a\overline{[x_i / y_i]} \wedge (\mathbf{h}, \sigma, \pi) = \mathrm{LAST}(\lambda) \Rightarrow \lambda[1, |\lambda| - 1]{:}(\mathbf{h}, \sigma\overline{[y_i \mapsto \sigma(x_i)]}, \pi) \models a$

Proof By structural induction on the Chalice precondition $\mathrm{Pre}(m)$ and realising that a Chalice precondition does not contain **old** expressions. □

Lemma B.19 $\forall m, \lambda.\, ([\![\mathrm{h}]\!], [\![\sigma]\!], [\![\pi]\!]) = \mathrm{LAST}(\lambda) \wedge \mathrm{h}_1 \overset{\pi}{\equiv} \mathrm{h} \Rightarrow \mathbf{Q}^{\mathrm{T}}([\![\mathrm{Pre}(m)]\!], \lambda, \mathrm{True}) = [\![\mathbf{Z}(\mathrm{Pre}(m), \mathrm{h}_1, \mathrm{h}_1, \sigma)]\!]$

Proof By structural induction on the Chalice precondition $\mathrm{Pre}(m)$, realising that a Chalice precondition does not contain **old** expressions and using lemma B.10 and lemma B.2 when the induction reaches the case of a conditional assertion. □

Lemma B.20 $\forall a.\, ([\![\lceil \mathbf{h}, \mathbf{r} \rfloor]\!], [\![\sigma]\!], [\![\pi]\!]) \models [\![a]\!] \wedge \mathbf{h}_1 \overset{\pi}{\equiv} \mathbf{h} \Rightarrow ([\![\lceil \mathbf{h}_1, \mathbf{r} \rfloor]\!], [\![\sigma]\!], [\![\pi]\!]) \models [\![a]\!]$

Proof By structural induction on the Chalice assertion a. □

Lemma B.21
$\forall m, \sigma, \sigma_1.\, \overline{x_i} = \mathrm{FV}(\mathrm{Pre}(m)) \wedge \overline{\sigma_1(y_i) = \sigma(x_i)} \wedge \mathbf{Z}(\mathrm{Pre}(m), \mathrm{h}, \mathrm{h}, \sigma) = \pi \Rightarrow \mathbf{Z}(\mathrm{Pre}(m)\overline{[y_i / x_i]}, \mathrm{h}, \mathrm{h}, \sigma_1) = \pi$

Proof By structural induction on the Chalice assertion a. □

Lemma B.22 $\forall m.\, ([\![\mathrm{h}]\!], [\![\sigma]\!], [\![\pi]\!]) \models [\![\mathrm{Pre}(m)]\!] \Rightarrow ([\![\mathrm{h}]\!], [\![\sigma]\!], [\![\mathbf{Z}(\mathrm{Pre}(m), \mathrm{h}, \mathrm{h}, \sigma)]\!]) \models [\![\mathrm{Pre}(m)]\!]$

Proof By structural induction on the Chalice precondition $\mathrm{Pre}(m)$ and realising that a Chalice precondition doest not contain **old** expressions. □

Appendix C

Viper: Concrete Syntax

This appendix defines the concrete syntax of Viper by means of specifying a context-free grammar using an EBNF notation.

Contents

C.1 Notation

Nonterminals are indicated by *italic type*, whereas any terminals are indicated by **"quoted bold type"**. The production rules make use of the following operators: a *bar* | represents an alternative, a *comma* , is used for concatenation, *brackets* () are used for grouping, a *semicolon* ; marks the end of the production rule, a *star* $\star$ is used to indicate zero or more repetitions, a *plus* + indicates one or more repetitions and a *question-mark* ? indicates an optional occurrence.

C.2 Concrete Syntax

C.2.1 Keywords

keyword := **"return"** | **"Int"** | **"Perm"** | **"Bool"** | **"Ref"** | **"Rational"** |
"true" |**"false"** | **"null"** | **"import"** | **"method"** | **"function"** |
"predicate" | **"program"** | **"domain"** | **"axiom"** | **"var"** |
"returns" | **"field"** | **"define"** | **"wand"** | **"requires"** |
"ensures" | **"invariant"** | **"fold"** | **"unfold"** | **"inhale"** |
"exhale" | **"new"** | **"assert"** | **"assume"** | **"package"** | **"apply"** |
"while" | **"if"** | **"elseif"** | **"else"** | **"goto"** | **"label"** | **"fresh"** |
"constraining" | **"Seq"** | **"Set"** | **"Multiset"** | **"union"** |
"intersection" | **"setminus"** | **"subset"** | **"unfolding"** | **"in"**
| **"folding"** | **"applying"** | **"packaging"** | **"old"** | **"lhs"** | **"let"** |
"forall" | **"exists"** | **"forperm"** | **"acc"** | **"wildcard"** | **"write"** |
"none" | **"epsilon"** | **"perm"** | **"unique"** ;

C.2.2 Identifiers

identifier := (*letter* | **"_"**) , (*letter* | *digit* | **"_"**)* ;

letter := *lower* | *upper* ;

lower := **"a"** | **"b"** | **"c"** | **"d"** | **"e"** | **"f"** | **"g"** | **"h"** | **"i"** | **"j"** |
"k" | **"l"** | **"m"** | **"n"** | **"o"** | **"p"** | **"q"** | **"r"** | **"s"** | **"t"** |
"u" | **"v"** | **"w"** | **"x"** | **"y"** | **"z"** ;

upper := **"A"** | **"B"** | **"C"** | **"D"** | **"E"** | **"F"** | **"G"** | **"H"** | **"I"** |
"J" | **"K"** | **"L"** | **"M"** | **"N"** | **"O"** | **"P"** | **"Q"** | **"R"** | **"S"** |
"T" | **"U"** | **"V"** | **"W"** | **"X"** | **"Y"** | **"Z"** ;

digit := **"0"** | **"1"** | **"2"** | **"3"** | **"4"** | **"5"** | **"6"** | **"7"** | **"8"** | **"9"** ;

C.2.3 Literals

literal := *null_literal* | *boolean_literal* | *integer_literal* | *permission_literal* ;

null_literal := **"null"** ;

boolean_literal := **"false"** | **"true"** ;

integer_literal := *digit*+ ;

permission_literal := **"none"** | **"wildcard"** | **"write"** | **"epsilon"** ;

C.2.4 Types

type := *primitive_type* | *domain_type* | *sequence_type* | *set_type* | *multiset_type* ;

primitive_type := **"Rational"** | **"Int"** | **"Bool"** | **"Perm"** | **"Ref"** ;

domain_type := *identifier* , **"["** , ((*type* | *identifier*) , **","**)* , (*type* | *identifier*)? , **"]"** ;

sequence_type := **Seq** , **"["** , *type* , **"]"** ;

set_type := **Set** , **"["** , *type* , **"]"** ;

multiset_type := **Multiset** , **"["** , *type* , **"]"** ;

C.2.5 Expressions

expression := *iff_expression* ;

iff_expression := *or_expression* , **"<==>"** , *iff_expression* | *or_expression* ;

or_expression := *equality_expression* , **"||"** , *or_expression* | *equality_expression* ;

equality_expression := *comparison_expression* , (**"=="** | **"!="**) , *equality_expression* | *comparison_expression* ;

comparison_expression := *sum_expression* , (**"<="** | **"<"** | **">"** |**">="** | **"in"**) , *comparison_expression* | *sum_expression* ;

sum_expression := *term_expression* , (**"++"** | **"+"** | **"-"** | **"union"** | **"intersection"** | **"setminus"** | **"subset"**) , *sum_expression* | *term_expression* ;

term_expression := *suffix_expression* , (**"*"** | **"/"** | **"\\"** | **"%"**) , *term_expression* | *suffix_expression* ;

suffix_expression := *atomic_expression* , *suffix*? ;

suffix := **"."** , *identifier* | **"[.."** , *expression* , **"]"** | **"["** , *expression* , **"..]"** | **"["** , *expression* , **".."** , *expression* , **"]"** | **"["** , *expression* , **"]"** | **"["** , *expression* , **":="** , *expression* , **"]"** ;

atomic_expression := *literal* | **"result"** | **"-"** , *sum_expression* |
"!" , *sum_expression* | **"+"** , *sum_expression* |
"(" , *expression* , **")"** | *function_application_expression* |
unfolding_expression | *folding_expression* | *perm_expression* |
old_expression | *labelled_old_expression* | *packing_wand_expression* |
applying_wand_expression ;

function_application_expression := *identifier* , **"("** , *actual_argument_list* , **")"** ;

actual_argument_list := (*expression* , **","**)* , *expression*? ;

unfolding_expression := **"unfolding"** , *access_assertion* , **"in"** ,
expression ;

folding_expression := **"folding"** , *access_assertion* , **"in"** , *expression* ;

perm_expression := **"perm"** , **"("** , *location_access* , **")"** ;

location_access := *field_access* | *predicate_access* ;

field_access := *atomic_expression* , *suffix*+ ;

predicate_access := *identifier* , **"("** , *actual_argument_list* , **")"** ;

old_expression := **"old"** , **"("** , *expression* , **")"** ;

labelled_old_expression := **"old"** , **"["** , *identifier* , **"]"** , **"("** , *expression* , **")"** ;

packing_wand_expression := **"packing"** ,
(**"("** , *magic_wand_assertion* , **")"** | *identifier*) , **"in"** , *expression* ;

applying_wand_expression := **"applying"** ,
(**"("** , *magic_wand_assertion* , **")"** | *identifier*) , **"in"** ,
expression ;

C.2.6 Assertions

assertion := *expression* | *access_assertion* | *conditional_assertion* |
separating_conjunction_assertion | *magic_wand_assertion* |
quantified_permission_assertion | *inhale_exhale_assertion* |
forperm_assertion ;

access_assertion := **"acc"** , **"("** , *location_access* , (**","** , *expression*)? ,
")" ;

conditional_assertion := *expression* , **"==>"** , *assertion* ;

separating_conjunction_assertion := *assertion* , **"&&"** , *assertion* ;

magic_wand_assertion := *assertion* , **"-*"** , *assertion* ;

quantified_permission_assertion := (**"forall"** | **"exists"**) ,
non_empty_formal_argument_list , **"::"** , **"{"** , *triggers* , **"}"** ,
assertion ;

non_empty_formal_argument_list := *formal_argument* , *formal_argument_list* ;

formal_argument := *identifier* , **":"** , *type* ;

formal_argument_list := (*formal_argument* , **","**)* , *formal_argument*? ;

triggers := (*trigger* , **","**)* , *trigger*? ;

trigger := *expression* ;

inhale_exhale_assertion := **"["** , *assertion* , **","** , *assertion* , **"]"** ;

forperm_assertion := **"forperm"** , **"["** , *identifiers* , **"]"** , *identifier* , **"::"** , *assertion* ;

identifiers := (*identifier* , **","**)* , *identifier*? ;

C.2.7 Statements

statement := *declaration_statement* | *new_statement* | *assignment_statement* | *label_statement* | *goto_statement* | *if_then_else_statement* | *while_statement* | *method_call_statement* | *assert_statement* | *assume_statement* | *inhale_statement* | *exhale_statement* | *fold_statement* | *unfold_statement* | *wand_package_statement* | *wand_application_statement* ;

declaration_statement := *variable_declaration_statement* | *define_declaration_statement* | *wand_declaration_statement* ;

variable_declaration_statement := **"var"** , *identifier* , **":"** , *type* , (**":="** , *expression*)? ;

define_declaration_statement := **"define"** , *identifier* , *assertion* ;

wand_declaration_statement := **"wand"** , *identifier* , **":="** , *assertion* ;

new_statement := *identifier* , **":="** , **"new"** , **"("** , (**"*"** | *identifiers*) , **")"** ;

assignment_statement := *variable_assignment_statement* | *field_assignment_statement* ;

variable_assignment_statement := *identifier* , **":="** , (*expression* | *method_call_statement*) ;

field_assignment_statement := *field_access* , **":="** , (*expression* | *method_call_statement*) ;

label_statement := **"label"** , *identifier* ;

goto_statement := **"goto"** , *identifier* ;

if_then_else_statement := **"if"** , **"("** , *expression* , **")"** , *block* , *else_if_else* ;

block := **"{"** , *statements* , **"}"** ;

statements := *statement** ;

else_if_else := *else_if* | *else* ;

else_if := **"elseif"** , **"("** , *expression* , **")"** , *block* , *else_if_else* ;

else := (**"else"** , *block*)? ;

while_statement := **"while"** , **"("** , *expression* , **")"** , *invariants* , *block* ;

invariants := (*invariant* , **";"**?)* , *invariant*? ;

invariant := **"invariant"** , *assertion* ;

method_call_statement := *identifier* , **"("** , *actual_argument_list* , **")"** ;

assert_statement := **"assert"** , *assertion* ;

assume_statement := **"assume"** , *assertion* ;

inhale_statement := **"inhale"** , *assertion* ;

exhale_statement := **"exhale"** , *assertion* ;

fold_statement := **"fold"** , *access_assertion* ;

unfold_statement := **"unfold"** , *access_assertion* ;

wand_package_statement := **"package"** , *magic_wand_assertion* ;

wand_application_statement := **"apply"** , *magic_wand_assertion* ;

C.2.8 Declarations

declaration := *define_declaration* | *domain_declaration* | *field_declaration* | *function_declaration* | *predicate_declaration* | *method_declaration* ;

define_declaration := *define_declaration_statement* ;

domain_declaration := **"domain"** , *identifier* , **"["** , *identifiers* , **"]"** , **"{"** , *function_signature** , *axiom_declaration** , **"}"** ;

function_signature := **"function"** , *identifier* , **"("** , *formal_argument_list* , **")"** , **":"** , *type* ;

axiom_declaration := **"axiom"** , *identifier* , **"{"** , *assertion* , **"}"** ;

field_declaration := **"field"** , *identifier* , **":"** , *type* ;

function_declaration := *function_signature* , *preconditions* , *postconditions* , **"{"** , *expression* , **"}"** ;

preconditions := (*precondition* , **";"**?)* , *precondition*? ;

precondition := **"requires"** , *assertion* ;

postconditions := (*postcondition* , **";"**?)* , *postcondition*? ;

postcondition := **"ensures"** , *assertion* ;

predicate_declaration := **"predicate"** , *identifier* , **"("** ,
formal_argument_list , **")"** , **"{"** , *assertion* , **"}"** ;

method_declaration := **"method"** , *identifier* , **"("** ,
formal_argument_list , **")"** , **"returns"** , *formal_argument_list* ,
preconditions , *postconditions* , *block*? ;

C.2.9 Program

program := *declaration** ;

Appendix D

Chalice: Concrete Syntax

This appendix defines the concrete syntax of Chalice by means of specifying a context-free grammar using an EBNF notation.

Contents

D.1 Notation

For an explanation of the notation used in the specification, refer to chapter C.1.

D.2 Concrete Syntax

D.2.1 Keywords

keyword := **"class"** | **"ghost"** | **"var"** | **"const"** | **"method"** |
"channel" | **"condition"** |**"assert"** | **"assume"** | **"new"** | **"this"** |
"reorder" | **"between"** | **"and"** | **"above"** | **"below"** | **"share"** |
"unshare" | **"acquire"** | **"release"** | **"downgrade"** | **"lock"** |
"fork" | **"join"** | **"rd"** | **"acc"** | **"credit"** | **"holds"** | **"old"** |
"assigned" | **"call"** | **"if"** | **"else"** | **"while"** | **"invariant"** |
"lockchange" | **"returns"** | **"requires"** | **"ensures"** | **"where"** |
"static" | **"int"** | **"bool"** | **"false"** | **"true"** | **"null"** | **"string"** |
"waitlevel" | **"lockbottom"** | **"module"** | **"external"** |
"predicate" | **"function"** | **"free"** | **"send"** | **"receive"** | **"ite"** |
"fold" | **"unfold"** | **"unfolding"** | **"in"** | **"forall"** | **"exists"** |
"seq" | **"nil"** | **"result"** | **"eval"** | **"token"** | **"empty"** | **"wait"** |
"signal" | **"unlimited"** | **"set"** | **"sum"** | **"max"** | **"refines"** |
"tracked" | **"transforms"** | **"replaces"** | **"by"** | **"spec"** | **"_"** | **"*"** ;

D.2.2 Types

type_declaration := (**"int"** | **"bool"** | *identifier* | **"string"** | **"seq"** | **"set"**) ,
(**"<"** , (*type* , **","**)* , *type*? , **">"**)? ;

D.2.3 Expressions

expression := *ite_expression* ;

expression_list := (*expression* , **","**)* , *expression*? ;

partial_expression_list := ((*expression* | **"_"**) , **","**)* , (*expression* | **"_"**)? ;

expression_body := **"{"** , *expression* , **"}"** ;

ite_expression := *iff_expression* , (**"?"** , *ite_expression* , **":"** , *ite_expression*)? ;

iff_expression := *impl_expression* , **"<==>"** , *iff_expression* ;

impl_expression := *logical_expression* , **"==>"** , *impl_expression* ;

logical_expression := *cmp_expression* ,
((**"&&"** , *cmp_expression*) | (**"||"** , *cmp_expression*))? ;

cmp_expression := *concat_expression* ,
(("==" | "!=" | "<=" | "<" | ">" | ">=" | "<<" | "in" | "!in") ,
concat_expression)? ;

concat_expression := *add_expression* , ("++" , *add_expression*)* ;

add_expression := *mult_expression* , (("+" | "-") , *mult_expression*)* ;

mult_expression := *unary_expression* , (("*" | "/" | "%") , *unary_expression*)* ;

unary_expression := ("!" , *unary_expression*) | ("-" , *unary_expression*) | *suffix_expression* ;

suffix_expression := *atom* , *suffix_thing** ;

atom := (*numeric_literal* | "false" | "true" | "null" | *string_literal* |
"waitlevel" | "lockbottom" | "this" | "result" |
back_pointer_member_access | *identifier* , ("(" , *expression_list* , ")")? |
"rd" , "(" , (((*identifier* , *read_perm_arg* , ")") |
((*atom* , ".")? , *back_pointer_member_access* , *read_perm_arg* , ")") |
(*select_expr_fer_sure_x* , *read_perm_arg* , ")")) |
(*read_perm_arg* , ")") | ("*" , ")")) |
"rd" , "*" , "(" , ((*identifier* , ")") |
((*atom* , ".")? , *back_pointer_member_access* , ")") |
(*select_expr_fer_sure_x* , ")")) |
"acc" , "(" , ((*identifier* , *access_perm_arg* , ")") |
((*atom* , ".")? , *back_pointer_member_access* , *access_perm_arg* , ")") |
(*select_expr_fer_sure_x* , *access_perm_arg* , ")")) |
"credit" , "(" , *expression* , ("," , *expression*)? , ")" |
"holds" , "(" , *expression* , ")" |
"rd" , "holds" , "(" , *expression* , ")" |
"rd" |
"assigned" , "(" , *identifier* , ")" |
"old" , "(" , *expression* , ")" |
"unfolding" , *suffix_expression* , "in" , *expression* |
"|" , *expression* , "|" |
"eval" , "(" , *eval_state* , "," , *expression* , ")" |
"ite" , "(" , *expression* , "," , *expression* , "," , *expression* , ")" |
"(" , *expression* , ")" |
quantifier_type | *quantifier_seq* | *quantifier_set* |
"[" , *expression* , ".." , *expression* , "]" |
"nil" , "<" , *type_declaration* , ">" |
"[" , *expression_list* , "]" |
"empty" , "<" , *type_declaration* , ">" |
"{" , *expression_list* , "}" ;

suffix_expression := *atom* , (*suffix_thing*)* ;

suffix_thing := **"["** , *expression* , **"]"** |
"[" , *expression* , **".."** , **"]"** |
"[" , **".."** , *expression* , **"]"** |
"[" , *expression* , **".."** , *expression* , **"]"** |
"." , *identifier* , (**"("** , *expression_list* , **")"**)? |
"." , *back_pointer_member_access* |
"." , **"acquire"** , *expr_body* |
"." , **"release"** , *expr_body* |
"." , **"fork"** , *call_target* , *partial_expression_list* , **")"** , *expr_body* ;

back_pointer_member_access := **"∼"** , **"("** , *identifier* , **"."** , *identifier* , **")"** ;

read_perm_arg := (**","** , **"*"** | **","** , *expression*)? ;

select_expr_fer_sure_x := *atom* , *ident_or_special* , (**"."** , *identifier*)⋆ ;

select_expr_fer_sure := *atom* , **"."** , *identifier* , (**"."** , *identifier*)⋆ ;

access_perm_arg := (**","** , *expression*)? ;

call_target := *identifier* , **"("** | *select_expr_fer_sure* , **"("** ;

D.2.4 Assertions

assertion := *expression* | *access_assertion* | *conditional_assertion* |
separating_conjunction_assertion | *magic_wand_assertion* |
quantified_permission_assertion | *inhale_exhale_assertion* |
forperm_assertion ;

access_assertion := **"acc"** , **"("** , *location_access* , (**","** , *expression*)? ,
")" ;

conditional_assertion := *expression* , **"==>"** , *assertion* ;

separating_conjunction_assertion := *assertion* , **"&&"** , *assertion* ;

magic_wand_assertion := *assertion* , **"-∗"** , *assertion* ;

quantified_permission_assertion := (**"forall"** | **"exists"**) ,
non_empty_formal_argument_list , **"::"** , **"{"** , *triggers* , **"}"** ,
assertion ;

non_empty_formal_argument_list := *formal_argument* , *formal_argument_list* ;

formal_argument := *identifier* , **":"** , *type* ;

formal_argument_list := (*formal_argument* , **","**)⋆ , *formal_argument*? ;

triggers := (*trigger* , **","**)⋆ , *trigger*? ;

trigger := *expression* ;

inhale_exhale_assertion := **"["** , *assertion* , **","** , *assertion* , **"]"** ;

forperm_assertion := **"forperm"** , **"["** , *identifiers* , **"]"** , *identifier* , **"::"** , *assertion* ;

identifiers := (*identifier* , **","**)* , *identifier*? ;

D.2.5 Statements

statement := *block_statement* | *assert_statement* | *assume_statement* | *local_variable_statement* | *call_statement* | *if_then_else_statement* | *while_statement* | *reorder_statement* | *share_statement* | *unshare_statement* | *acquire_statement* | *release_statement* | *lock_statement* | **"[["** , *block_statement_body* , **"]]"** | *rd_statement* | *downgrade_statement* | *free_statement* | *assignment_statement* | *fold_statement* | *unfold_statement* | *fork_statement* | *join_statement* | *wait_statement* | *signal_statement* | *send_statement* | *receive_statement* ;

block_statement := **"{"** , *block_statement_body* , **"}"** ;

block_statement_body := **"{"** , *statement** , **"}"** ;

assert_statement := **"assert"** , *expression* , **";"** ;

assume_statement := **"assume"** , *expression* , **";"** ;

local_variable_statement := *variable_specifier* , *variable_announcement* ;

variable_specifier := (**"var"** | **"spec"** | **"const"** | **"ghost"** , (**"const"** | **"var"**)) ;

variable_announcement := (*identifier_type* , **","**)* , *identifier_type* , **";"** | *identifier_type* , (**":="** , *rhs*)? , **";"** ;

identifier_type := *identifier* , (**":"** , *type_declaration*)? ;

rhs := **"new"** , *identifier* , (**"{"** , *field_init_list* , **"}"**)? , *install_bounds*? | *expression* ;

field_init_list := (*field_init* , **","**)* , *field_init*? ;

field_init := *identifier* , **":="** , *expression* ;

install_bounds := **"between"** , *expression_list* , **"and"** , *expression_list* | **"below"** , *expression_list* , (**"above"** , *expression_list*)? | **"above"** , *expression_list* , (**"below"** , *expression_list*)? ;

call_statement := **"call"** , *call_signature* , **";"** ;

call_signature := (*identifier_list* , **":="**)? , *call_target* , *expression_list* , **")"** , **";"** ;

call_target := *identifier* , **"("** | *select_expr_fer_sure* , **"("** ;

if_then_else_statement := **"if"** , **"("** , *expression* , **")"** , *block_statement* , (**"else"** , *else_statement*)? ;

else_statement := **"if"** , *if_then_else_statement* | *statement* ;

while_statement := **"while"** , **"("** , *expression* , **")"** , *loop_spec* ,
block_statement ;

loop_spec := *loop_spec_x** ;

loop_spec_x := **"invariant"** , *expression* , **";"** |
"lockchange" , *expression_list* , **";"** ;

reorder_statement := **"reorder"** , *expression* , *install_bounds*? , **";"** ;

share_statement := **"share"** , *expression* , *install_bounds*? , **";"** ;

unshare_statement := **"unshare"** , *expression* , **";"** ;

acquire_statement := **"acquire"** , *expression* , **";"** ;

release_statement := **"release"** , *expression* , **";"** ;

lock_statement := **"lock"** , **"("** , *expression* , **")"** , *block_statement* ;

rd_statement := **"rd"** , (**"lock"** , **"("** , *expression* , **")"** , *block_statement* |
"acquire" , *expression* , **";"** | **"release"** , *expression* , **";"**) ;

downgrade_statement := **"downgrade"** , *expression* , **";"** ;

free_statement := **"free"** , *expression* , **";"** ;

assignment_statement := *identifier* , **":="** , *rhs* , **";"** |
select_expr_fer_sure , **":="** , *rhs* , **";"** ;

fold_statement := **"fold"** , *expression* , **";"** ;

unfold_statement := **"unfold"** , *expression* , **";"** ;

fork_statement := **"fork"** , *call_signature* ;

join_statement := **"join"** , (*identifier_list* , **":="**)? , *expression* , **";"** ;

wait_statement := **"wait"** , *member_access* , **";"** ;

signal_statement := **"signal"** , **"forall"**? , *member_access* , **";"** ;

send_statement := **"send"** , *suffix_plus_expr* , (**"("** , *expression_list* , **")"**)? , **";"** ;

receive_statement := **"receive"** , (*identifier_list* , **":="**)? , *expression* , **";"** ;

D.2.6 Declarations

class_declaration := **"class"** , *identifier* , **"{"** , *member_declaration** , **"}"** ;

member_declaration := *field_declaration* | *invariant_declaration* |
method_declaration | *condition_declaration* | *predicate_declaration* |
function_declaration | *coupling_declaration* | *transform_declaration* ;

field_declaration := **"tracked"**? , **"ghost"**? , **"var"** , *identifier* , **":"** ,
type , **";"** ;

invariant_declaration := **"invariant"** , *expression* , **";"** ;

method_declaration := **"method"** , *identifier* , *formal_parameters* ,
(**"returns"** , *formal_parameters*)? , *method_spec** , *block_statement* ;

formal_parameters := **"("** , *formal_list*? , **")"** ;

formal_list := (*formal* , **","**)* , *formal*? ;

formal := *identifier* , (**":"** , *type_declaration*)? ;

method_spec := **"requires"** , *expression* , **";"** |
"ensures" , *expression* , **";"** |
"lockchange" , *expression_list* , **";"** ;

condition_declaration := **"condition"** , *identifier* , (**"where"** , *expression*)? , **";"** ;

predicate_declaration := **"predicate"** , *identifier* , **"{"** , *expression* , **"}"** ;

function_declaration := **"unlimited"**? , **"static"**? , **"function"** , *identifier* ,
formal_parameters , **":"** , *type_declaration* , *method_spec** , (**"{"** , *expression* , **"}"**)? ;

channel_declaration := **"channel"** , *identifier* , *formal_parameters* ,
(**"where"** , *expression*)? , **";"** ;

D.2.7 Program

program := (*class_declaration* | *channel_declaration*)* ;

Bibliography

[1] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," tech. rep., ETH Zurich, 2014.

[2] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with chalice," in *Foundations of Security Analysis and Design V*, pp. 195–222, Springer, 2009.

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

[4] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *International Workshop on Computer Science Logic*, pp. 1–19, Springer, 2001.

[5] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *European Conference on Object-Oriented Programming*, pp. 148–172, Springer, 2009.

[6] K. R. M. Leino, "This is boogie 2," in *Manuscript KRML 178*, p. 131, 2008.

[7] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *European Symposium on Programming*, pp. 125–128, Springer, 2013.

[8] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[9] A. J. Summers and S. Drossopoulou, "A formal semantics for isorecursive and equirecursive state abstractions," in *European Conference on Object-Oriented Programming*, pp. 129–153, Springer, 2013.

[10] K. R. M. Leino and P. Müller, "A basis for verifying multi-threaded programs," in *European Symposium on Programming*, pp. 378–393, Springer, 2009.

[11] M. Schwerhoff and A. J. Summers, *Lightweight support for magic wands in an automatic verifier*, vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[12] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Abstract read permissions: Fractional permissions without the fractions," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 315–334, Springer, 2013.

[13] G. Huet, G. Kahn, and C. Paulin-Mohring, "The coq proof assistant a tutorial," in *Rapport Technique 178*, 1997.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

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 Formal Semantics for Viper

Authored by (in block letters):

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

Name(s):	**First name(s):**
Gössi	**Cyrill Martin**

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)**
Zurich, 24 November 2016	

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